Revista Cubana de Ciencias Informáticas
Vol. 10, No. Especial “Informtica 2016”, 2016
ISSN: 2227-1899 | RNPS: 2301
http://rcci.uci.cu
Pág. 1-14

Tipo de artículo: Artículo original
Temática: Software libre
Recibido: 15/04/2016 | Aceptado: 05/05/2016

XEOS para el desarrollo de software base de sistemas embebidos

XEOS for base software development on embedded systems

Reinier Millo-Sánchez[1*], Waldo Paz-Rodríguez[1], Alexy Gallardo-Segura[2], Humberto López-León[2]

[1]Universidad Central “Marta Abreu” de Las Villas, Santa Clara, Cuba
[2]Empresa de Tecnologías de la Información para la Defensa, La Habana, Cuba

[*]Autor para correspondencia: rmillo@uclv.cu

Resumen
El empleo de sistemas embebidos se ha convertido en un fenómeno común de nuestros días. Podemos encontrar dispositivos embebidos desde las fábricas automatizadas hasta en nuestro hogar. Aunque el funcionamiento de los sistemas embebidos es predeterminado por la funcionalidad para la cual se van a emplear, es necesario tener un sistema operativo que controle el dispositivo. El desarrollo de sistemas operativos para controlar estos sistemas embebidos tiene nuevos retos con el auge de nuevas plataformas de *hardware* y nuevos requisitos de sistema. Dos de los enfoques más empleados para el desarrollo de *kernels* o núcleo de sistema son el monolítico y el *microkernel*. El enfoque basado en *microkernel* tiene un enfoque minimalista y parece ser una mejor alternativa para el desarrollo de sistemas embebidos. En este trabajo se comparan los enfoques de *kernel* monolítico y *microkernel*, resaltando las ventajas que ofrece el enfoque basado en *microkernel*. Se propone XEOS, una combinación del *microkernel* Fiasco.OC con el *framework* de desarrollo GenodeOS, para el desarrollo de un *software* base para ser empleado en sistemas embebidos.

Palabras claves: microkernel, sistemas embebidos, XEOS, Fiasco.OC, GenodeOS

Abstract
Use of embedded systems has become a common phenomenon today. We can find embedded devices from automated factories to even our home. Although the functioning of embedded systems is predefined by the functionality for which is to be used, it needs an operating system to control the device. The development of operating systems to control this embedded systems has new challenges with the apparition of new hardware platforms and new system requirements. Two of the most used approaches to develop system kernels are monolithic and microkernel. The microkernel-based approach takes a minimalist approach and seems to be a better alternative for the development of embedded systems. This paper compare approaches microkernel and monolithic kernel, highlighting the advantages of the microkernel-based approach. It propose XEOS, a combination of Fiasco.OC microkernel with GenodeOS development framework for developing a base software to be used in embedded systems.

Keywords: *microkernel, embbeded systems, XEOS, Fiasco.OC, GenodeOS*

Revista Cubana de Ciencias Informáticas
Vol. 10, No. Especial "Informtica 2016", 2016
ISSN: 2227-1899 | RNPS: 2301
http://rcci.uci.cu
Pág. 1-14

Introducción

El empleo de dispositivos embebidos ha tenido un auge en los últimos años. Su empleo en el control automático de procesos de producción en fábricas se ha ido extendiendo hasta ser un elemento común en nuestros hogares. Estos dispositivos embebidos o empotrados, como también se les conoce, es común observarlos ya presentes en lavadoras de control automático, en sistemas decodificadores para la televisión, en hornos, ventiladores, sistemas de alumbrado automático, reproductores DVD, entre otros muchos dispositivos del hogar.

Los sistemas embebidos no son más que un sistema de computación cuyo *hardware* y *software* están específicamente diseñados y optimizados para resolver un problema concreto de forma eficiente. El término embebido o empotrado hace referencia al hecho que la electrónica o el sistema electrónico de control es una parte fundamental del sistema donde se emplea. La característica principal que diferencia a los embebidos de los demás sistemas electrónicos, es que, por estar insertados dentro del dispositivo que controlan, están sujetos en mayor medida a cumplir requisitos de tamaño, fiabilidad, consumo y coste, y su existencia puede no ser aparente (Tanenbaum, 2009; Silberschatz et al., 2013).

Estos sistemas son empleados también en aplicaciones donde es necesario la respuesta en tiempo real ante ciertos eventos, como es el caso de equipos médicos en los hospitales y sistemas de radares tanto marítimos como aéreos. Varios han sido los enfoques empleados para el desarrollo de sistemas operativos para el control de los sistemas embebidos (Tanenbaum, 2009).

