

Un Modelo de Arquitectura para un Sistema de Virtualización Distribuido

Pablo Andrés Pessolani

Tesis presentada para obtener el grado de Doctor en Ciencias Informáticas



UNIVERSIDAD NACIONAL DE LA PLATA
FACULTAD DE INFORMÁTICA

Mayo 2018

Directores:

Dr. Toni Cortés – UPC – BSC- Barcelona – España.

Dr. Fernando G. Tinetti – UNLP, CIC Prov. de Bs. As. – La Plata – Argentina

Co-Director:

Dr. Silvio Gonnet – INGAR (CONICET - UTN FRSF) – Santa Fe - Argentina.

RESUMEN

Si bien los Sistemas Operativos disponen de características de seguridad, protección, gestión de recursos, etc. éstas parecen ser insuficientes para satisfacer los requerimientos de los sistemas informáticos que suelen estar permanente y globalmente conectados. Las actuales tecnologías de virtualización han sido y continúan siendo masivamente adoptadas para cubrir esas necesidades de sistemas y aplicaciones por sus características de particionado de recursos, aislamiento, capacidad de consolidación, seguridad, soporte de aplicaciones heredadas, facilidades de administración, etc. Una de sus restricciones es que el poder de cómputo de una Máquina Virtual (o un Contenedor) está acotado al poder de cómputo de la máquina física que la contiene. Esta tesis propone superar esta restricción abordando la problemática con el enfoque de un sistema distribuido.

Para poder alcanzar mayores niveles de rendimiento y escalabilidad, los programadores de aplicaciones nativas para la Nube deben partirlas en diferentes componentes distribuyendo su ejecución en varias Máquinas Virtuales (o Contenedores). Dichos componentes se comunican mediante interfaces bien definidas tales como las interfaces de Web Services. Las Máquinas Virtuales (o Contenedores) deben configurarse, asegurarse y desplegarse para poder ejecutar la aplicación. Esto se debe, en parte, a que los diferentes componentes no comparten la misma instancia de Sistema Operativo por lo que no comparten los mismos recursos abstractos tales como colas de mensajes, mutexes, archivos, pipes, etc. El defecto de esta modalidad de desarrollo de aplicaciones es que impide una visión integral y generalizada de los recursos. En ella, el programador debe planificar la asignación de recursos a cada componente de su aplicación y, por lo tanto, no solo debe programar su aplicación sino también gestionar la distribución de esos recursos.

En este trabajo se propone un modelo de arquitectura para un Sistema de Virtualización Distribuido (DVS) que permite expandir los límites de un dominio de ejecución más allá de una máquina física, explotando el poder de cómputo de un cluster de computadores. En un DVS se combinan e integran tecnologías de Virtualización, de Sistemas Operativos y de Sistemas Distribuidos, donde cada una de ellas le aporta sus mejores características. Esta arquitectura, por ejemplo, le brinda al programador una visión integrada de los recursos distribuidos que dispone para su aplicación relevándolo de la responsabilidad de gestionarlos. El modelo de DVS propuesto dispone de aquellas características que son requeridas por los proveedores de servicios de infraestructura en la Nube, tales como: mayor rendimiento, disponibilidad, escalabilidad, elasticidad, capacidad de replicación y migración de procesos, balanceo de carga, entre otras.

Las aplicaciones heredadas pueden migrarse más fácilmente, dado que es posible disponer de la misma instancia de un Sistema Operativo Virtual en cada nodo del cluster de virtualización. Las aplicaciones desarrolladas bajo las nuevas metodologías para el diseño y desarrollo de software para la Nube también se benefician adaptándose su utilización a un sistema que es inherentemente distribuido.

Palabras claves: Virtualización, Contenedores, Sistemas Distribuidos, IaaS.

Esta tesis está dedicada a mis amores: Flor, Vicky, Inés y Marie, por apoyarme en mi dedicación a la carrera de Doctorado, a mis padres: Merce y Pety, por el amor, los valores y la educación que marcaron mi vida, y a mis tías: Mary y Chiche, quienes fueron mis segundas madres.

Reconocimientos

Quiero hacer expresa mi gratitud y agradecimiento por el apoyo recibido, la dedicación y paciencia de mis directores y co-director.

Durante todo el período en que realicé mis estudios e investigaciones recibí el apoyo de Telecom Argentina S.A. a través del otorgamiento de licencias que me permitieron trabajar en la presentación de artículos para congresos y publicaciones, como así también asistir a cursos de post-grado, muchas veces haciéndose cargo de los costos de las inscripciones y viáticos.

También debo agradecer a la Dra. Ana Tymoschuck y a Marcela Tulián de la Sec. de Ciencia y Tecnología de la UTBN-FRSF, al ex-director Dr. Aldo Vechietti, y a la actual directora del departamento de Sistemas de Información, Dra. Milagro Gutierrez. Todos actuaron como verdaderos facilitadores para que pudiese realizar diferentes trabajos de investigación, viajes y presentaciones en congresos relacionados al Doctorado.

Como parte de mi formación realicé una estancia en el Barcelona Supercomputing Center bajo la supervisión de Dr. Toni Cortés patrocinada por el proyecto PROMINF y en parte por la UTN-FRSF. Quiero agradecer a Dr. Horacio Leone, quién se desempeñaba como Director del Departamento de Sistemas de Información de UTN-FRSF en ese momento, por sus gestiones al respecto.

También quisiera agradecer a los docentes de la UTN-FRSF, ingenieros Mariela Alemandi, Diego Padula y Oscar Jara, como así también a varios alumnos por sus trabajos realizados sobre los componentes específicos del prototipo del sistema.

No quiero dejar pasar esta oportunidad para expresar mi gratitud a Alejandra Pizarro y el resto del departamento de post-grado de la Facultad de Informática de UNLP, por sus gestiones y excelente atención recibida durante todo el período que me demandaron las carreras de posgrado de Magister de Redes y Doctorado en Informática.

PREFACIO

El modelo de arquitectura para el sistema de virtualización propuesto es el resultado de 6 años dedicados al estudio, investigación, desarrollo y enseñanza de Sistemas Operativos, Sistemas Distribuidos y tecnologías de Virtualización.

Los resultados parciales de este trabajo han sido publicados en los siguientes artículos en revistas y congresos:

- Pablo Pessolani, Tony Cortes, Fernando G. Tinetti, Silvio Gonnet, Oscar Jara; “*A Fault-Tolerant Algorithm For Distributed Resource Allocation*”; IEEE Latin America Transactions (Volume: 15, Issue: 11, Nov. 2017); ISSN: 1548-0992.
- Pablo Pessolani, Tony Cortes, Fernando G. Tinetti, Silvio Gonnet; “*Sistema de Virtualización con Recursos Distribuidos*”, WICC 2012 - Workshop de Investigadores en Ciencias de la Computación; Posadas, Abril 2012.
- Pablo Pessolani, Tony Cortes, Fernando G. Tinetti, Silvio Gonnet; “*Un mecanismo de IPC de microkernel embebido en el kernel de Linux*”; WICC 2013 - Workshop de Investigadores en Ciencias de la Computación; Paraná, Abril 2013.
- Pablo Pessolani, Tony Cortes, Fernando G. Tinetti, Silvio Gonnet, Diego Padula, Mariela Alemandi; “*A User-space Virtualization-aware Filesystem*”; 3er Congreso Nacional de Ingeniería Informática/Sistemas de Información – CONAIISI 2015; 19 al 20 de Noviembre de 2015, UTN – FRBA.
- Pablo Pessolani, Tony Cortes, Fernando G. Tinetti, Silvio Gonnet, “*Sistema de Virtualización Distribuido*”; WICC 2017 - Workshop de Investigadores en Ciencias de la Computación; Buenos Aires 27 y 28 de Abril de 2017.
- Pablo Pessolani, Tony Cortes, Fernando G. Tinetti, Silvio Gonnet; “*An IPC Software Layer for Building a Distributed Virtualization System*”; CACIC 2017 - Congreso Argentino de Ciencias de la Computación, La Plata, Argentina, 2017.
- Pablo Pessolani, Tony Cortes, Fernando G. Tinetti, Silvio Gonnet; “*An Architecture Model for a Distributed Virtualization System*”; Cloud Computing 2018; The Ninth International Conference on Cloud Computing, GRIDs, and Virtualization; ISSN: 2308-4294, ISBN: 978-1-61208-607-1. Barcelona, España.2018.

Dada la magnitud e implicancias tecnológicas de la propuesta, el proceso de investigación y desarrollo no se detiene con este trabajo. Es mi intención seguir explorando las posibilidades que ofrece y continuar investigando acerca de la inclusión de nuevas funcionalidades como así también acerca de su integración con otros sistemas de infraestructura más maduros y establecidos como estándares de facto.

CONTENIDO

1. INTRODUCCION	9
1.1. Motivación	18
1.2. Contribución	21
1.3. Alcance de la Tesis	22
1.4. Organización de la Tesis	22
2. TECNOLOGIAS RELACIONADAS	24
2.1. Tecnologías de Virtualización	24
2.1.1. Virtualización de Hardware/Sistema	26
2.1.2. Paravirtualización.....	35
2.1.3. Virtualización basada en Sistema Operativo.....	36
2.1.4. Virtualización de Lenguaje/Proceso/Aplicación.....	38
2.1.5. Virtualización de Sistema Operativo.....	39
2.1.6. Emulación de Plataforma	40
2.1.7. Emulación de Sistema Operativo.....	41
2.1.8. Virtualización Reversa.	41
2.1.9. Capacidades de las actuales Tecnologías de Virtualización	42
2.2. Otras Tecnologías Relacionadas	44
2.2.1. Sistemas Operativos Distribuidos con Imagen de Sistema Unico	45
2.2.2. Unikernel	48
2.2.3. Sistemas Operativos Multiservidor o de Microkernel	50
2.2.4. Comunicaciones de Grupo	53
2.2.5. Migración de Procesos, Máquinas Virtuales y Contenedores	54
2.2.6. Redundancia y Replicación	59
3. MODELO DE ARQUITECTURA PROPUESTO	61
3.1. Topología de un cluster DVS	63
3.2. Abstracciones	64
3.2.1. Relaciones entre las Abstracciones	66
3.2.2. Semántica del IPC	67
3.3. Modelo de Arquitectura de DVS	74
3.4. Descripción de los Componentes del DVS	74

3.4.1. Núcleo de Virtualización Distribuida (DVK).....	75
3.4.2. Representante de Comunicaciones de Nodos Remotos (Proxies).....	76
3.4.3. Contenedores.....	77
3.4.4. Contenedores Distribuidos.....	78
3.4.5. Sistema Operativo Virtual.....	80
3.4.6. Relaciones entre Procesos y Endpoints.....	85
3.5. Otros Componentes Requeridos.....	86
3.5.1. Replicación Primario-Respaldo.....	87
3.5.2. Replicación de Máquina de Estados.....	88
3.5.3. Migración de Procesos.....	88
3.5.4. Seguridad y Protección.....	89
3.5.5. Planificación Distribuida.....	89
4. PROTOTIPO DE DVS.....	91
4.1. Núcleo de Virtualización Distribuida (DVK).....	93
4.1.1. Módulo para el Kernel de Linux.....	94
4.1.2. APIs de Gestión del DVS.....	99
4.1.3. Conformación del cluster del DVS (nodos unidos por <i>proxies</i>).....	100
4.1.4. APIs de Gestión de Contenedores Distribuidos.....	106
4.1.5. APIs de Gestión de Procesos.....	107
4.1.6. APIs de Gestión de Privilegios.....	109
4.1.7. M3-IPC.....	110
4.1.8. APIs para Migración y de Replicación de Procesos.....	122
4.1.9. Directorio <i>/proc/dvs</i>	123
4.1.10. Sistema de Comunicaciones Grupales (GCS).....	126
4.2. Módulo <i>Webmin</i> para Gestión del DVS.....	127
4.3. Contenedor Distribuido (DC).....	132
4.4. Sistemas Operativos Virtuales (VOS).....	133
4.4.1. VOS Multiservidor.....	133
4.4.2. VOS Unikernel (ukVOS).....	141
4.4.3. Algoritmo de Asignación de Ranuras (<i>SLOTS</i>).....	142
4.4.4. Gestor de Discos Virtuales Replicados.....	146
4.4.5. Sincronización Dinámica de Réplicas.....	150
4.5. Misceláneos.....	151
4.5.1. Transferencia de Archivos entre MoL (Guest) y Linux (Host).....	152
4.5.2. Servidor FTP y Cliente FTP utilizando M3-IPC.....	152

4.5.3. Servidor Web y Cliente Web utilizando M3-IPC.....	152
4.5.4. Proxy HTTP – M3-IPC	153
5. EVALUACION DEL PROTOTIPO	155
5.1. M3-IPC.....	155
5.2. Algoritmo de Asignación de Ranuras.....	163
5.3. MOL-VDD	168
5.4. MOL-FS.....	170
5.5. MOL-NWEB	173
5.6. Resumen de la Evaluación del Prototipo	175
6. TRABAJOS FUTUROS	178
7. CONCLUSIONES	180
<i>Glosario</i>	<i>183</i>
<i>Referencias.....</i>	<i>187</i>

1. INTRODUCCION

Las actuales tecnologías de virtualización han sido y son masivamente adoptadas para satisfacer aquellos requerimientos en que los Sistemas Operativos (OS: Operating System) han presentado debilidades, tales como el aislamiento de fallos, de seguridad y de rendimiento. Además, la virtualización incorpora nuevas facilidades tales como el particionado de recursos, la posibilidad de realizar la consolidación de servidores, el soporte para la ejecución de aplicaciones legadas, la disponibilidad de herramientas que facilitan su gestión, etc. Todas estas características resultan muy atractivas para las organizaciones en general y para los proveedores de Servicios en la Nube (Cloud Services) en particular.

Actualmente, existen varias tecnologías de virtualización que permiten brindar Infraestructura como Servicio (IaaS: Infrastructure as a Service) desplegadas sobre un cluster de servidores enlazados por redes de alta velocidad. Las redes de almacenamiento (SAN: Storage Area Networks), los dispositivos de seguridad (firewalls de red, firewalls de aplicación, Sistemas de Detección y Prevención de Intrusos, etc.), los dispositivos de red (switches, routers, balanceadores de carga, etc.), y un conjunto de sistemas de gestión y monitoreo complementan la infraestructura (informática) requerida para brindar este tipo de servicios a terceros en Nubes Públicas, o en la propia compañía (on-premise) para el caso de Nubes Privadas.

Para establecer los límites de las responsabilidades entre el proveedor de Servicios en la Nube y el cliente, parece adecuado adoptar la definición del NIST respecto a Infraestructura como Servicio (IaaS):

La capacidad de proveer al consumidor procesamiento, almacenamiento, redes y otros recursos informáticos fundamentales sobre los cuales el consumidor puede implementar y ejecutar software arbitrario, pudiendo incluir sistemas operativos y aplicaciones. El consumidor no

*administra ni controla la infraestructura subyacente de la Nube, pero **tiene control** sobre los sistemas operativos, el almacenamiento y otras aplicaciones implementadas; y posiblemente un control limitado de los componentes de red seleccionados (por ejemplo, firewalls) [1].*

Las tecnologías de virtualización más utilizadas son *Virtualización de Hardware*, *Paravirtualización* y *Virtualización basada en Sistema Operativo*. Cada una de ellas presenta diferentes características en cuanto a niveles de consolidación, rendimiento, escalabilidad, disponibilidad y aislamiento. Una *Máquina Virtual* (VM: Virtual Machine) es un entorno de ejecución aislado creado por el software encargado de gestionar todos los recursos del computador al que se lo conoce como *Hipervisor* o *Monitor de Máquina Virtual* (VMM: Virtual Machine Monitor). Generalmente, el término VM se relaciona a las tecnologías de Virtualización de Hardware y de Paravirtualización, aunque también es utilizado en la virtualización de procesos tales como la *Máquina Virtual Java* (JVM: Java Virtual Machine). Los entornos de ejecución aislados creados a nivel de OS se denominan *Contenedores*, *Prisiones* o *Zonas* (según el OS). A los efectos de utilizar una única denominación en el resto de la tesis, a estos entornos o dominios de ejecución se los referirá como *Contenedores*.

Sin considerar cuál es la denominación del *entorno de ejecución* creado por la virtualización según su tipo, queda claro que éste resulta del particionado de los recursos de un computador. Como consecuencia de esto, *el poder de cómputo y la utilización de recursos de esos entornos estarán acotados por el computador que lo contiene*. Al menos dos preguntas surgen al considerar esta limitación:

1. *¿Cómo se pueden expandir el poder de cómputo y la utilización de recursos de una VM o Contenedor a varios computadores?*

Se pretende disponer de un entorno de ejecución aislado equivalente a una VM o Contenedor, pero que su alcance no se limite a un único computador.

2. *¿Cómo lograr mayores niveles de rendimiento y escalabilidad de las aplicaciones que se ejecutan en la Nube?*

Las aplicaciones que ejecutan en la Nube demandan cada vez más potencia de cómputo y recursos (memoria, almacenamiento, red, etc.). El rendimiento y la escalabilidad requerida son imposibles de lograr con un único computador. Además, dada la criticidad de las aplicaciones, existen otros requerimientos que deben cumplirse de tal modo que se

les permita a éstas expandirse o contraerse según la demanda (elasticidad), tolerar fallos, ahorrar energía y facilitar su despliegue y gestión.

Para la primera pregunta existen al menos dos soluciones como posibles respuestas:

- a) Utilizar un *hipervisor distribuido*, como *ScaleMP* [2], el cual crea un sistema de MultiProcesamiento Simétrico Virtual (vSMP) a partir de un cluster de servidores. A la tecnología se la conoce como *Virtualización para Agregación* o como *Virtualización Reversa*, dado que múltiples computadores físicos crean una única VM. Sobre esta VM se podría instalar un OS tal como Linux y generar diferentes Contenedores (Fig. 1). La asignación de recursos a los Contenedores la lleva a cabo el OS, el cual desconoce o ignora que está ejecutando sobre un hardware virtualizado. De esta forma el OS no puede asignar en forma específica procesos o Contenedores a determinados nodos del cluster. ScaleMP es una tecnología de Virtualización de Hardware, por lo que tiene sus ventajas e inconvenientes, los que serán analizados en la [Sección 2.1.1](#), además todas las aplicaciones deben utilizar el mismo OS.

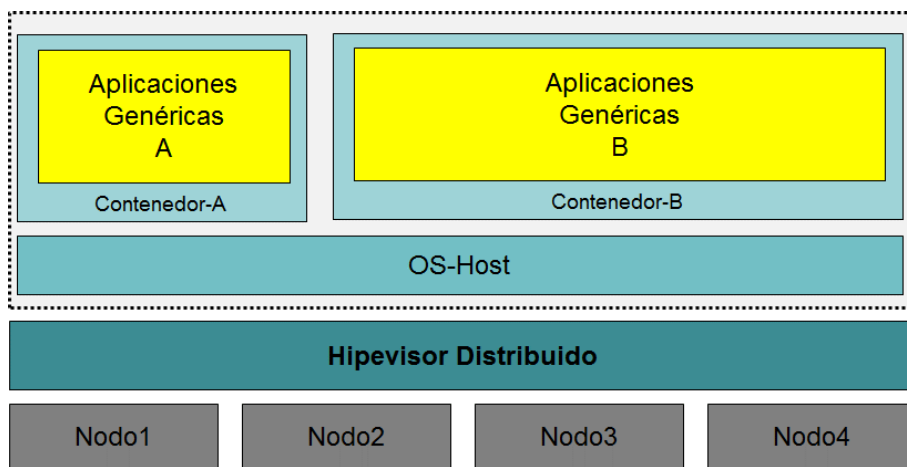


Figura 1. Hipervisor Distribuido.

- b) Utilizar un *Sistema Operativo Distribuido de Imagen Unica* (SSI: Single System Image – DOS: Distributed Operating System). Existen aquí al menos 2 posibilidades [3]: una opción (Fig. 2) es crear Contenedores sobre este DOS, el cual distribuye los procesos en diferentes nodos según la carga o las directivas del administrador del sistema. Como se utilizaría una tecnología de virtualización basada en OS, todos los Contenedores ejecutan por sobre el mismo OS anfitrión (*OS-host*), disponiendo cada uno de ellos su propio entorno o dominio aislado. En este caso también, todas las aplicaciones deben utilizar el mismo OS.

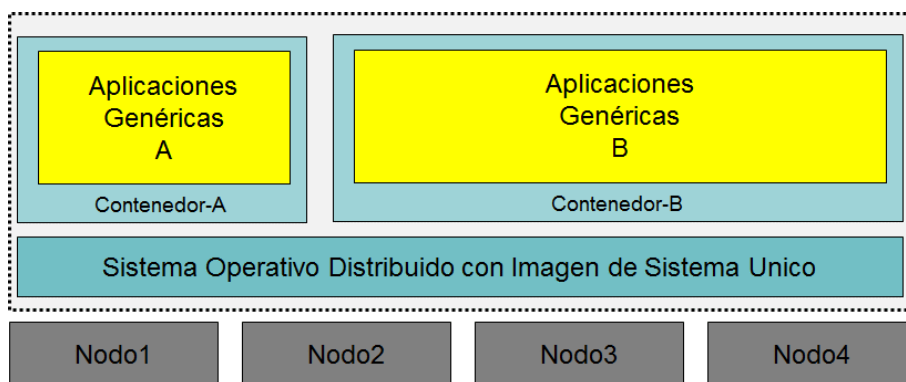


Figura 2. Sistema Operativo Distribuido con Imagen de Sistema Único (opción 1).

La otra opción es ejecutar un DOS sobre VMs (Fig. 3). Esta opción tiene varios problemas. No hay forma de asegurar que cada VM (donde reside un nodo que compone el DOS) se localice en nodos diferentes del cluster de virtualización, por lo que no se podrán asumir ciertas condiciones para tolerar fallos o distribuir carga. Por ejemplo en la Fig. 3, las **VM2** y **VM3** comparten el mismo **Nodo2**. Aunque se pudiese controlar la localización de cada VM, los mecanismos que dispone un DOS para monitorear el desempeño de los nodos que lo componen y evaluar el factor de utilización de los recursos de los mismos serían *engañados* por el hipervisor subyacente. Este engaño es consecuencia directa de la tecnología de virtualización de hardware la cual emula y particiona dispositivos y otros recursos computacionales (CPU, memoria, DMA, etc).

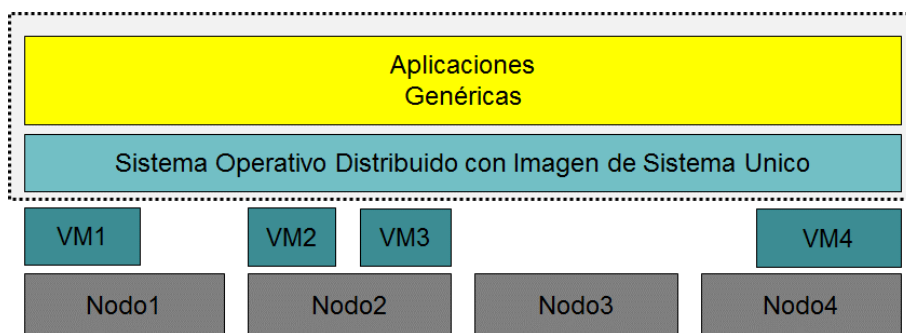


Figura 3. Sistema Operativo Distribuido con Imagen de Sistema Único (opción 2).

El DOS solo accede a los parámetros y estado de cada una de las VMs sobre las que se apoya, no accede a los parámetros y estado de los componentes del cluster. Esta imposibilidad de acceder a métricas reales provocaría que el DOS tome decisiones posiblemente incorrectas en cuanto a planificación de procesos, balanceo de cargas, migración de procesos, replicación de datos, etc. Además, hay que considerar que el hipervisor tiene su propio planificador, gestor de memoria, gestor de temporizadores,

etc. y el SSI también dispone de componentes equivalentes, duplicándose así sus funciones. Como los SSIs están generalmente diseñados para apoyarse en un cluster de máquinas reales, la distorsión de las métricas y de la localización es a consecuencia de apoyarse sobre VMs.

Actualmente, la segunda pregunta (*¿Cómo lograr mayores niveles de rendimiento y escalabilidad de las aplicaciones que se ejecutan en la Nube?*), se responde mediante el despliegue de diferentes componentes de las aplicaciones sobre varias VMs o Contenedores (Fig. 4).

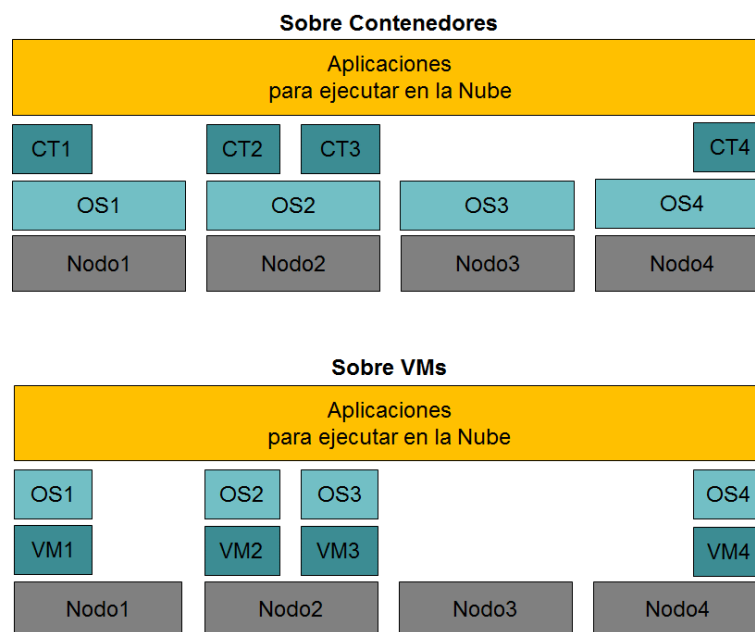


Figura 4. Aplicaciones en la Nube sobre Contenedores (CTi) y sobre VMs.

Los diseñadores de aplicaciones para la Nube utilizan metodologías de desarrollo de software basada en la Arquitectura de Micro-Servicios (MSA: Micro Service Architecture) [4] o Arquitectura Orientada al Servicio (SOA: Service Oriented Architecture) [5]. Ambas arquitecturas proponen dividir las aplicaciones en múltiples servicios desplegados en múltiples VMs o Contenedores. Para realizar la configuración y el despliegue de los diferentes componentes se utilizan gestores de Contenedores tales como Docker [6], Mesos [7] o Kubernetes [8]. En el caso de Mesos, éste se anuncia como un DC/OS (Data Center Operating System), pero esto resulta confuso ya que en realidad no es un OS, al menos en lo que a la definición tradicional de un OS refiere. Mesos es el software que permite gestionar todos los recursos de un cluster, tratando de integrar la aplicación a esos recursos.

Por otro lado, cuando dos procesos componentes de la aplicación deben comunicarse, si los mismos se encuentran ejecutando en la misma VM o Contenedor (bajo el control del mismo OS) se

pueden comunicar utilizando pipes o tuberías, memoria compartida, colas de mensajes, señales, semáforos, etc. Pero, cuando se comunican con procesos que se encuentran en otras VMs o Contenedores deben utilizar protocolos de red, sean éstos de alto nivel como HTTP, de nivel medio como RPC, o de bajo nivel como sockets TCP o UDP.