En la actualidad existen varios sistemas operativos de propósito general, centrados principalmente en dos grandes familias: Microsoft Windows y GNU/Linux. Estos sistemas en su mayoría están formados por un núcleo o *kernel* monolítico, el cual concentra todas las funcionalidades básicas del sistema operativo: gestión de los procesos, manipulación del sistema de archivos, control de los dispositivos de *hardware*, gestión de memoria, entre otras.

Por otro lado, existen otros tipos de *kernel* llamados *microkernel* que tienen una concepción minimalista. Como plantea Liedtke (1995) el *microkernel* sólo debe implementar los elementos básicos o primitivas para el funcionamiento del sistema, el resto de los elementos deben ser implementados a nivel de aplicaciones de usuarios. Esta concepción permite tener un sistema más flexible, modular y tolerable a fallos.

El *kernel* del sistema es la capa inferior de un sistema operativo y es el que interactúa directamente con el *hardware*, proveyendo interfaces de trabajo hacia las capas superiores. Actualmente todo el proceso de desarrollo no se realiza de forma manual, para ello se han desarrollado un conjunto de marcos de trabajo o *frameworks* que permiten agilizar el proceso de desarrollo.

Estos se centran principalmente en las etapas de desarrollo, compilación y prueba. En su mayoría tienen una estructura bien definida lo que muchas veces facilita crear aplicaciones muy específicas, modulares y escalables. Además cuentan con un conjunto de bibliotecas que facilitan el desarrollo de nuevas aplicaciones e incluso reutilizar las existentes como base, logrando agilizar el proceso de desarrollo de estos sistemas.

En este trabajo se hace un estudio sobre los enfoques de desarrollo del *kernel* del sistema y se propone un conjunto de tecnologías, *kernel* y *framework* de desarrollo, para el desarrollo de *software* base de sistemas embebidos.

Enfoques de desarrollo de *kernel* del sistema operativo

Como se explicó anteriormente el *kernel* es el elemento principal para el funcionamiento del sistema operativo, por lo que hay tener cuidado durante todo el proceso de desarrollo. Las tendencias o enfoques empleados para el desarrollo del *kernel* han ido variando con el desarrollo de la propia tecnología. Actualmente existen cuatro grandes enfoques bien marcados para su desarrollo: *kernel* monolítico, *microkernel*, *kernel* híbrido y *exokernel*.

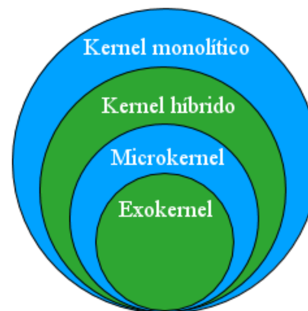

Figura 1. Relación de tamaño del TCB en cada uno de los enfoques de desarrollo de *kernel* del sistema

Estos enfoques difieren entre sí por la forma en que se organizan y estructuran las funcionalidades básicas del sistema, tanto dentro como fuera del *kernel*. Esta organización se refleja directamente en el tamaño del *kernel* (en líneas de código) en cada uno de los enfoques y por tanto en el TCB (*Trusted Computing Base*) del sistema. Los enfoques de *microkernel* y *exokernel* reducen el TCB de sistema por su tamaño reducido, como se muestra en la figura 1.

Los *kernels* monolíticos y los *microkernels* han sido los dos enfoques más empleados en el desarrollo de *kernel* para sistemas embebidos (Tanenbaum, 2009). A partir de estos dos enfoques, ha tomado auge el desarrollo de *kernels* híbridos, tratando de aprovechar las ventajas de los enfoques anteriores. Los *exokernels* por su características solo han sido empleado en aplicaciones muy específicas.

Kernel monolítico

Un *kernel* monolítico es definido por (Tanenbaum, 2009; Silberschatz et al., 2013) como una colección de procedimientos enlazados entre sí en un único programa binario. Hasta ahora el enfoque monolítico para el diseño de *kernel* se considera como el más utilizado en la construcción de los mismos. En este enfoque todas las funcionalidades básicas del sistema se ejecutan como un solo programa en modo *kernel*. Con esta técnica cada procedimiento del sistema puede llamar a cualquier otro y solicitar algún servicio de forma directa. Como el sistema tiene miles de procedimientos y estos se pueden llamar entre sí sin restricciones esto puede hacer que el sistema sea difícil de comprender (Tanenbaum, 2009).

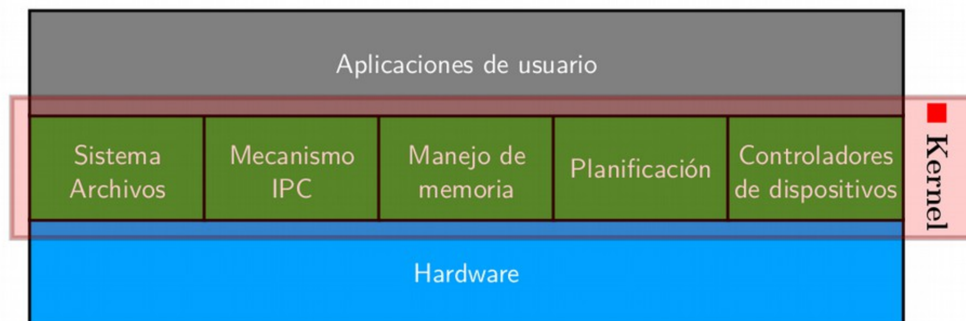

Figura 2. Estructura de un sistema basado en *kernel* monolítico

A pesar de esto, el diseño de los *kernels* monolíticos es más sencillo que el de los *microkernels* y los otros enfoques (Tanenbaum, 2009). Los *kernels* monolíticos se implementan en su totalidad como un único proceso que se ejecuta en un solo espacio de direcciones. Por lo general en el disco se pueden ver como un único binario estático. En la figura 2 se pueden observar los diferentes servicios o abstracciones de *hardware* que brinda este enfoque y los cuales se ejecutan en el gran espacio de direcciones del *kernel* (Love, 2010).

La comunicación dentro del *kernel* es trivial, como todos los servicios se ejecutan en modo *kernel* y dentro del mismo espacio de direcciones: el *kernel* puede invocar funciones directamente sin necesidad de realizar un cambio de contexto. Los defensores de este modelo citan la simplicidad y rendimiento del enfoque monolítico. La mayoría de los sistemas Unix tienen un diseño monolítico.

A diferencia de otros enfoques de desarrollo de *kernel* del sistema, los *kernels* monolíticos son más fáciles de diseñar correctamente y el código para el control del sistema es mucho más sencillo ya que todos los elementos se encuentran en un mismo espacio de direcciones, el espacio de direcciones del *kernel*.

La implementación más conocida de *kernel* monolítico es el *kernel* de Linux. Este desde sus inicios es un *kernel*

monolítico que ha ido evolucionando y actualmente aunque es modular, o sea, extensible mediante módulos cargables en tiempo de ejecución, continúa siendo un *kernel* monolítico (Bovet and Cesati, 2000). Todos los elementos cargados en tiempo de ejecución a través de módulos, así como los controladores de dispositivos siguen ejecutando en el modo privilegiado, lo cual no permite garantizar la integridad completa del sistema. A pesar de ello, ha sido muy empleado en sistemas embebidos (Suppiah and Abbas, 2014) y sistemas de tiempo real (Dozio and Mantegazza, 2003). Varios han sido los estudios y trabajos realizados con el objetivo de mejorar la seguridad en estos, en algunos casos haciendo empleo de la virtualización (Govindharajan, 2013).

Microkernel

A diferencia de los *kernels* monolíticos, los *microkernels* tienen un enfoque de desarrollo minimalista, donde se debe implementar a nivel de *kernel* solo los elementos principales que son nombrados conceptos o primitivas. En Liedtke (1995) se definen cuatro primitivas necesarias y suficientes para el correcto desarrollo de un *microkernel*: espacio de direcciones, abstracción de procesos, mecanismo de comunicación e identificadores únicos, como se muestra en la figura 3. Este enfoque busca establecer una separación entre los mecanismos y las políticas del sistema operativo.

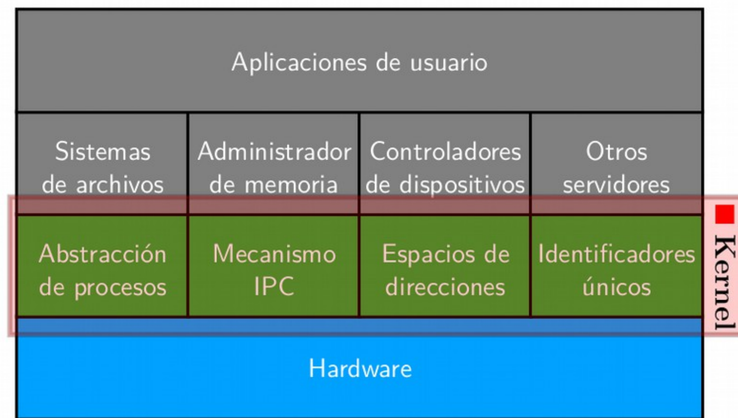

Figura 3. Estructura de un sistema basado en *microkernel*

El autor plantea que la definición de las primitivas está dada por la funcionalidad y no por el rendimiento del sistema. Liedtke considera una primitiva aquel elemento que implementado correctamente fuera del *kernel* no permite la implementación de las funcionalidades requeridas del sistema.