Si bien existen numerosas herramientas y una multiplicidad de software para desarrollar aplicaciones para la Nube, no conforman una solución transparente, consistente y homogénea como en un sistema centralizado. La complejidad recae entonces en el usuario/programador/administrador, el cual debe hacerse cargo de aspectos relativos al sistema y a los recursos distribuidos [9]. Esto se debe, a que todo el conjunto de recursos dispersos (de cómputo, de almacenamiento y de red) presentan una visión no homogénea que el usuario debe integrar, gestionar y mantener. Esto significa, que desde el punto de vista netamente práctico, se requiere realizar tareas tales como: asignar direcciones IP, puertos TCP/UDP, configurar reglas en los firewalls, establecer servidores de autenticación, mantener versiones de OS, mantener actualizadas bibliotecas y paquetes de software, y mantener archivos de configuración sincronizados entre VMs/Contenedores.

Cuando se utiliza IaaS para desplegar las aplicaciones de esta manera, no se puede asegurar que los Contenedores o las VMs contratadas se localicen en nodos diferentes (como en el caso del Nodo2 de la Fig. 4) con lo cual, muchos supuestos realizados para distribuir cargas o para tolerar fallos a través de la redundancia dejan de tener sustento.

Algo análogo respecto a la responsabilidad adicional que recae sobre el usuario o el programador de aplicaciones en la nube ocurre en los siguientes ejemplos:

1. *Un OS de los años 60 vs. OS actual:* En el primero, el usuario/administrador debía asignar la cantidad de memoria y el tiempo de cómputo a cada programa a ejecutar. También, debía definir cuáles áreas de memoria de un proceso debían volcarse a disco o recuperarse de éste durante su ejecución (overlay), establecer cuantos bloques de disco debían asignárseles a un archivo y en qué conjunto de pistas se ubicarían, por mencionar algunas operaciones básicas. Actualmente, todas esas operaciones las realiza el propio OS sin intervención del usuario/administrador, aunque algunas de ellas son parametrizables o configurables a nivel sistema o proceso.
2. *Sistema de Procesamiento Paralelo vs. Sistema Operativo Distribuido de Imagen Unica en un cluster:* En el primero, el programador tiene plena conciencia de que sus programas se ejecutarán en un cluster y por lo tanto utiliza los lenguajes, las bibliotecas, las herramientas y técnicas adecuadas para obtener el máximo rendimiento, sacrificando transparencia. En tanto, en un SSI, se ocultan tanto al programador como a los usuarios, la localización de procesos, de los datos y

objetos, la migración de procesos, la replicación de procesos y de datos, facilitando así su programación, despliegue y mantenimiento, pero a costa de un rendimiento sub-óptimo.

Estas analogías sugieren que se requiere de disponer de un sistema de virtualización con una visión integrada de los recursos, con características de SSI de tal forma de ocultar los detalles del cluster y de los entornos de virtualización (VMs o Contenedores) a los programadores y a las aplicaciones.

Una tecnología que ofrece transparencia tal como un DOS de tipo SSI es la VM Java Distribuida de JESSICA2 [10]. Es un middleware que soporta la ejecución en un cluster, pero que está limitada a la ejecución de aplicaciones en lenguaje Java. De forma similar trabaja la VM de Inferno [11] en donde las funciones de virtualización no son implementadas en un código monolítico sino en múltiples servicios son factorizados.

Una alternativa a contratar infraestructura en la Nube (IaaS) es contratar servicios de tipo *Sin Servidor* (Serverless), tal como AWS Lambda [12]. En esta modalidad, se transfieren las aplicaciones a la Nube, pero no se contrata servidor alguno. Las aplicaciones utilizan Servicios Complementarios o de *Backend* (BaaS: Backend as a Service) tales como gestores de bases de datos (DBMS: DataBase Management Server), servicios de autenticación, servicios de directorio, etc., en los cuales sólo se paga por el uso de los recursos que hacen la aplicación y sus servicios. Es decir, a mayor utilización mayor será el costo del servicio. El proveedor de estos servicios acuerda con el cliente un determinado nivel de servicio (SLA: Service Level Agreement), que consiste en especificar, por ejemplo, el tiempo de respuesta de la aplicación y la disponibilidad de la misma. Si la aplicación tiene mucha demanda, el proveedor deberá desplegar varias instancias de ejecución de la aplicación para así lograr los parámetros de rendimiento comprometidos al cliente.

Muchos proveedores de Servicios en la Nube ofrecen una multiplicidad de herramientas y servicios para el desarrollo de las aplicaciones en la forma de Plataforma como Servicio (PaaS: Platform as a Service) [1], pero éstos difieren entre sí. Esto provoca una dependencia fuerte respecto al proveedor del servicio lo que dificulta una futura migración hacia otra plataforma.

Los proveedores de IaaS generalmente utilizan SANs en sus Datacenters para virtualizar el almacenamiento y proveer de discos rígidos virtuales a las VMs de los servidores de virtualización. De esta forma, los recursos de la VM (en este caso el disco rígido) se expanden por fuera de los límites de su computador anfitrión (host). Esto puede ser interpretado como una excepción a la limitante en cuanto a la capacidad de las VM (y Contenedores) previamente mencionada. Si ésta

modalidad de procesamiento se extiende a varios tipos de servicios y recursos, devendría un nuevo modelo de procesamiento distribuido en las tecnologías de virtualización.

Esta tesis propone un modelo de arquitectura que se basa en este enfoque distribuyendo procesos, servicios y recursos para proveer un entorno de ejecución aislado basado en múltiples *Contenedores Distribuidos* y en la factorización de los OSs. El resultado es un Sistema de Virtualización Distribuido que combina tecnologías de OS y de virtualización de tal forma de que las aplicaciones puedan acceder a los recursos de su OS sin importar en el host donde se éstas se ejecutan. Esta característica simplifica el desarrollo de aplicaciones (costo y tiempo). Un par de casos pueden ejemplificarlo:

Supóngase un DBMS ejecutando en un host o en una VM (Fig. 5) y que el almacenamiento lo provee una SAN. Periódicamente se necesita realizar un respaldo (backup) en línea de la Base de Datos, pero asegurando la consistencia de la misma en su restauración.

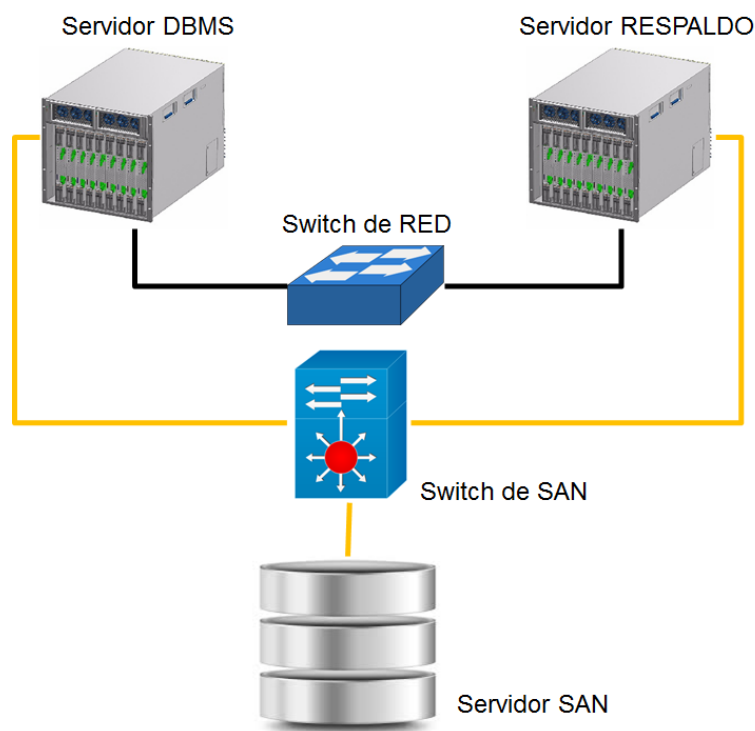


Figura 5. Respaldo en línea de una Base de Datos residente en una SAN.

El proceso de respaldo podría ejecutarse en un servidor o en una VM diferente al DBMS por cuestiones de rendimiento, pero conectado a la misma red del DBMS, y ambos accediendo a la misma SAN. Cómo el DBMS y el proceso de respaldo se ejecutan en diferentes (instancia de un) OS, deben comunicarse entre sí estableciendo un protocolo ad-hoc por la red de tal forma de sincronizar el

acceso a la DB. Esto requiere la configuración de direcciones IP, puertos, nombres, reglas de firewalls, autenticación de usuarios, que describen la topología, facilidades y restricciones del sistema.

En cambio, si el DBMS y el proceso de respaldo se ejecutan sobre el mismo OS, podrían utilizar recursos de sincronización que éste ofrece tales como mutexes o semáforos. De esta forma se simplifica la programación, el despliegue y el mantenimiento. Pero, para ello, la misma instancia del OS debe estar disponible en el servidor donde ejecuta el DBMS y en el servidor donde ejecuta el proceso de respaldo. Este funcionamiento es equivalente al resultante de implementar la solución utilizando un DOS de tipo SSI.

Otra aplicación podría ser un Sistema de Prevención contra Intrusos de Red (NIPS: Network Intrusion Prevention System) [13] (Fig. 6). Generalmente estos sistemas están constituidos por una serie de procesos Sensores (sniffers) distribuidos en diferentes redes, los cuales procesan en forma parcial los paquetes y conexiones que atraviesan las redes.

Todos los dispositivos Sensores tributan su información a un Gestor del IDS (IDS: Intrusion Detection System), pero el análisis lo pueden realizar tanto los Sensores, el Gestor o todos ellos. Entre todos los Sensores y el Gestor deben acordar cuáles paquetes y conexiones deben ser capturados y analizados por cada uno de los Sensores. Para que dos o más Sensores no se hagan cargo de los mismos paquetes o conexiones o sesiones, deben sincronizarse entre sí. El mecanismo para la comunicación y para la sincronización suele ser ad-hoc entre el Gestor y los Sensores, haciendo más dificultosa su programación y configuración.

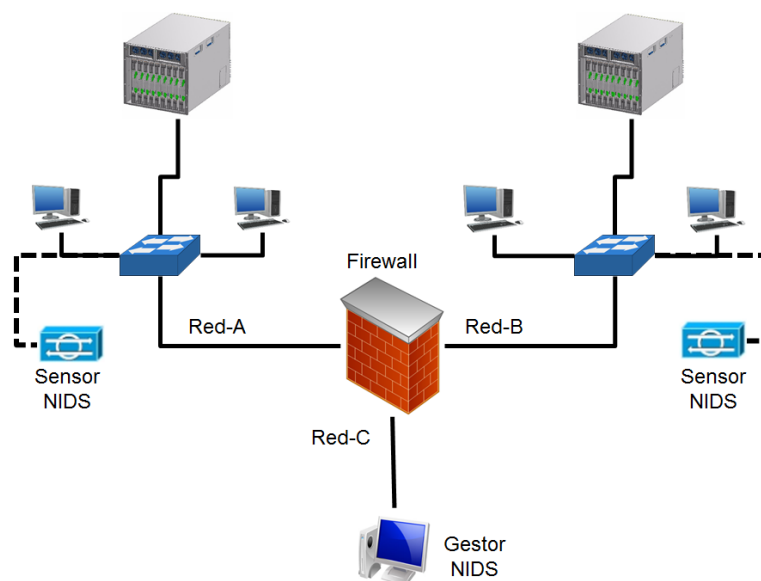


Figura 6. IDS de red.

Al utilizar un sistema tipo SSI, todos los procesos (Gestores y Sensores) ejecutan bajo la supervisión de un mismo OS, sin considerar el hecho de que cada uno de ellos lo hace en computadores diferentes. Por esta razón, los procesos involucrados pueden utilizar los mecanismos de Comunicación entre Procesos (IPC: Inter-Process Communications) estándares (por ejemplo, POSIX) que ofrece el OS, tales como colas, señales, tuberías, semáforos o mutexes.

1.1. Motivación

Conforme una VM o Contenedor resulta de una partición de recursos de un computador, su potencia de cómputos y disponibilidad de recursos están limitadas al computador que lo contiene (las partes no pueden ser mayores que el todo). Esto nos permite realizar la siguiente afirmación:

En los sistemas de virtualización de la actualidad, los recursos y el poder de cómputo de una VM o de un Contenedor se encuentran limitados al computador donde se ejecutan.

Como vimos, esta afirmación tiene algunas excepciones. Una de ellas es la tecnología mencionada de *virtualización inversa o reversa* en donde múltiples computadores físicos conforman un única VM [2, 14, 15, 16]. Esta tecnología permite integrar el hardware de múltiples nodos de un cluster y que éstos se presenten ante el OS como un único hardware constituido por la agregación de todos esos recursos. Como se mencionó previamente, sobre el OS que ejecuta sobre ella, se podría utilizar otro sistema de virtualización (tal como Contenedores) cuyos componentes podrían estar distribuidos ejecutando en diferentes nodos, pero sin percibirlo. Dicha configuración carece de la integración necesaria de ambas tecnologías como para facilitar la administración unificada que permita el mapeo de recursos computacionales a Contenedores.

En los Datacenters actuales es común encontrar, como parte de la infraestructura, dispositivos de almacenamiento tipo SAN o NAS (Network Attached Storage) que se acceden a través de una red dedicada y específica para tal fin. Los hipervisores que se utilizan en las infraestructuras de esos Datacenters disponen de Gestores de Dispositivos (Device Drivers) de discos que permiten virtualizar el almacenamiento remoto (Storage Virtualization) contenido en una SAN tal como si fuese almacenamiento interno del propio servidor de virtualización. Esto, de alguna manera distribuye el procesamiento y el almacenamiento entre el servidor de virtualización y la SAN, por lo que una VM que mapea un disco en la SAN se extiende más allá de los límites del computador que la contiene. En esta tesis se propone una extensión de este concepto o forma de procesamiento, constituyendo en sí misma un nuevo modelo de arquitectura de virtualización.