Actualmente la política de planificación de la CPU es la única política que aún se implementa a nivel de *microkernel*, buscando garantizar la seguridad del sistema, aunque existen algunos trabajos en los cuales se ha

incursionado en la planificación a nivel de usuario (Zoor and Nagibin, 2014). Las políticas para el manejo de la memoria son implementadas en servidores a nivel de usuario, permitiendo la existencia de varios servidores para la manipulación de la memoria sin que estos entren en conflicto entre sí (Klimiankou, 2014). De igual forma los controladores de dispositivos son implementados a nivel de usuario, implementándose a nivel de *microkernel* solo los elementos mínimos e indispensables para multiplexar los recursos de forma segura.

Adicionalmente a las primitivas, el desarrollo de un *microkernel* se rige por dos principios básicos, como se plantea en Liedtke (1995), el "principio de independencia" y el "principio de integridad", este último nombrado también "principio básico de la comunicación". El principio de independencia expresa que debe ser posible implementar un subsistema o aplicación A_1 arbitrario de forma tal que este no pueda ser perturbado o corrompido por otro subsistema A_2. Este principio debe dar garantía de independencia de A_1 respecto a cualquier otro subsistema A_2.

Varios son los *kernel* de sistema que se han diseñado siguiendo este enfoque de *microkernel* desde sus orígenes en los años 80, entre los que podemos encontrar: Mach (Accetta et al., 1986), FreeRTOS (Goyette, 2007), QNX (Hildebrand, 1992), GNU/Hurd (Le Mignot, 2005), Minix (Woodhull and Tanenbaum, 1997), Fiasco.OC (PartheymÃ¼ller et al., 2012), OKL4 (Heiser and Leslie, 2010), seL4 (Klein et al., 2010). Este último siendo el primer *kernel* de sistema al cual se le realiza una verificación formal de su especificación.

Kernel monolítico vs. *microkernel*

El *kernel* monolítico libre más empleado hoy en día es el *kernel* de Linux. Su arquitectura modular, portabilidad y soporte para gran variedad de dispositivos de *hardware* han permitido su empleo desde sistema de propósito general hasta sistemas embebidos (Raghavan et al., 2006), como es el caso más conocido en el sistema operativo Android (Faruki et al., 2015).

A diferencia del *kernel* de Linux, los *microkernels* en su enfoque minimalista disminuyen el tamaño en LOC (*Lines of Code*) del TCB, como se muestra en la figura 4. A partir de la versión 3.0 el *kernel* de Linux alcanza un aproximado de 2.5 millones de LOC (Wheeler, 2004) sin incluir los controladores de dispositivos de *hardware*, mientras que los *microkernels* están en el orden de los miles de LOC, por ejemplo Fiasco.OC está en aproximadamente 35 mil LOC (Wheeler, 2004), lo cual es una diferencia significativa si se tiene en cuenta que cada mil LOC es posible encontrar al menos un error o vulnerabilidad (Vemuri and Al-Hamdani, 2011). Como se puede ver en la figura, muchos de los elementos que se encontraban a nivel de *kernel* en el enfoque monolítico, ahora en el *microkernel* se encuentran a nivel de usuario, con lo cual se logra reducir drásticamente el tamaño del *kernel* y por tanto el tamaño del TCB.

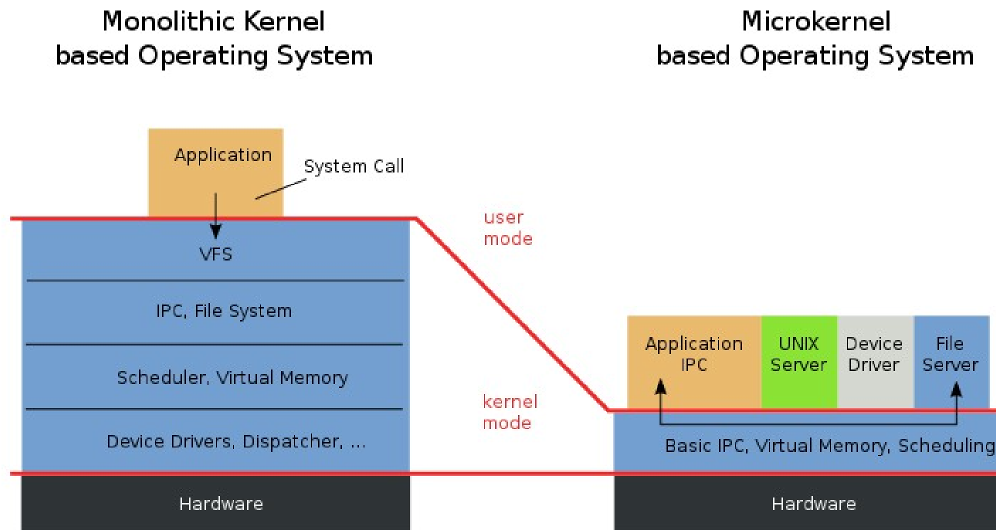

Figura 4. TCB de un sistema basado en *kernel* monolítico y *microkernel*

Una de las principales ventajas de los *microkernels* sobre los *kernels* monolíticos es la extensibilidad del sistema. En un sistema que emplea un *microkernel* se pueden agregar nuevas funcionalidades de sistema sin afectar el funcionamiento del mismo y sin necesidad de ser modificado y recompilado, presto que las funcionalidades son implementadas como aplicaciones a nivel de usuario y se interactúa con ellas a través del mecanismo de comunicación. Por otra parte en los sistemas de *kernel* monolítico, agregar nuevas funcionalidades de sistema puede poner en riesgo el funcionamiento del sistema y es necesario la modificación del algunos subsistemas, así como la recompilación del *kernel* del sistema para el empleo de la nueva funcionalidad.

Otra de las principales diferencias entre estos dos enfoques de diseño de *kernel* es la recuperación rápida ante fallos, fundamentalmente en los controladores de dispositivos de *hardware*. En el *kernel* monolítico la mayoría de los controladores funcionan a nivel de *kernel*, y un controlador defectuoso puede hacer fallar el sistema en su totalidad. A diferencia de lo anterior, los *microkernels* implementan los controladores a nivel de usuario, por lo que un controlador defectuoso no afecta el funcionamiento del sistema.

Estos dos enfoques también difieren en la forma en que las aplicaciones de modo usuario interactúan con las funcionalidades implementadas a nivel de *kernel*. En el enfoque monolítico, esta interacción se realiza a través de las llamadas al sistema, proceso en el cual se cambia el modo de ejecución del modo usuario al modo *kernel*. A diferencia de este, en el enfoque de *microkernel*, esta interacción se realiza a través del mecanismo de comunicación enviando mensajes al *microkernel*. Por ejemplo, en la figura 5 se muestra una aplicación haciendo

una llamada a un controlador de un dispositivo de *hardware*. Como se puede ver en el caso del *microkernel* es necesario realizar dos cambios de contexto o modo de ejecución, mientras en el *kernel* monolítico es suficiente con un sólo cambio.

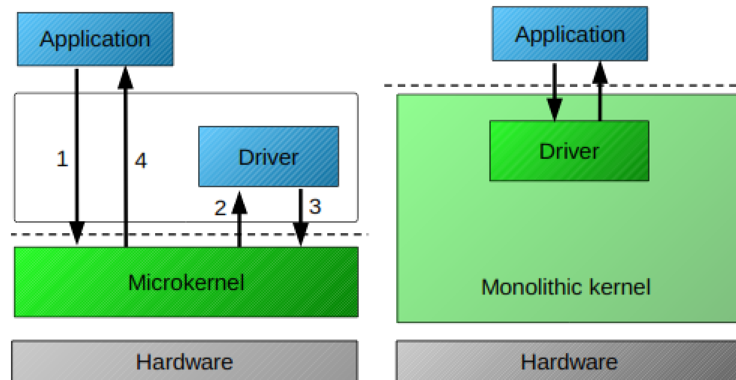

Figura 5. Interacción entre las aplicaciones de usuario y el *kernel* del sistema

Este ha sido un aspecto crítico en el desarrollo de los *microkernels*, pero Liedtke (1995, 1996) demostró que esto no era más que el resultado de un mal diseño y una mala implementación, lo cual no afecta considerablemente el rendimiento del sistema.

Ambos enfoques de diseño de *kernel* han sido empleados para el desarrollo de sistemas embebidos, pero el enfoque basado en *microkernel* brinda un mejor aprovechamiento del *hardware*. A diferencia de los *kernels* monolíticos, los *microkernels* no proveen una capa de abstracción para la interacción con el *hardware*, como el caso de la CPU, estos son implementados de forma específica y optimizada para explotar al máximo todas las funcionalidades del *hardware* y alcanzar altos niveles de rendimiento.

XEOS para el desarrollo de *software* base de sistemas embebidos

Teniendo en cuenta las características del enfoque de *microkernel* y la necesidad de dominar una tecnología de *software* base en su totalidad, para garantizar una soberanía tecnológica, se propone XEOS (*XETID Embedded Operating System*) como *software* base empleando tecnología basada en *microkernel*. Se propone el empleo del *microkernel* Fiasco.OC y *framework* de desarrollo GenodeOS, por las características de estas dos tecnologías que se exponen a continuación.