Tesis: Se propone un modelo de Arquitectura de un Sistema de Virtualización Distribuido (DVS: Distributed Virtualization System), el cual permite construir dominios de ejecución denominados Contenedores Distribuidos (DC: Distributed Containers) que pueden extenderse más allá de los límites de una máquina física.

El objetivo es que las aplicaciones desplegadas en DCs dispongan de mayores cantidades de recursos y mayor poder de cómputo para poder brindar niveles de rendimiento, disponibilidad, escalabilidad y elasticidad más elevados. Los recursos (tanto físicos como abstractos) y procesos que constituyen un DC, pueden encontrarse dispersos (y eventualmente replicados) en diferentes nodos de un cluster de virtualización.

Un DVS permite ocultar tanto a usuarios como a las aplicaciones cuáles son los nodos que componen un DC, cuántos de esos componentes se encuentran replicados, e incluso cuáles componentes se activan/desactivan cuando se producen fallos. Los usuarios y las aplicaciones de cada uno de los OS que ejecutan en el entorno de un DC perciben un SSI.

El modelo propuesto conserva las características tan apreciadas de las tecnologías de virtualización actuales tales como confinamiento, consolidación y seguridad, y los beneficios de los DOSs tipo SSI, tales como la transparencia, mejor rendimiento, mayor elasticidad, disponibilidad y escalabilidad.

Un DVS correctamente implementado permite cumplir con los requerimientos que tienen los proveedores de Servicios en la Nube (de tipo IaaS) tales como alta disponibilidad y rendimiento, capacidad de replicación recursos y migración de procesos, mayor elasticidad, balanceo de carga y facilidad en su gestión. Además, un DVS admite la ejecución de varias instancias de diferentes Sistemas Operativos Virtuales (VOS: Virtual Operating Systems) [17]. Cada VOS se ejecuta confinado dentro de su propio DC, el cual puede abarcar un subconjunto de nodos del cluster (agregación de recursos) y, por otro lado, múltiples DCs pueden compartir nodos entre sí (partición de recursos).

Tanto las aplicaciones legadas como las aplicaciones nativas para la Nube se benefician con sus características. En general, la migración de aplicaciones legadas que se ejecutan en servidores propios (on-premise) hacia aplicaciones desarrolladas para ejecutar en la Nube requiere de una importante inversión de dinero y tiempo. Si en la Nube estuviese disponible una interface estándar, tal como POSIX, pero sobre una infraestructura con capacidades extendidas a varios nodos de un cluster, las tareas de migración se simplificarían reduciendo los costos y tiempos.

Las aplicaciones desarrolladas bajo el modelo de MSA [4] o SOA [5] también se verían beneficiadas con un DVS. Un DVS es inherentemente distribuido, por lo que dispone de herramientas de comunicación que facilitan el intercambio de datos y la sincronización entre servicios o micro-servicios. Un DVS presenta una vista única integral del conjunto de componentes de la aplicación que facilitan su despliegue, mantenimiento y operación.

La elasticidad de un DVS también lo hace adecuado para proveer Servicios Complementarios Sin Servidor (Serverless) dado que se pueden activar y desactivar servicios a medida que se lo requiere. Los servicios a ofrecer pueden integrarse como componentes del DVS y la gestión de rendimiento del DVS le permite expandir o contraer la cantidad de nodos y recursos asignados a un cliente.

La adopción y utilización de un DVS se basa en muchos de los argumentos del uso de los SSI-DOS. Varios procesos inter-relacionados pueden ejecutarse en diferentes nodos, pero utilizando accediendo o utilizando los mismos recursos abstractos que ofrece un VOS en forma transparente a la localización de los mismos. Esta característica, *simplifica la programación de aplicaciones y de bibliotecas de software* (dado que el sistema se comporta como un sistema centralizado), y por lo tanto puede realizar operaciones sobre recursos tales como perfiles de usuarios, semáforos, mutexes, tuberías, archivos y señales. Adicionalmente, un DVS simplifica el despliegue, la gestión y operación de aplicaciones complejas porque no se necesitan configurar aspectos relacionados a la infraestructura y la plataforma *por cada aplicación* tales como direcciones IP, puertos TCP/UDP, reglas de firewalls, privilegios de usuarios o URLs. Las configuraciones se hacen a nivel de cluster del DVS, lo que incluye a todas las aplicaciones que se ejecutan en él. Los administradores disponen de una visión global e integrada de procesos y recursos lo que reduce los costos de operación.

Desde los inicios del proyecto, se comenzó con el desarrollo de un prototipo de DVS que permitiera validar diferentes aspectos considerados en el modelo, detectar errores de diseño y proponer mejoras. Se describen aquí también los detalles de implementación de los diferentes componentes del prototipo y se presentan resultados de las mediciones previas a las optimizaciones. El prototipo demuestra la factibilidad de implementar un DVS, mediante el desarrollo módulos de kernel, bibliotecas, programas de pruebas, etc. para Linux. También se han convertido dos OSs a VOSs, y se han utilizado herramientas y facilidades ya existentes en Linux.

1.2. Contribución

La contribución esperada de esta tesis es un modelo de arquitectura de un DVS que permite conformar entornos de ejecución aislados (DCs) que aprovechan de la agregación de recursos computacionales de los nodos de un cluster. Como se pueden adoptar múltiples estrategias, enfoques, mecanismos y variantes para el modelo del DVS, esta tesis se limita en su alcance a desarrollar aquellos componentes considerados centrales en importancia ([Sección 1.3](#)). Otras características y cualidades potenciales del modelo se proponen para futuros trabajos de investigación.

Además del modelo de arquitectura de DVS, en esta tesis se realizan otras contribuciones a saber:

- ✓Un mecanismo de IPC que es independiente de la localización de los procesos (se encuentren los mismos en el mismo o en diferentes nodos), que soporta en forma transparente la migración de procesos, la distribución de cargas y la utilización de mecanismos de replicación de tipo Primario-Respaldo o tipo Máquina de Estado.
- ✓Un algoritmo distribuido tolerante a fallos de asignación de recursos (utilizado en el VOS distribuido del prototipo), que permite asignar recursos y mantener la consistencia y gran parte de la disponibilidad tanto en fallos de tipo de Detención Total o de Partición de Red.
- ✓Un VOS distribuido compuesto de varios procesos servidores dispersos en varios nodos del cluster, algunos de ellos replicados.
- ✓Un VOS tipo unikernel que contiene un servidor web, un sistema de archivos y una pila de protocolos TCP/IP del cual se pueden separar varios componentes hacia otros nodos, tales como el servidor de archivos o el servidor de almacenamiento en disco.

Más allá de prototipo construido, para el lector debe quedar claro que el modelo propuesto es de carácter teórico y genérico, el cual puede extenderse, modificarse o simplemente ser inspirador para otros tipos de enfoques.

1.3. Alcance de la Tesis

El alcance de esta tesis abarca la descripción del modelo de arquitectura de un DVS que se propone como una nueva tecnología de virtualización, la descripción de sus componentes, el análisis de los beneficios que se obtienen con este sistema, la descripción del prototipo de DVS y una evaluación de rendimiento del mismo.

Aspectos relacionados a su integración con arquitecturas de gestión de servicios en la Nube quedan más allá del alcance. Estos y otros muchos tópicos relacionados quedan abiertos para futuros trabajos de investigación.

1.4. Organización de la Tesis

El resto de esta tesis está organizado en los siguientes capítulos:

- [Capítulo 2](#): En este capítulo se analizan diferentes tecnologías de virtualización, algunas de las cuales componen el modelo de arquitectura del DVS. Como se pretende que el DVS propuesto pueda ser adoptado por corporaciones, institutos de investigación, universidades, organismos gubernamentales y proveedores de Servicios en la Nube, se describen aquí esas otras tecnologías complementarias que dan soporte a las características exigidas por estos tipos de usuarios.
- [Capítulo 3](#): Se describen en este capítulo los detalles del modelo de arquitectura del DVS y sus componentes. Esto le permite, a aquellos interesados en implementarla, que puedan adoptar o adaptar sus propias tecnologías al modelo sin necesariamente seguir las estrategias aplicadas para prototipo de DVS desarrollado.
- [Capítulo 4](#): En este capítulo se presenta el prototipo de DVS detallando sus componentes, incluidos dos tipos de VOS desarrollados para comprobar su factibilidad.
- [Capítulo 5](#): Gran parte de los componentes del prototipo de DVS fueron sometidos a pruebas de rendimiento (complementarias a las pruebas de funcionamiento). En este capítulo se presentan y analizan los resultados de esas pruebas, muchas de las cuales ya fueron sometidas a discusión de la comunidad científica y tecnológica.

-
- [Capítulo 6](#): El tema de tesis es muy amplio, y admite numerosos enfoques y estrategias de implementación. Tanto el modelo como el prototipo admiten modificaciones, ampliaciones y, posiblemente para hacer más atractiva su adopción, se deban focalizar los esfuerzos en una mayor integración con sistemas de gestión de Servicios en la Nube. El [Capítulo 6](#) presenta algunos de los trabajos futuros previstos como continuidad en la investigación y desarrollo de la tecnología de DVS.
 - [Capítulo 7](#): Esta tesis culmina con las conclusiones respecto a los aportes que la misma hace a la comunidad acerca de una nueva forma de virtualización.

2. TECNOLOGÍAS RELACIONADAS

El modelo de arquitectura de un DVS surge a partir de integrar múltiples tecnologías de virtualización, de OSs y de sistemas distribuidos en general. Todas ellas fueron fuentes de inspiración para construir el modelo propuesto. En este capítulo se realiza una revisión de éstas tecnologías, y en aquellas que correspondan, se destacan las ideas y conceptos básicos fuertemente relacionados a la propuesta de esta tesis.

2.1. Tecnologías de Virtualización

Si bien los OSs disponen de características de seguridad, protección, gestión de recursos, etc. éstas parecen ser insuficientes para satisfacer los requerimientos de los sistemas informáticos que suelen estar permanente y globalmente conectados. Las actuales tecnologías de virtualización han sido y continúan siendo masivamente adoptadas para cubrir esas necesidades de sistemas y aplicaciones por sus características de particionado de recursos, aislamiento, capacidad de consolidación, seguridad, soporte de aplicaciones heredadas, facilidades de administración, etc.

La tecnología de virtualización, desarrollada a fines de la década del '60 [18], fue prácticamente ignorada durante décadas, pero resurgió a fines de los años '90. Esto se debe a varios factores, tales como: los cambios en los modos de procesar la información, los avances en el hardware, la omnipresencia de Internet y las propias características atractivas que ofrece esta tecnología. Tal es el impacto producido, que dieron origen a nuevas formas de procesamiento, desarrollo y utilización de aplicaciones denominada *Computación en la Nube* (Cloud Computing).

Las tecnologías de virtualización originales o clásicas se basan en una abstracción denominada *Máquina Virtual* (VM). Según se define en [19], una VM es un **duplicado eficiente** y **aislado** de una máquina real, donde:

-
- **Duplicado:** Porque debe comportarse de forma idéntica a una máquina real.
 - **Eficiente:** Porque una mayor proporción de instrucciones deben ser ejecutadas directamente sobre el hardware de la plataforma y una menor proporción de ellas debe ser emulada.
 - **Aislado:** Porque debe ofrecer aislamiento de:
 - *Rendimiento:* El abuso en la utilización de recursos por parte de un OS o una aplicación dentro de una VM no debe afectar el rendimiento de otras VMs.
 - *Fallos:* La ocurrencia de un fallo en el OS o aplicación que se ejecuta dentro de una VM no debe comprometer el funcionamiento de las otras VMs.
 - *Seguridad:* Un incidente de seguridad ocurrido dentro de una VM no debe comprometer la seguridad de las otras VMs.

El Hipervisor o Monitor de Máquina Virtual (VMM: Virtual Machine Monitor) es el componente fundamental en varias de las tecnologías existentes. Un VMM es el software que permite crear, ejecutar, modificar y finalizar VMs. Se entiende entonces por VM *al entorno de ejecución creado por el VMM para la ejecución de programas, presentándose esencialmente equivalente a una máquina real*, a excepción de una pequeña reducción en su rendimiento. Esa reducción de rendimiento se debe a que determinado tipo de instrucciones (*sensibles*) que son ejecutadas por los programas dentro de una VM terminan siendo emuladas por el VMM.

Los sistemas de virtualización son capaces de virtualizar un conjunto completo de recursos de hardware, incluyendo al procesador (o procesadores), memoria, recursos de almacenamiento y dispositivos periféricos. De todos modos, desde su surgimiento a la actualidad la tecnología virtualización ha progresado y se presentan múltiples variantes o alternativas.

Una de las características más interesantes y útiles de las tecnologías de virtualización es la capacidad de consolidar múltiples VMs en un único servidor físico brindando entornos de ejecución aislados, seguros y protegidos a las aplicaciones críticas. Al reducir la cantidad de servidores físicos se incrementa la utilización de cada uno de ellos y se mejora la eficiencia energética de la infraestructura física en general. Esta reducción en la cantidad de servidores impacta directamente en el espacio físico requerido, en el consumo energético de los propios servidores y, como éstos generan calor, en el consumo energético de los sistemas de acondicionamiento ambiental necesarios para que operen dentro del rango térmico especificado por sus fabricantes.

En 1974, Popek y Goldberg [19] establecieron los *requerimientos para virtualización*, los cuales son un conjunto de condiciones suficientes para que una arquitectura de computadores soporte eficientemente la virtualización de hardware. Estos requerimientos son:

- *Equivalencia/Fidelidad*: Un programa ejecutando bajo el control del VMM debe exhibir un comportamiento esencialmente idéntico a aquel demostrado cuando se ejecuta directamente en una máquina equivalente.
- *Control de recursos/Seguridad*: El VMM debe tener el control completo de los recursos virtualizados.
- *Eficiencia/Performance*: Una fracción estadísticamente dominante de las instrucciones de máquina debe ser ejecutada sin la intervención del VMM.