El *microkernel* Fiasco.OC fue desarrollado en la Universidad de Dresden y está liberado bajo la licencia GNU/GPL v2. Pertenece a la tercera generación de *microkernels* y está basado en el *microkernel* Fiasco de

la familia L4, al cual añade un diseño basado en capacidades de objetos. La mayor parte del código fuente está desarrollada en C/C++ con un dialecto de preprocesamiento. Además cuenta con una lista de distribución que permite la interacción con una comunidad activa de desarrolladores, lo cual beneficia el intercambio científico y a la vez sirve como sistema de consulta, para reportar y conocer la existencia de *bugs* o errores, así como sus posibles soluciones, entre otros.

Fiasco.OC utiliza sincronización no bloqueante para la comunicación entre los objetos del *kernel*. Es un *kernel* multitarea con soporte para múltiples procesadores. Ha sido portado para la arquitectura x86 y ARM en varias versiones y plataformas de *hardware*.

Este *microkernel* puede ser empleado para la virtualización (Härtig et al., 2008) asistida por *hardware* en AMD SVM e Intel VT. También provee soporte para la paravirtualización. Además Fiasco.OC tiene soporte para la planificación de tareas e hilos de ejecución en tiempo real, con un soporte de prioridades, lo cual permite que pueda ser empleado en sistemas de tiempo real.

De forma general el *microkernel* implementa las abstracciones y mecanismos propuestos en Liedtke (1995), centrados en el manejo de la memoria, la planificación de las tareas y la comunicación entre los procesos y servicios del sistema. En Fiasco.OC todo es considerado un objeto, dividiendo las abstracciones implementadas en siete objetos, también llamados objetos de *kernel*.

Para el desarrollo de sistema embebidos empleando Fiasco.OC como *microkernel* existen dos *frameworks* de desarrollo, L4Re (*L4 Runtime Environment*) (TU-DresdenOSGroup, 2015) y GenodeOS (Feske, 2014). L4Re provee a Fiasco.OC de un conjunto de bibliotecas y servidores para el desarrollo de nuevas aplicaciones. Provee un paginador inicial (Sigma0) y un cargador de aplicaciones (Moe).

Estos dos elementos permiten la ejecución de las aplicaciones desarrolladas sobre Fiasco.OC. Inicialmente L4Re surgió como un proyecto para la validación de Fiasco.OC y se ha ido consolidando como *framework* para el desarrollo de aplicaciones. Actualmente posee soporte para la biblioteca estándar de C/C++, compresión Zlib, bibliotecas para el trabajo con aplicaciones de interfaces de usuario, soporte para las bibliotecas OpenMP, PThread y SDL, entre otros.

GenodeOS es un *framework* que posee un conjunto de herramientas para la construcción y personalización de sistemas operativos de propósitos específicos, aunque el proyecto pretende convertirse en una herramienta para el desarrollo de sistemas operativos de propósito general (Feske, 2014). El *framework* brinda un conjunto de componentes tales como controladores de dispositivos, protocolos, aplicaciones de terceros y soporte para varios *kernels* de sistema.

A diferencia de otros *frameworks* de desarrollo que se integran de forma específica a un *kernel*, como es el caso del L4Re, GenodeOS tiene soporte para un conjunto diverso de *kernels*, entre ellos el *kernel* de Linux y una variedad de *microkernels*. El empleo de GenodeOS como *framework* de desarrollo permite a los desarrolladores abstraerse por completo del *kernel* del sistema que se esté empleando. Esto permite que una aplicación desarrollada sobre GenodeOS pueda ser ejecutada sobre varios *kernels* del sistema sin necesidad de ser modificada.

Actualmente GenodeOS tiene soporte para ocho *kernels* diferentes: OKL4, L4/Fiasco, L4ka::Pistachio, NOVA, Fiasco.OC, Codezero, Linux y más reciente el seL4. Además de estos *kernels* GenodeOS tiene su propio *kernel* (*bare-metal*) (Feske, 2014). Esta independencia del *kernel* le ha permitido al *framework* adquirir ciertas características de otros *kernels* como es el soporte para virtualización de sistemas como L4Linux y OKLinux de Fiasco.OC y OKL4, respectivamente.