Algunas de las tecnologías de virtualización no requieren de un VMM, como por ejemplo *Virtualización basada en Sistema Operativo*. En ésta, es el propio OS el que construye múltiples entornos de ejecución denominados *Contenedores* [20], *Zonas* [21] o *Prisiones* [22], a los que se referirá genéricamente como Contenedores. Cada Contenedor tiene su propio espacio de nombres (usuarios, PIDs, IPCs, Sistema de Archivos, etc.), y su propio conjunto de recursos (CPU, memoria, E/S, red, etc.).

La Fig. 7 presenta una taxonomía de tecnologías de virtualización existentes, las cuales serán descritas y revisadas en las siguientes secciones.

2.1.1. Virtualización de Hardware/Sistema

Las CPUs modernas disponen de un conjunto de instrucciones (ISA: Instruction Set Architecture) en donde un subconjunto de ellas son instrucciones privilegiadas. Sólo el código ejecutado con suficientes privilegios, tal como el kernel de un OS, puede ejecutar dichas instrucciones. Las VMs encapsulan OSs, a los que se los denomina como Invitados (OS-Guest), creándoles la ilusión de estar ejecutándose sobre un computador real. Los kernels o núcleos de los OS-Guests creen disponer de todos los privilegios para operar sobre el hardware (modo kernel o supervisor), aunque realmente se ejecutan con privilegios de modo usuario.

Los kernels de los OS utilizan instrucciones privilegiadas para controlar el hardware, para controlar la forma de operar de la CPU, para configurar los mecanismos de administración de memoria, para controlar el sistema de interrupciones, de DMA, etc., por lo que lo hacen frecuentemente. Si el OS-Guest ejecuta en modo usuario instrucciones privilegiadas, tales como aquellas que se utilizan para controlar puertos de Entrada/Salida o inhabilitar interrupciones, al disponer de menores privilegios de los requeridos se producirán fallos de protección (faults) en la CPU. El tratamiento de estos fallos es derivado al hipervisor o VMM el cual toma el control de la ejecución permitiéndole emular el comportamiento del hardware o realizar operaciones en representación del OS-Guest.

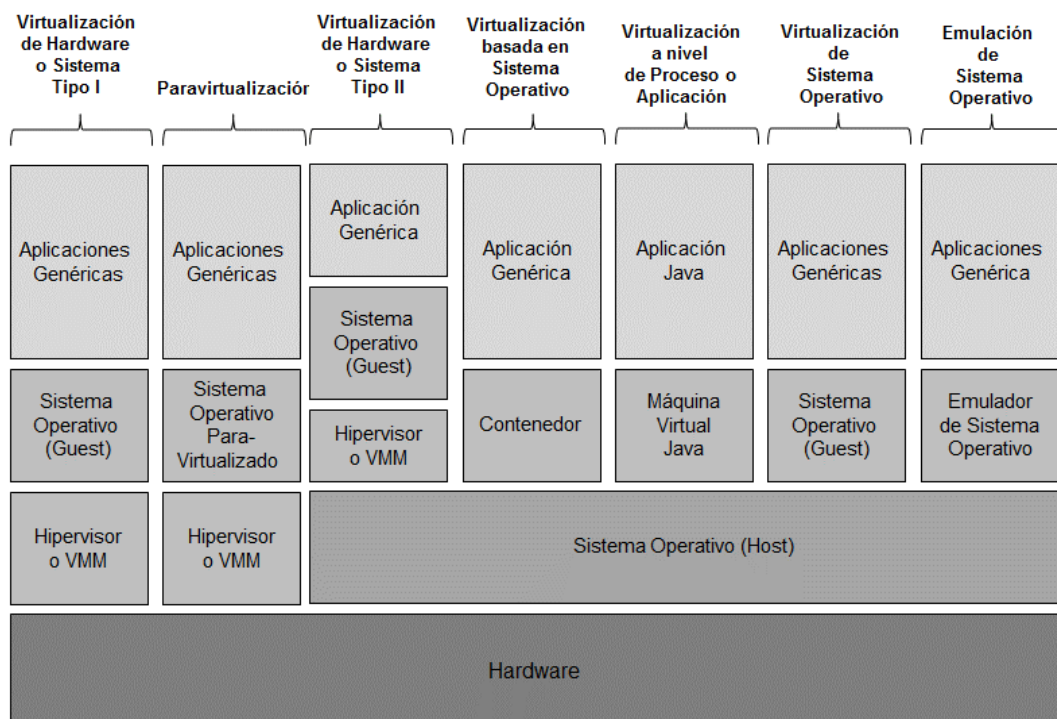


Figura 7. Taxonomía de Tecnologías de Virtualización.

El VMM puede ejecutar directamente sobre el hardware (Tipo-I) o utilizar servicios de un OS (Tipo-II). Las características de cada uno son:

❑ **Virtualización de Hardware Tipo I o Virtualización Completa (Full Virtualization):**

Esta tecnología representa el concepto de virtualización propuesto originalmente por Popek y Goldberg [19] en 1974. En ella, el VMM es el responsable de la gestión de todo el hardware de la plataforma como así también de virtualizar CPU, conjunto de chips de la arquitectura, memoria y dispositivos de Entrada/Salida. El VMM le oculta al OS-Guest la plataforma real de ejecución presentándole un computador abstracto, incluso compuesto de dispositivos físicamente

inexistentes, los cuales son emulados (Fig. 8-A). Por esto, no se requiere ninguna característica particular del OS-Guest que se ejecuta dentro de una VM, excepto que utilice el mismo ISA.

Por el ocultamiento realizado por el VMM, el OS-Guest asume que está controlando un hardware real y por lo tanto utiliza las mismas instrucciones tal como si lo fuese. Muchas de esas instrucciones constituyen un conjunto de instrucciones denominadas *sensibles*, las cuales deben ser emuladas por el VMM para llevar a cabo la virtualización del hardware. Para que dichas instrucciones puedan ser emuladas, el VMM debe tomar el control de la ejecución cada vez que el OS-Guest las ejecuta. Por esta razón, el OS-Guest debe ejecutar con la CPU en modo usuario de tal modo que VMM pueda *atrapar* (dar tratamiento a) las instrucciones sensibles invocadas por el OS-Guest. En la arquitectura Intel 386 original, el conjunto de instrucciones *sensibles* no estaba incluido en el conjunto de instrucciones *privilegiadas*, por lo que para poder llevar a cabo la virtualización debieron utilizarse varios mecanismos ingeniosos que introducían una sobrecarga adicional al sistema [23]. Actualmente, la mayoría de las CPUs de Intel (y AMD) disponen de soporte para la virtualización en las que se deriva el tratamiento de las instrucciones *sensibles* al VMM, por lo que esto dejó de ser un problema a resolver por los fabricantes de hipervisores para estas arquitecturas de CPU.

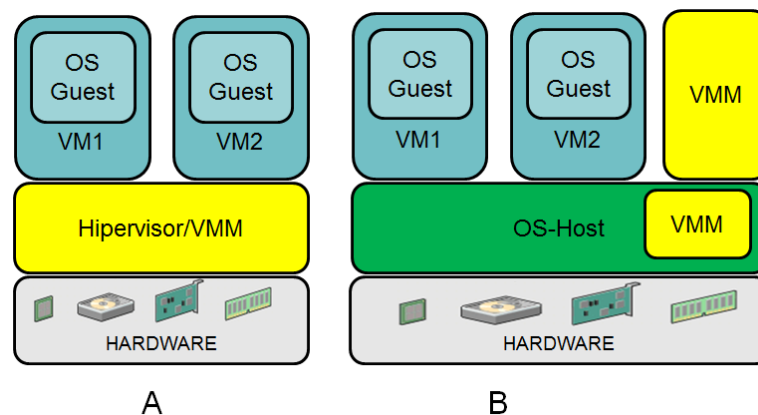


Figura 8. Virtualización de Hardware

Generalmente, los OS también dividen el procesamiento en modo usuario y modo kernel o supervisor. Cuando OS se ejecuta dentro de una VM, la gestión del modo de ejecución también es virtualizada por el VMM. En el momento en que un proceso o aplicación en modo usuario ejecuta una instrucción privilegiada, ésta debería ser atrapada y el OS actuaría en consecuencia, generalmente enviando una señal (signal) a esa aplicación. En cambio, si el OS se ejecuta dentro de una VM como invitado, será el VMM el que atraparé la ejecución de esa instrucción privilegiada en modo usuario. El VMM luego invocará al Gestor de Fallos correspondiente al OS-Guest, obteniéndose el mismo resultado tanto para la ejecución en máquina real como en una VM.

En cambio, cuando el OS-Guest ejecuta una instrucción privilegiada, ésta es atrapada por el VMM, el cual analiza la instrucción conjuntamente con el estado actual de la VM y emula la instrucción para esa VM, retornando luego el control al OS-Guest. Ejemplos de instrucciones privilegiadas son la activación/desactivación de interrupciones (STI/CLI en Intel). Las mismas son emuladas simplemente manteniendo un bit en una estructura de datos interna del VMM correspondiente al estado de cada VM. Si el bit está prendido (STI) cuando se produce una interrupción, el VMM ejecutará la *Rutina de Servicio de Interrupción* (ISR: Interrupt Service Routine) correspondiente establecida por el OS-Guest. Si el bit está apagado (CLI), no se invocará la ISR del OS-Guest. A la invocación de estas ISRs se las denomina interrupciones virtuales. Como ejemplo de interrupción virtual podemos mencionar la producida por el Temporizador (Timer) cuando se cumple un determinado plazo. No se enviará a la VM una interrupción de temporizador en el momento en que se produce una interrupción del temporizador real por lo que no deben hacerse suposiciones respecto a precisiones temporales.

Otro ejemplo son las interrupciones virtuales de una Tarjeta Virtual de Interface de Red (vNIC: Virtual Network Interface Card), dado que son dispositivos de hardware emulados por software, no producen interrupciones reales. Aun así, el VMM invocará las ISR del OS-Guest correspondiente tal como lo haría una interrupción producida por un hardware real.

Dentro de las ventajas que brinda la Virtualización tipo I podemos mencionar:

- Dentro de las VMs se pueden ejecutar OS-Guests obsoletos y sin soporte de dispositivos actuales mediante la emulación de dispositivos y arquitecturas discontinuadas. Por ejemplo, se puede presentar al OS-Guest un disco tipo SCSI o SATA como si fuese un tipo IDE o un archivo de imagen como si fuese una unidad de diskette ya desaparecida, o una tubería (pipe) tal como si fuese un puerto serie RS-232.
- El OS-Guest no necesita ninguna modificación para ejecutar dentro de una VM.
- Por sobre el mismo VMM se pueden ejecutar múltiples VMs con diferentes OS-Guests.
- Las VMs pueden migrarse (en forma automática o por gestión manual) desde un host a otro por cuestiones relacionadas a la distribución de carga entre los nodos del cluster de virtualización, o por la advertencia de un fallo de algún componente de la infraestructura, o para consolidar servidores y así mejorar el desempeño energético del cluster.

-
- Las VMs pueden portarse de un computador a otro en dispositivos de almacenamiento portátiles o mediante la transferencia por red de los archivos que las componen.
 - Las VMs pueden clonarse como respaldo o para facilitar su distribución y así reducir los tiempos de aprovisionamiento.
 - Las VMs tienen excelentes características de aislamiento de rendimiento, de fallos y de seguridad.

Como desventajas que presenta esta tecnología son:

- Cada instrucción sensible ejecutada por el kernel del OS-Guest dentro de una VM será atrapada y emulada por el VMM. El kernel de un OS y sus Gestores de Dispositivos (Device Drivers) hacen un uso frecuente de instrucciones privilegiadas (sensibles), las que les permiten controlar el hardware de la plataforma y de los dispositivos periféricos. Por cada ejecución de una esas instrucciones dentro de la VM, la CPU produce un *fallo de protección* que es atrapado por el VMM, el cual debe emular el comportamiento de esa instrucción para la VM. La alta frecuencia de invocaciones a este tipo de instrucciones produce una sobrecarga que reduce el rendimiento del sistema.
- Por la misma razón mencionada arriba, la latencia de ciertas operaciones se ve incrementada por la emulación de instrucciones. Esto impacta directamente sobre el tiempo de respuesta de las aplicaciones.
- Si en un mismo host se ejecutan VMs con múltiples instancias del mismo OS-Guest, existirían múltiples copias del OS-Guest utilizando memoria del VMM (mayor consumo de memoria). Actualmente, algunos VMMs aplican técnicas que permiten compartir páginas de memoria pertenecientes a diferentes VMs (deduplicación). Esto reduce el consumo de memoria para los casos en que los OS-Guests y eventualmente las aplicaciones dentro de las VMs son idénticas.
- Existe un doble mecanismo de administración de memoria. El hipervisor gestiona la memoria de las diferentes VMs, que cantidad de memoria real le es asignada a cada VM, cuáles páginas de memoria son enviadas al almacenamiento secundario (swap-out), y cuáles páginas son rescatadas de él (swap-in). Por otra parte, los OS-Guest gestionan la memoria de las diferentes aplicaciones que se ejecutan sobre él aplicando su propia política.

-
- El VMM es responsable de administrar el tiempo y cantidad de CPUs asignados a cada VM en ejecución. El tiempo de cada CPU es particionado entre las VMs conforme a una política de planificación del VMM. De todos modos, en muchos casos el VMM ignora qué tipo de tareas están siendo ejecutadas en la VM, y si el uso que le dan a la CPU representa trabajo útil o no. Es imposible que el VMM puede conocer acerca del uso del tiempo del OS-Guest y sus aplicaciones dado que el VMM solo se activa por eventos (trampas, fallos, excepciones, interrupciones). En consecuencia, un OS-Guest puede estar desperdiciando tiempo de CPU realizando un ciclo improductivo el cual podría asignarse a otra VM.