El diseño de GenodeOS se rige por varios principios arquitectónicos que permiten darle al sistema una estructura modular, flexible y escalable:

- Principio de la computación segura: este principio se basa en reducir el número de dependencias a bajo nivel cuando se ejecuta determinada aplicación, de forma que si dicha aplicación falla, no haga colapsar el sistema. Se debe garantizar que cada proceso debe funcionar lo más aislado posible del resto de los procesos.
- Principio de privilegios mínimos: este principio se basa en la comunicación segura entre procesos y grupos de proceso como se plantea en Liedtke (1992, 1995). Para ello los procesos se agrupan siguiendo la estructura en forma de árbol, y la comunicación es gestionada y controlada por los nodos padres. Este principio también es aplicado a la gestión de recursos y memoria.

Para garantizar la seguridad y estabilidad del sistema GenodeOS tiene una estructura jerárquica en forma de árbol. Cada nodo del árbol representa un proceso en ejecución. Cuando un proceso es creado se le asignan una serie de recursos que puede manejar. Estos recursos asignados son recursos que pertenecen a su nodo padre. De igual forma los procesos hijos de este nodo, sólo podrán ser creados con los recursos que le fueron asignados al nodo. El proceso inicial del sistema tiene acceso a todos los recursos del sistema, y a partir de ese nodo se comienza la ramificación de los procesos.

Esta estructura jerárquica permite un control seguro sobre la comunicación entre los procesos y el acceso a los servicios que proveen cada uno de los sistemas. De esta forma GenodeOS logra disminuir el TCB de una aplicación, reduciéndolo solamente a los nodos que componen la rama que va desde el nodo de la aplicación hasta la raíz del árbol.

XEOS propone una estructura como se muestra en la figura 6, donde se empleen las tecnologías base (Fiasco.OC+GenodeOS) con soporte para diferentes plataformas de *hardware*. Se propone continuar extendiendo el *framework* de desarrollo para agregarle nuevas funcionalidades que representen requisitos para los diferentes escenarios en que se emplee XEOS, así como mecanismos de seguridad y APIs de programación que permitan el desarrollo de aplicaciones sobre la tecnología base.

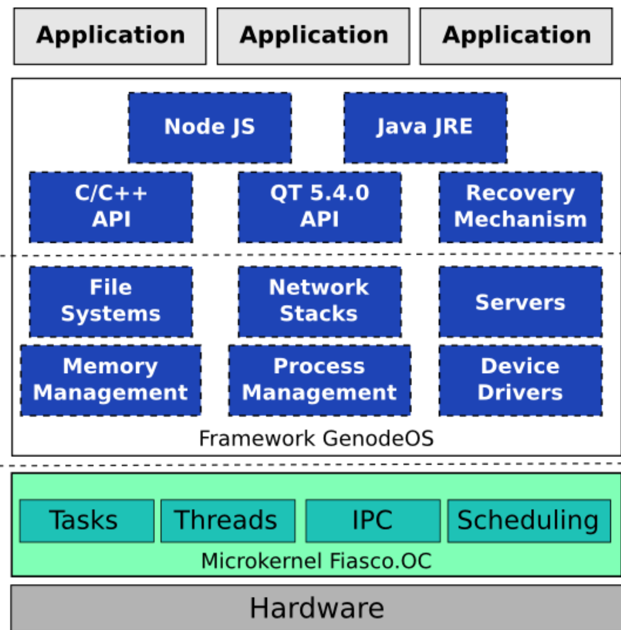

Figura 6. Estructura de XEOS

XEOS ha sido portado y probado con éxito en las plataformas de *hardware* RaspberryPI y Odroid-X2. Este soporte ha sido incorporado a la distribución oficial de ambas tecnologías a través de las comunidades de desarrollo. Actualmente se trabaja en el soporte para controladores de dispositivos de *hardware* y el soporte para QT 5.4.0.

Bajo el precepto de soberanía tecnológica y las características de las tecnologías que componen a XEOS, se pretende que este sea empleado por el país para el desarrollo de sistemas embebidos. Actualmente se trabaja en probar XEOS en un recolector de datos industriales que está siendo desarrollado en conjunto por la XETID y la empresa Serconi. También puede ser empleado para el desarrollo industrial del país y el desarrollo de la robótica en general.

Conclusiones

En este trabajo se describieron los enfoques más empleados en la actualidad para el desarrollo de *kernels* de sistemas operativos: *kernel* monolítico y *microkernel*. Estos enfoques difieren en cuanto a la organización interna de las funcionalidades del sistema. Se realizó una comparación entre el enfoque de desarrollo monolítico y *microkernel*, donde los *microkernels*:

- Tiene un tamaño reducido del TCB del sistema (en el orden de los miles de líneas de código).
- Tiene un alto grado de extensibilidad, ya que se pueden agregar nuevas funcionalidades de sistema sin afectar el funcionamiento del mismo y sin necesidad de ser modificado y recompilado, a diferencia del enfoque monolítico.
- Tiene mayor estabilidad y posibilidades de recuperación ante fallos, solo las primitivas son implementadas a nivel de *kernel*, el resto de los componentes se implementan como aplicaciones a nivel de usuario, por lo que una aplicación con un mal funcionamiento no compromete la integridad del sistema.
- Hacen un uso más eficiente del *hardware* subyacente.