 - Existe una doble planificación del tiempo de CPU y de prioridades con políticas diferentes entre el VMM y los OS-Guests. Las aplicaciones dentro de una VM deben esperar primero la planificación de la VM por parte del VMM, y luego la planificación de la aplicación por parte del OS-Guest.

 - El particionado equitativo del tiempo de CPU y de otros recursos hacen que el VMM incremente su latencia, la cual impacta directamente en las aplicaciones. Esta característica hace a esta tecnología inadecuada para ser utilizada en *Sistemas de Tiempo Real*. En éstos deben cumplirse con plazos para la finalización de las tareas (deadline) o con límites establecidos para los tiempos de respuesta.

 - Las interrupciones de hardware son tratadas por el VMM, el cual eventualmente transforma esa interrupción real en interrupciones virtuales en cada una de las VMs que así lo hubiesen requerido. Este doble tratamiento de las interrupciones, sumado a los tiempos de planificación del VMM, incrementa la latencia de las ISR de los OS-Guests.

 - Generalmente las llamadas al sistema de los OSs se implementan utilizando Trampas (Traps) o Compuertas (Gates). Estas son atrapadas por el VMM, el cual invoca a través del descriptor o vector correspondiente la función del OS-Guest que dará tratamiento a la llamada al sistema requerida. Los ingenieros de VMware [24] realizaron mediciones de los tiempos de llamadas al sistema en un Pentium 4 de 3.8 GHz obteniendo un consumo de 242[Hz] para una llamada al sistema para un OS sobre hardware nativo y de 2308[Hz] para el mismo OS de 32 bits como Guest en una VM. Esto demuestra que la tasa de llamadas al sistema por unidad de tiempo se reduce en un factor de 10 aproximadamente al ejecutar como OS-Guest.

Las características particulares de la virtualización de hardware Tipo I son muy atractivas para múltiples usos, pero el impacto en el rendimiento (CPU, memoria, latencia, etc.) obligó a muchos proveedores de Servicios en la Nube de tipo IaaS a analizar alternativas de virtualización que les permitan colocar una mayor densidad de VMs por cada servidor físico. No siendo ajenos a estos problemas, los fabricantes de hardware llevaron a cabo mejoras y adiciones a sus arquitecturas de CPU y chipsets a fin de reducir la sobrecarga producto de la virtualización y aumentar así la eficiencia. A la tecnología de virtualización que utiliza estas características provistas por el hardware se la denomina Virtualización Asistida por Hardware (HAV: Hardware-Assisted Virtualization) [25, 26]. Varios productos de software de virtualización solo admiten su utilización en computadores que soportan HAV, dado que ésta simplifica sustantivamente el desarrollo y mantenimiento del VMM.

Generalmente, la virtualización de dispositivos almacenamiento se realiza utilizando archivos. Por ejemplo, un disco rígido asignado a una VM generalmente es un archivo del sistema de virtualización. El VMM deberá entonces disponer de su propio Sistema de Archivos (FileSystems) para contener imágenes de discos rígidos, imágenes de CD/DVDs, archivos de configuración de VMs, archivos de registros (logs), etc. La criticidad de estos archivos obliga a los proveedores de IaaS a disponer de productos de respaldo y recuperación específicos. Por otro lado, al incrementarse la cantidad de servicios de un VMM, se incrementa el tamaño del código y por lo tanto se incrementa la Base de Computador Confiable (TCB: Trusted Computing Base) del VMM, reduciendo la seguridad y confiabilidad de la plataforma de virtualización.

Al igual que lo que ocurre con los OS, conforme evolucionan los dispositivos periféricos, deben disponerse de los Gestores de Dispositivos correspondientes, de otra forma el VMM no podrá utilizarlos. Esto prácticamente obliga a los fabricantes de hardware a desarrollar los Gestores de Dispositivos para cada VMM en particular, si es de su interés vender sus productos para que sean utilizados en servidores dedicados a la virtualización. O viceversa, los fabricantes de productos de virtualización deberán desarrollar los Gestores de Dispositivos respectivos, si es de su interés que su producto pueda ejecutar o utilizar un nuevo dispositivo periférico. No disponer de un gestor de un nuevo tipo de dispositivo para un sistema de virtualización representa un problema para los proveedores de Servicios en la Nube. Esto, aparentemente trivial, tiene impacto considerable cuando se analizan las ventajas de otras alternativas de virtualización.

Por todo lo mencionado anteriormente respecto a los componentes y al tamaño resultante en líneas de código fuente (SLOC: Source Lines Of Code) de los VMMs, éstos se asemejan cada vez más a los OSs. Existen varios artículos de investigación que analizan la similitud y

diferencias entre ambos [27, 28, 29]. En la Tabla I, se comparan características de un OS y un VMM.

TABLA I. COMPARACIÓN ENTRE OS Y VMM

	OS	VMM
Planificación	De procesos	De VMs
Gestión del Tiempo	Para el Sistema y procesos	Para Sistema y VMs
Sistema de Archivos	Para uso del Sistema y de sus usuarios	Para Imágenes de VMs, Discos Virtuales, Archivos de configuración
Gestores de Dispositivo	Para los dispositivos soportados	Para los dispositivos soportados
Pila de protocolos TCP/IP	De uso genérico	Para los administradores, accesos a dispositivos externos, acceso a interface de gestión Web
Usuarios	Genéricos	Administradores
Terminales	De caracteres, gráficas, virtuales y remotas	Consolas de VMs

Algunos VMMs, tales como XEN [30] o KVM [31], utilizan las facilidades de las que dispone un OS tan difundido y adoptado como Linux para no tener que reescribir componentes ya resueltos por el OS. Microsoft incorpora Hyper-V [32] como parte del OS Windows dado que éste dispone de muchas de las facilidades requeridas por su VMM.

La tecnología de virtualización de Hardware tipo I es utilizada en productos tales como VMware vSphere, KVM, XEN y Microsoft Hyper-V.

❑ **Virtualización de Hardware/Sistema Tipo II:**

En esta tecnología, el VMM hace recaer la gestión del hardware de la plataforma en un OS-Host subyacente. Parte del VMM se encuentra como módulo del kernel de un OS-Host, y otra parte se encuentra como programa aplicación en modo usuario del OS-Host (Fig. 8-B).

Al igual que la Virtualización de Hardware Tipo I, no se requiere ninguna característica particular del OS-Guest para ejecutar dentro de una VM excepto que éste utilice el mismo ISA.

A diferencia del Tipo I, cuando el OS-Guest ejecuta una instrucción sensible, ésta es atrapada por el OS-Host, derivando su tratamiento al componente del VMM embebido en su kernel. Lo mismo sucede con las trampas, excepciones y fallos que ocurren dentro de una VM.

Las interrupciones son tratadas por el OS-Host, el VMM es notificado de su ocurrencia a fin de, eventualmente, tomar acciones en sus VMs. Por ejemplo, la interrupción del temporizador es

tratada por la ISR del OS-Host, el cual invoca en forma inmediata o diferida a una función del OS-Guest para notificarlo de la ocurrencia de la misma.

La planificación para la ejecución de VMs, la gestión de la memoria y la gestión de dispositivos ya no son de exclusiva responsabilidad del VMM, sino que éste utiliza servicios del OS-Guest.

Muchas de las ventajas de esta tecnología de virtualización son las mismas que las de Tipo I, a excepción de sus características de aislamiento de rendimiento, de fallos y de seguridad ya no dependen exclusivamente del VMM, sino también de la seguridad del OS-Host. Como generalmente el OS-Host tiene un TCB mayor a un VMM, se incrementa la probabilidad de que se produzcan errores de software lo que reduce su confiabilidad, haciéndolo más vulnerable a fallos de seguridad al incrementar la superficie de ataque.

En cuanto a las desventajas de este sistema de virtualización, las mismas son equivalentes a la del Tipo I, pero más acentuadas debido al mayor camino crítico que debe recorrer la ejecución de la emulación de instrucciones sensibles. Como consecuencia, se produce un incremento de la sobrecarga del sistema y una disminución de rendimiento. Además, como se mencionó anteriormente, el responsable de la planificación de CPU es el OS-Host, esto incluye administrar el tiempo y cantidad de CPUs asignados a cada VM en ejecución dado que al OS-Guest se le da el mismo tratamiento que a las aplicaciones genéricas.

Las características particulares de la virtualización de hardware Tipo II son muy atractivas para ser utilizadas por desarrolladores que instalan el sistema de virtualización por sobre el sistema operativo de preferencia. Se utiliza también en la investigación de la operación de códigos maliciosos (virus, rootkits, etc.) ejecutándolos en un OS-Guest dentro de una VM y monitoreándolos desde el OS-Host (sandbox). También son muy útiles para experimentación por lo que puede utilizarse en educación, investigación y desarrollo. De hecho, el prototipo de DVS se desarrolló utilizando un cluster virtual en un sistema de virtualización de hardware Tipo II (VMWare Workstation Player).

A diferencia de la virtualización de hardware Tipo I, el fabricante del sistema de virtualización de hardware Tipo II no requiere disponer de gestores para utilizar nuevos dispositivos; se utilizan los Gestores de Dispositivos del OS-Host. Tampoco se requiere que el sistema de virtualización disponga de un sistema de archivos (ni de herramientas para administrarlo) para utilizar en la virtualización de dispositivos (discos rígidos, CD/DVD, etc.). Simplemente el VMM utiliza el sistema de archivos que está disponible en el OS-Host, incluso independiente del dispositivo

físico donde éste se encuentre (Pendrive, Tarjeta SIM, disco rígido, SSD, sistema de archivos remoto como NFS, etc.).

Algunos productos que utilizan virtualización de Hardware Tipo II son VMWare Workstation Player y Oracle Virtual Box [33].

2.1.2. Paravirtualización

Como se mencionó en la [Sección 2.1.1](#), las tecnologías de virtualización de hardware provocan la invocación del VMM cada vez que se producen fallos, trampas, excepciones o interrupciones. El OS-Guest utiliza frecuentemente instrucciones privilegiadas para gestionar el hardware de la plataforma y de E/S, pero como lo hace en modo usuario éstas provocan fallos que son atrapados por el VMM. En general, las VMs no gestionan en forma directa el hardware o los dispositivos periféricos (pass through), en su lugar se les asignan dispositivos virtualizados (CPU, memoria, discos, consolas, DVDs, etc.) por el VMM. Esto lleva a que el OS-Guest ejecute código inútil cuyas instrucciones privilegiadas deben ser emuladas por el VMM para operar sobre un hardware inexistente.

A los fines de reducir la sobrecarga del sistema, por la emulación de instrucciones privilegiadas sobre un hardware inexistente y la frecuencia de ejecución de las mismas, se desarrolló la tecnología de Paravirtualización. Para poder utilizarla, se requiere modificar el código del OS-Guest y sus Gestores de Dispositivos para que ejecuten en un entorno virtualizado que *coopera* con el VMM. El OS-Guest (Paravirtualizado) solicita servicios al VMM a través de las que se denominan Llamadas al Hipervisor (*Hypervisor Calls* o *hypercalls*). Estas ofrecen a los OS-Guest servicios de gestión de la plataforma y de dispositivos virtuales, y mecanismos optimizados de transferencias de datos entre el OS-Guest y el VMM. Con el fin de normalizar o estandarizar las interfaces entre los OS-Guests y el VMM se propuso el estándar VMI [34] (del inglés Virtual Machine Interface) que evitaría que el OS-Guest deba desarrollarse en forma específica para un VMM paravirtualizado en particular. Al reducir la sobrecarga del sistema en lo que a procesamiento y memoria se refiere, ésta tecnología permite una mayor densidad de VMs por cada hosts físico y mayor escalabilidad.

La paravirtualización ya fue utilizada en VM/370 [19], Disco [35] y Denali [36] y actualmente se encuentra incorporada en productos tales como Xen [30], Hyper-V [32] y Virtual Box [33]. VMware, en cambio, ha optado por incorporar solo paravirtualización de dispositivos, es decir que no se requiere modificar los OS-Guests sino incorporarle los Gestores de Dispositivos correspondientes para operar sobre dispositivos paravirtualizados.

Las tecnologías de OS basados en microkernel comparten características con los VMM o hipervisores. Existen varios trabajos de investigación [37, 38, 39] donde se explora la convergencia de características y objetivos de ambas tecnologías. Otro ejemplo es Mhyper [40], un OS/hipervisor de microkernel basado en Minix [41] desarrollado por el autor de esta tesis que soporta paravirtualización entre otras modalidades de virtualización.

2.1.3. Virtualización basada en Sistema Operativo

La virtualización basada en OS es un método de particionado por el cual el kernel del OS crea diferentes entornos o dominios de ejecución (*Contenedores, Prisiones, Zonas*) en espacio de usuario para ejecutar grupos de procesos, en lugar de un único entorno como es lo habitual. Todos los entornos o dominios comparten el mismo OS (Fig. 9).

Los OSs ofrecen una *Interfaz de Programación de Aplicaciones* (API: Application Programming Interfaces) y *Llamadas al Sistema* (System Calls o Syscalls) utilizadas por las aplicaciones para solicitar servicios al OS. La virtualización basada en OS consiste en interceptar las Llamadas al Sistema y alterar su comportamiento para simularle a la aplicación un entorno específico.

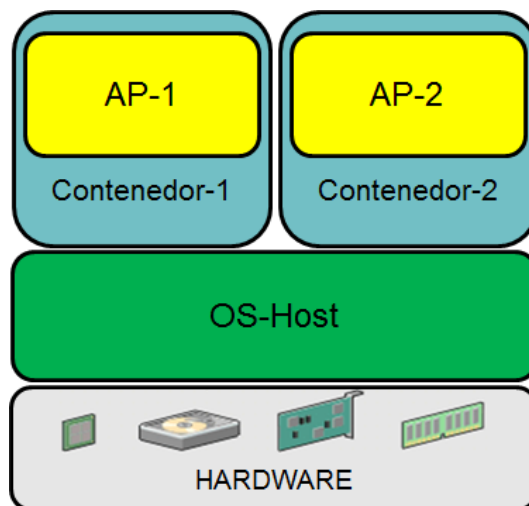


Figura 9. Virtualización basada en Sistema Operativo.

El origen de esta tecnología parte de los entornos *chroot*, en donde a un proceso, a todos sus hijos y a toda su descendencia se les asigna un pseudo Sistema de Archivos *root* que es mapeado a un subdirectorio del Sistema de Archivos *root* real. De esta forma, los procesos que ejecutan en este entorno (conocido como prisión *chroot*) no disponen de acceso a archivos y directorios que están por fuera de su sistema de archivos *root* en el árbol de directorios (aislamiento de Sistema de Archivos).

El sistema de virtualización basada en OS es una extensión de *chroot* que construye un Contenedor [20], Zona [21] o Prisión [22] que confina a un conjunto de aplicaciones en su propio entorno de ejecución. Se presenta como más adecuado referirse a él como *Entorno Virtual* y no *Máquina Virtual* dado que no se virtualiza Hardware.

Si bien Linux no dispone de Contenedores como tales, ofrece todas las herramientas y recursos abstractos para construirlos. Por ejemplo, por cada Contenedor se crea un espacio de nombres (*namespaces*) propio con sus PIDs, *hostnames*, puntos de montaje, IPC, UIDs, etc. Esto no suele ser suficiente para considerarlo un aislamiento equivalente a una VM de hardware. Por ejemplo, un proceso dentro de un Contenedor que sea altamente demandante de recursos puede impactar negativamente en el rendimiento del resto de los Contenedores. Para reducir el impacto que producen situaciones como la mencionada, Linux (y otros OSs) prevén mecanismos de asignación y control de recursos (CPU, memoria, utilización de recursos de red, utilización de dispositivos de bloques, utilización de caches, etc.) a grupos de procesos (*Cgroups* [44]).

Las ventajas comparativas de esta tecnología con las otras vistas anteriormente son:

- Mejor Rendimiento*: Los Contenedores son entornos de ejecución para conjuntos de procesos que se construyen sobre el mismo OS. Esto evita que se deba disponer de múltiples copias en memoria de diferentes OS como en el caso de la Virtualización de Hardware o Paravirtualización. Por otro lado, existe un único planificador, el del OS, el cual reconoce de cada proceso que gestiona la utilización de recursos que éste hace y por lo tanto, puede lograr una planificación más equitativa y un mejor aprovechamiento de los recursos. También la memoria es gestionada por el OS, por lo que puede tomar mejores decisiones respecto a la asignación de este recurso en particular como por ejemplo, cuáles páginas de un proceso dentro de un Contenedor deben desplazarse hacia el dispositivo de intercambio de páginas (swap) o viceversa. Lo mismo sucede con la gestión de E/S, buffers, caches, interfaces de red, etc.
- Alta Densidad/Escalabilidad* [42]: El mayor rendimiento y menor sobrecarga permite una alta densidad de Contenedores por computador físico, característica ésta muy apreciada por los proveedores de IaaS respecto a las tecnologías anteriormente mencionadas (Virtualización de Hardware y Paravirtualización).
- Rápido Arranque*: Cuando se requiere arrancar un Contenedor con sus procesos, el tiempo que le demanda es prácticamente idéntico al arranque de esos procesos sin Contenedores [43].

Las limitaciones que presenta esta tecnología son:

-Aislamiento Discreto: Todos los Contenedores de un computador se ejecutan sobre un único OS. Generalmente los OSs más utilizados son monolíticos y con una gran TCB, por lo que los hace más vulnerables. Si existiese una vulnerabilidad (por ejemplo, un desborde de buffer) en el kernel del OS que pudiese ser explotada por un proceso de usuario dentro de un Contenedor, todo el OS y eventualmente todos los Contenedores serían afectados por este ataque. También, el aislamiento de rendimiento está limitado dado que todos los Contenedores comparten los recursos del mismo OS tales como buffers, caches, CPUs, interfaces de red, etc. Este aspecto fue recientemente solucionado con la implementación de lo que se denomina Cgroups (Control groups) [44] en el kernel de Linux. Cgroups permite asignar y controlar la utilización de recursos tales como CPU, memoria, dispositivos de bloques e interfaces de red a cada grupo de procesos.

-Unico Sistema Operativo: A diferencia de las tecnologías mencionadas anteriormente, no se pueden ejecutar OS diferentes.

Con este mecanismo de virtualización se pueden ejecutar múltiples entornos en forma aislada y segura pero todos ellos hacen uso de los servicios del mismo OS-Host. Esta tecnología es utilizada por Linux LXC/LXD [45], OpenVZ/Virtuozzo [46], Linux-VServer [47], Solaris Zones [21], Windows Containers [48], FVM [49] y FreeBSD Jails [50] entre otros.

2.1.4. Virtualización de Lenguaje/Proceso/Aplicación

Las aplicaciones utilizan las interfaces que le ofrece un OS tales como Llamadas al Sistema y APIs. La tecnología de *Virtualización de Aplicación* permite encapsular una aplicación a fin de no modificar esas interfaces aun cuando se ejecute en otro OS. En el OS-host, la aplicación no se ejecuta de igual forma que en el OS original para el que la misma se desarrolló. Para hacerlo, se crea un entorno de ejecución cuando el proceso arranca, y se termina cuando éste finaliza. Para ejecutar la aplicación se requiere encapsularla conjuntamente con un módulo de tiempo de ejecución (runtime), el cual intercepta las Llamadas al Sistema originales y las convierte a Llamadas al Sistema del OS-host. Generalmente se encapsula una única aplicación.

Algunos productos que utilizan esta tecnología son: Citrix XenApp, Microsoft App-V, Oracle Secure Global Desktop, Symantec Workspace Virtualization y VMware ThinApp, entre otros varios.

La *Virtualización de Lenguaje* es similar a la de aplicación, pero su objetivo es proveer un entorno de programación y ejecución independiente de la plataforma, de los detalles del hardware y del OS subyacente. El ejemplo más conocido es la Máquina Virtual Java (JVM). La gran ventaja de estas tecnologías es la portabilidad de aplicaciones.

2.1.5. Virtualización de Sistema Operativo.

Un OS típico es una capa de software que se encuentra localizada entre las aplicaciones y el hardware. En la tecnología de *Virtualización de Sistema Operativo*, el OS-Guest no gestiona hardware real, sino que opera sobre dispositivos virtualizados provistos por una capa inferior de software tal como un OS-host real. Por esta característica, parece adecuado referirse al OS-Guest como un *Sistema Operativo Virtual* (VOS: Virtual Operating System) [17] (Fig. 10). Son evidentes las similitudes con la tecnología de Paravirtualización, entre ellas la necesidad de modificar el OS-Guest para convertirlo en un VOS.

Un proyecto muy conocido que permite ejecutar múltiples instancias de Linux como OS-Guest sobre otro Linux como OS-Host es User-Mode Linux (UML) [51]. CoLinux [52] es otro proyecto de similares características pero que permite ejecutar un Linux como OS-Guest sobre un Windows como OS-Host. Inferno es otro OS que puede trabajar sobre plataforma de hardware o como VOS [53] sobre Linux, y Windows [54].

Otro proyecto no tan conocido, desarrollado por el autor de esta tesis, se denomina MoL (Minix over Linux) [55]. MoL permite ejecutar múltiples instancias de un VOS multiservidor derivado de Minix [41] sobre un Linux como host. En este caso, para construir los mecanismos de IPC de Minix se utilizaron *enchufes* o *sockets* (opcionalmente UNIX, TCP o UDP). Esto habilitaba a que los procesos de la misma instancia de MoL pudiesen ejecutarse en diferentes hosts, siendo esta la versión germinal del objeto de esta tesis. Esa versión de MoL estaba constituida por un pseudo-kernel de Minix ejecutando en modo usuario de Linux y operaba a modo de proceso coordinador de IPC. Cada vez que un proceso MoL requería enviar un mensaje a otro proceso MoL, la transferencia se implementaba enviando un mensaje al pseudo-kernel, el cual lo retransmitía al proceso MoL destinatario; es decir, se realizaban dos transferencias de mensajes reales por cada transferencia invocada por un proceso MoL.

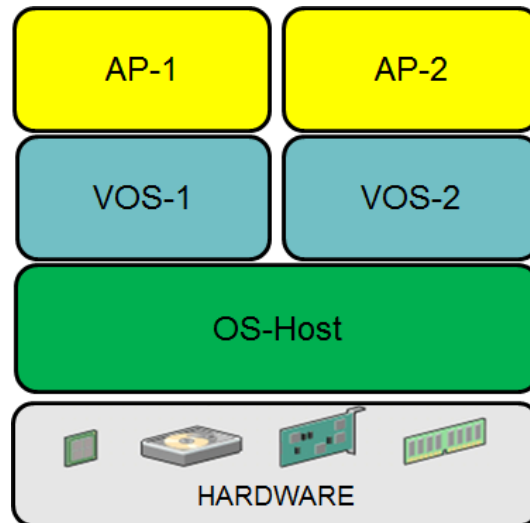


Figura 10. Virtualización de Sistema Operativo.

Por otro lado, la utilización de los mecanismos de IPC propios de Linux colaboró a que el rendimiento fuese una de sus debilidades. Para mejorar el desempeño, el autor decidió implementar un mecanismo de IPC embebido directamente en el kernel de Linux al que denominó M3-IPC [56, 57], el cual será tratado en detalle en capítulo correspondiente al prototipo del DVS.

Las ventajas e inconvenientes de la tecnología de virtualización de OS son similares a las de Paravirtualización. Una característica muy apreciada se da en el caso de los OS-Guests multiservidor que utilizan mecanismos de IPC entre procesos localizados en diferentes hosts. Esta habilita la posibilidad de ejecutar procesos de una misma instancia de un OS-Guest más allá de los límites de un computador, transformándose entonces en una aproximación a un SSI-DOS.

2.1.6. Emulación de Plataforma

Si bien Emulación y Virtualización son conceptos diferentes, éstos están muy relacionados dado que en la Virtualización (de Hardware) se emulan las instrucciones sensibles como se mencionó en la [Sección 2.1.1](#). Un software Emulador hace que un computador (host) se comporte como otra plataforma completa (guest), incluso con diferente ISA, de tal forma que se puedan ejecutar programas y OSs desarrollados para esa otra plataforma. En general, las instrucciones de máquina se interpretan y se ejecutan en forma dinámica, esto hace que su rendimiento sea sustantivamente inferior a la de cualquier forma de virtualización.

Los emuladores para plataforma x86 de Intel más conocidos son Bochs [58] y QEMU [59] siendo este último parte del proyecto de virtualización del kernel de Linux denominado KVM [31] y

también Xen [30]. Existen varios emuladores de teléfonos inteligentes (smartphones), de consolas de juegos, de plataformas discontinuadas, etc. Una lista de ellos puede encontrarse en wikipedia.

La utilidad de los emuladores radica en su propia definición de poder ejecutar código de una plataforma sobre otra. Los desarrolladores de software aprovechan esta característica en la etapa de desarrollo hasta la que sus aplicaciones se encuentran en un estado de madurez tal que puede llevarse a la plataforma destino real. Los emuladores también son utilizados para el análisis de software malicioso (malware) y la depuración (debugging) de aplicaciones.

2.1.7. Emulación de Sistema Operativo.

La tecnología de *Emulación de Sistema Operativo* emula las Llamadas al Sistema de un OS, sus bibliotecas de enlace dinámico, sus comandos, herramientas y utilidades en otro OS con diferentes Llamadas al Sistema y ABI. El objetivo es poder ejecutar una aplicación desarrollada para un OS origen en otro OS destino.

Por ser los OSs más utilizados, existen varios emuladores de Windows en Linux y viceversa. Un primer emulador de MS-DOS para Linux se denominaba *DOSEmu*, pero actualmente no dispone de soporte y fue reemplazado por *DOSbox* [60]. Un emulador de Windows bajo Linux muy difundido es *Wine* [61]. En 2016 Microsoft anunció el Windows Subsystem for Linux (WSL) el cual permite ejecutar aplicaciones desarrolladas para Linux en Windows 10.

A diferencia de la *Emulación de Plataforma*, en la emulación de OS las instrucciones de máquina se ejecutan directamente sobre el hardware (se requiere el mismo ISA), por lo que su rendimiento es muy bueno y cercano a su ejecución sobre el OS nativo.

2.1.8. Virtualización Reversa.

La tecnología de *Virtualización Reversa* [2, 16] permite integrar los recursos de múltiples computadores (nodos de un cluster) para conformar la imagen de un *Sistema de Multi-Procesamiento Simétrico Virtual* (vSMP: virtual Symetric Multiprocessing System) (Fig. 11). Esta tecnología se basa en un VMM distribuido el cual le virtualiza al OS-guest los recursos de hardware de todos los nodos del cluster en una única VM.

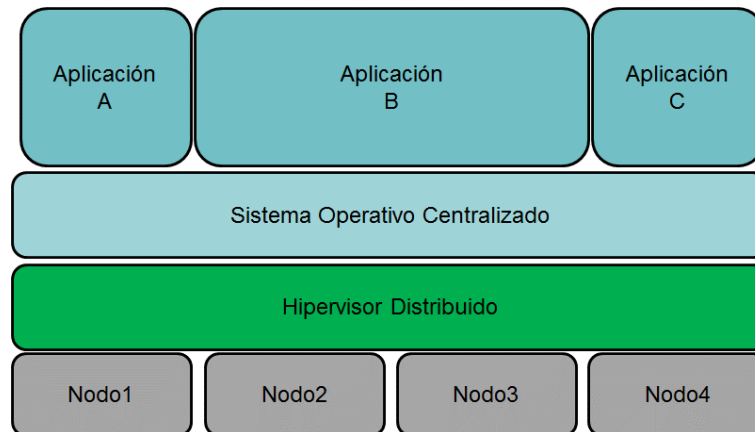


Figura 11. Virtualización Reversa.