Se propone XEOS como un conjunto de tecnologías para el desarrollo de *software* base de sistemas embebidos. XEOS basado en el *microkernel* Fiasco.OC y el *framework* de desarrollo GenodeOS ha sido portado a varias plataformas de *hardware*. Este soporte ha sido validado y aceptado por la comunidad de desarrollo internacional de estas tecnologías.

Se deben continuar implementado componentes de seguridad sobre las tecnologías y realizar pruebas de rendimiento sobre el sistema para determinar los tiempos de respuesta del mismo. También se debe trabajar en función de implementar algunas pruebas que permitan medir las capacidad de *RealTime*, así como implementar algoritmos de planificación *RealTime*.

Referencias

M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for unix development. In *Proceedings Usenix Summer'86 Conference*, pages 93–113, Atlanta, Georgia, 1986.

Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 1ed edition, 2000.

Lorenzo Dozio and Paolo Mantegazza. Linux real time application interface (rtai) in low cost high performance motion control. *Motion Control*, 2003(1):1–15, 2003.

P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M.S. Gaur, M. Conti, and M. Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys & Tutorials*, 17(2): 998–1022, May 2015.

Norman Feske. Genode as general-purpose os - progress report and demonstration. In *Free and Open Source Software Developers European Meeting*, 2014.

Hariprasad Govindharajan. Porting linux to a hypervisor based embedded system. Technical report, Institutionen för informationsteknologi, Department of Information Technology, Uppsala Universitet, 2013.

Rich Goyette. An analysis and description of the inner workings of the freertos kernel. *Operating System Methods for Real-Time Applications*, 2007.

Hermann Härtig, Michael Roitzsch, Adam Lackorzynski, Börn Döbel, and Alexander Böttcher. L4-virtualization and beyond. In *Korean Information Science Society Review*, 2008.

Gernot Heiser and Ben Leslie. The okl4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the First ACM Asia-Pacific Workshop on Systems*, pages 19–23, New Delhi, India, August 2010. ACM.

Dan Hildebrand. An architectural overview of qnx. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, and Michael Norrish. sel4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, June 2010. The original version of this paper was published in the Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles, Oct. 2009.

Yauhen Klimiankou. An enhanced multi-pager environment support for second generation microkernels. *International Journal of Computer Science and Information Security*, 12(1), 2014.

Gaël Le Mignot. The gnu hurd. In *Extended Abstract of Talk at Libre Software Meeting, Dijon, France*, 2005.

Jochen Liedtke. Clans & chiefs. In *Proceedings 12th GI/ITG Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, 1992. Springer-Verlag.

Jochen Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, December 1995.

Jochen Liedtke. Towards real microkernels. *Comunications of the ACM*, 39(9):70–77, 1996.

Robert Love. *Linux Kernel Development*. Addison-Wesley, 3ed edition, 2010.

M. PartheymÃ¼ller, J. Stecklina, and B. Döbel. Fiasco.oc on the scc. In *Proceedings of the 4th Many-Core Applications Research Community Symposium*, pages 79–82, 2012.

P. Raghavan, Amol Lad, and Sriram Neelakandan. *Embedded Linux System Design and Development*. Auerbach Publications, 2006.

Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, 2013.

R. Suppiah and M.F. Bin Abbas. Introducing embedded systems development on a robotics-based platform. In *International Conference on Teaching, Assessment and Learning*, pages 103–108, Wellington, Dec. 2014. IEEE.

Andrews S. Tanenbaum. *Sistemas Operativos Modernos*. Prentice-Hall, 2009.

TU-DresdenOSGroup. L4re - l4 runtime environment. *(En: http://os.inf.tu-dresden.de/l4re)*, 2015.

Durga Vemuri and Wasim Al-Hamdani. Measures to improve security in a microkernel operating system. In *Proceedings of the Information Security Curriculum Development Conference*, pages 25–33. ACM, 2011.

David A. Wheeler. Sloccount (version 2.26). *(En: http://www.dwheeler.com/sloccount.)*, 2004.

Andrew S Woodhull and Andrew S Tanenbaum. *Operating Systems Design and Implementation*. Prentice-Hall, 1997.

Andreas Zoor and Nikolai Nagibin. User-level scheduling mechanisms. In Robert Kaiser, editor, *First Wiesbaden Workshop on Advanced Microkernel Operating Systems*, pages 25–28. ACM, Februrary 2014.