La ventaja de esta tecnología es que el OS gestiona los recursos que resultan de la agregación de los recursos de todos los nodos del cluster tal como si fuese un único computador (vSMP). Por otro lado, ese ocultamiento de la realidad le impide al OS distribuir cargas entre los diferentes nodos según su política porque desconoce la existencia de un cluster. La distribución de cargas es entonces una tarea del hipervisor distribuido.

Esta tecnología, es una variante de virtualización de hardware, compartiendo muchas de sus ventajas e inconvenientes, a las que se les incorporan características de sistema distribuido, tales como: Memoria Compartida Distribuida (DSM: Distributed Shared Memory), planificación distribuida, mecanismos de redundancia y tolerancia a fallos.

2.1.9. Capacidades de las actuales Tecnologías de Virtualización

Desde sus primeras implementaciones (fines de los años '60) hasta la actualidad, se encontraron varias características de la virtualización que resultan de interés para su utilización por las áreas de Tecnología de la Información (IT: Information Technology). Las actuales tecnologías de virtualización tienen las siguientes capacidades:

- 1) *Consolidar múltiples VMs o Contenedores en un mismo computador físico:* Esto permite reducir el espacio físico y el consumo energético del equipamiento. Se puede lograr una mayor densidad de servicios para un mismo espacio físico y un menor consumo energético de la infraestructura de servicios de un Datacenter, tales como los sistemas de acondicionamiento ambiental.
- 2) *Crear compartimentos aislados para la ejecución de aplicaciones con mayor seguridad y protección:* Dentro de las características atractivas que ofrece la virtualización está la

capacidad de aislamiento. Existen 3 tipos de aislamiento que debe brindar un VM o Contenedor:

a. Aislamiento de Fallo: Esto refiere a que un fallo que ocurre dentro de una VM o Contenedor no puede afectar a procesos que ejecutan en otras VMs o Contenedores ni al propio sistema de virtualización.

b. Aislamiento de Seguridad: Esto refiere a que un ataque a la seguridad de un sistema contenido dentro de una VM o Contenedor no pueda afectar la seguridad del sistema de virtualización o a otra VM o Contenedor.

c. Aislamiento de Rendimiento: Esto refiere a que el sistema de virtualización debe permitir la ejecución de todas las VMs o Contenedores mediante el particionado de los recursos físicos o abstractos, y que el uso excesivo de un recurso por parte de una VM o Contenedor no debe afectar (apreciablemente) el desempeño de otras VMs o Contenedores.

3) *Migrar VMs o Contenedores entre computadores:* Dado que tanto las VMs como Contenedores se encuentran contenidas y lo suficientemente desacopladas del hipervisor o del OS respectivamente, pueden migrarse entre computadores de forma menos compleja. Esta facilidad puede utilizarse para mejorar la disponibilidad de servicios, como por ejemplo, ante la detección de un fallo de hardware en el host. El administrador, o el propio sistema de virtualización pueden ordenar la migración de todas las VMs o Contenedores que podrían verse afectadas por el fallo hacia otros computadores mejorando así su tiempo de servicio y su disponibilidad. La migración de VMs o Contenedores puede utilizarse también para la redistribución por balanceo de cargas entre los diferentes computadores, mejorando así los tiempos de respuesta de las aplicaciones.

4) *Conformar redes virtuales:* Es frecuente que las aplicaciones que se ejecutan dentro de una VM deban comunicarse con otras aplicaciones que quizás se ejecutan en otras VMs dentro del mismo computador. Generalmente, a las VMs o Contenedores se le asignan vNICs que no necesariamente se asocian a una interface física. Estas interfaces pueden “conectarse” a Switches Virtuales (vSwitch) implementados por el mismo sistema de virtualización o por algún módulo adicional. Dentro de las ventajas de esta facilidad podemos mencionar a:

a. La reducción en la utilización de puertos de switch físicos y sus cableados correspondientes simplificando las instalaciones.

-
- b. La reducción del consumo energético y espacio físico por la menor cantidad de switches físicos requeridos.
 - c. La transferencia a velocidad de bus dentro del propio computador en la comunicación de red entre dos VMs o Contenedores.
 - d. La simplificación en las tareas de mantenimiento e implementaciones por la versatilidad en el conexionado que pueden realizarse directamente desde una consola de administración.
 - e. La posibilidad de implementación de Dispositivos Virtuales (Virtual Appliances) tales como firewalls de red, Sistema de Detección/Prevención de Intrusos, agentes Antivirus o Firewalls de aplicación (WAF: Web Application Firewall) como módulos de software que se encastran dentro del sistema de virtualización. Esto facilita su mantenimiento, conexionado y configuración reduciendo también el consumo energético y espacio físico de la infraestructura.
- 5) *Ejecutar diferentes Sistemas Operativos*: En el caso de la Virtualización de Hardware y Paravirtualización, ambos admiten la ejecución de diferentes OSs en cada VM, los cuales deben utilizar el mismo ISA que la plataforma de virtualización.
- 6) *Permitir la Portabilidad de VMs/Contenedores y Clonarlos*: Tanto las VMs como los Contenedores admiten la portabilidad hacia otros computadores y su clonación. Esto permite disponer de procedimientos de aprovisionamiento más ágiles y elásticos.

Aunque existen diversos tipos de tecnologías de virtualización con diferentes capacidades, ellas habilitaron la creación de nuevos servicios informáticos de carácter global, tales como los mencionados Servicios en la Nube (Cloud Computing). Muchas de las características de la virtualización impulsaron cambios y mejoras en tecnologías de redes (SDN: Software-Defined Networking, NFV: Network Functions Virtualization) y de almacenamiento (vSAN: Virtual Storage Area Network).

2.2. Otras Tecnologías Relacionadas

Entre los requerimientos de diseño para el modelo de arquitectura de DVS se incluyeron capacidades tales que lo habiliten brindar Servicios de Infraestructura (IaaS) en la Nube. Este

requerimiento le impone al modelo características que lo habiliten para lograr, en forma transparente, un mayor rendimiento, una más alta disponibilidad, una mayor escalabilidad y elasticidad.

El *rendimiento* y la *escalabilidad* se logran a partir de disponer de la capacidad de extender, en forma dinámica, un entorno de ejecución más allá de los límites de un nodo de un cluster, y por disponer de servicios que permitan distribuir las cargas entre los diferentes nodos (redundancia y migración de procesos).

La *elasticidad* se logra por la capacidad para incrementar o reducir en forma dinámica la cantidad nodos que abarca un entorno de ejecución para las aplicaciones.

La mayor *disponibilidad* se logra por disponer de mecanismos y servicios que permiten desplegar procesos y datos replicados para tolerar fallos y migrar los entornos de ejecución desde computadores fallidos a computadores operativos.

Si bien muchas de estas características hoy se pueden lograr con otras tecnologías, la característica de transparencia (o de ocultamiento, como se quiera ver) no es común en ellas. La transparencia facilita la programación, despliegue, mantenimiento y operación de las aplicaciones lo que reduce en forma directa tiempos y costos.

Todas estas características del modelo propuesto requirieron de la consideración de otras tecnologías complementarias a la virtualización, las que se describen en las siguientes subsecciones.

2.2.1. Sistemas Operativos Distribuidos con Imagen de Sistema Unico

A diferencia de un vSMP que virtualiza hardware, un SSI-DOS ejecuta procesos en forma distribuida virtualizando abstracciones de software. Estas abstracciones son idénticas o similares a las que brinda un OS tradicional (centralizado) tales como archivos, tuberías (pipes), sockets, colas de mensajes, memoria compartida, semáforos, mutexes, entre otros. Los usuarios y aplicaciones tienen la visión de un único computador virtual conformado por los recursos computacionales, procesos, abstracciones y servicios, aunque éstos se encuentren distribuidos en los distintos nodos componentes del cluster.

Podría interpretarse que un SSI-DOS también representa una forma de virtualización reversa. A diferencia de la virtualización tradicional que permite crear múltiples computadores virtuales en un único computador físico, los SSI-DOS integran en un OS los recursos de múltiples computadores físicos en un único computador virtual.

Las tecnologías de Virtualización y SSI-DOS, aunque aparentan ir en direcciones opuestas, conforman los cimientos para el modelo de arquitectura del DVS objeto de esta tesis.

Según Tanenbaun [62], “*un Sistema Operativo Distribuido es una colección de computadores independientes unidos por una red que aparecen a los usuarios como un único sistema coherente*”. En el modelo de DOS más relajado (Asincrónico), los nodos que componen el cluster no comparten memoria ni reloj común ni tampoco se garantizan tiempos máximos de ejecución de procesos. La red que une esos nodos no garantiza la entrega, ni la latencia máxima de las comunicaciones. Todas estas consideraciones hacen que el diseño, desarrollo y programación de un DOS sean significativamente más complejos que para un OS centralizado.

La Fig. 12 muestra un cluster de 4 nodos unidos por una red de alta velocidad. En cada nodo del cluster se ejecuta un OS local, el cual puede contener módulos embebidos requeridos por el DOS.

Generalmente sobre los OS locales, se ejecuta una capa de software intermedio (Middleware) que es responsable de la coordinación de acciones del OS en todos los nodos, y de presentar a las aplicaciones una vista de un único OS (SSI). El SSI-DOS suministra las APIs y Llamadas al Sistema requeridas tanto por aplicaciones como por bibliotecas.

Una aplicación puede ejecutar en un único nodo, tal como la Aplicación-A o la Aplicación-C, o puede ejecutar en forma distribuida tal como la Aplicación-B. Los mismos recursos abstractos existen en todos los nodos del cluster. Esta propiedad, hace que la programación sea más sencilla porque los procesos pueden compartir archivos, sockets, semáforos, colas de mensajes, señales, etc., sin realizar ninguna consideración respecto a la localización de esos objetos. Por ejemplo, un proceso ejecutando en un nodo puede forzar la terminación de otro proceso ejecutando en otro nodo diferente, simplemente utilizando el comando *kill* o la llamada al sistema *kill* mencionando el PID de la víctima.

A este tipo de OS se le denomina SSI (con Imagen de Sistema Unico) porque el procesamiento distribuido se oculta a las aplicaciones y usuarios. A esta propiedad, a la que se denomina genéricamente *transparencia*, se la puede clasificar según Tabla II.

Los DOS tienen las mismas ventajas en lo que refiere a programación, despliegue y mantenimiento de aplicaciones que los sistemas centralizados y el rendimiento, escalabilidad y disponibilidad de los Sistemas Distribuidos. Aun así, los DOS no han tenido demasiado éxito en ser adoptados. Posiblemente esto se deba a que carecían del soporte y actualizaciones necesarias. Con la creciente aparición de nuevo hardware y nuevas facilidades de los OS más utilizados, los DOS no podían actualizarse con la dinámica requerida.

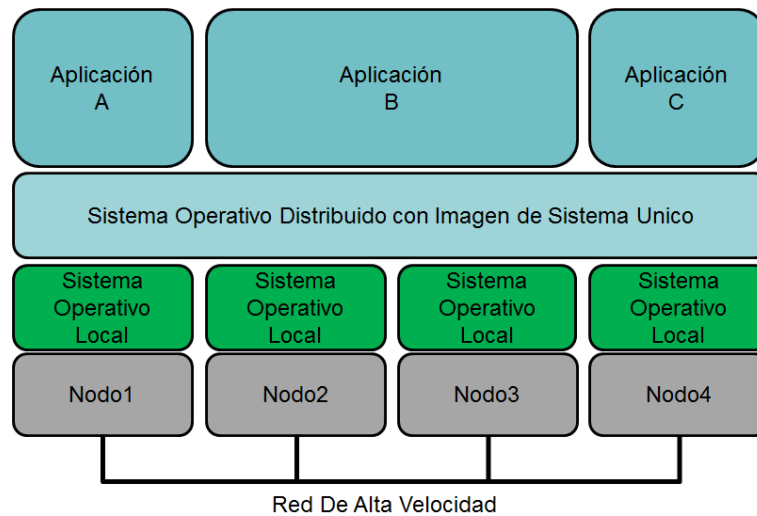


Figura 12. Sistema Operativo Distribuido con Imagen de Sistem Unico.

En [3] los autores se hacen la siguiente pregunta: “*Son las tecnologías de virtualización la causa de la falta de investigación acerca de Imagen de Sistema Unico (SSI)?*”. Los autores de [3] plantean diferentes escenarios donde se las muestra como tecnologías complementarias, pero particularmente, un escenario similar al modelo planteado por esta tesis es descartado por infactibilidad. El modelo de arquitectura propuesto en esta tesis y el prototipo de DVS demuestran que una variante de esa modalidad permite la integración entre las tecnologías de virtualización y la de los SSI-DOS. Esa modalidad descartada en [3] es absolutamente factible, demostrando que ambas tecnologías son complementarias.

TABLA II. TIPOS DE TRANSPARENCIA. (OBTENIDO DE [62])

•Transparencia	•Descripción
•Acceso	•Oculta las diferencias en cuanto a la representación de los datos y la forma en que un recurso es accedido
•Ubicación	•Oculta donde se encuentra un recurso
•Migración	•Oculta si un recurso se ha movido
•Reubicación	•Oculta si un recurso se ha movido mientras se lo está utilizando
•Replicación	•Oculta el número de copias del recurso o de que copia obtiene los datos.
•Concurrencia	•Oculta que el recurso está siendo accedido por múltiples usuarios.
•Fallas	•Oculta los fallos y la recuperación de un recurso
•Persistencia	•Oculta si el software está en memoria o disco

La modalidad actual de desarrollo y despliegue de aplicaciones en la Nube es bastante primitiva y no ofrece una visión integrada de aplicaciones y recursos. En [9], los autores resaltan esta

falencia del modelo actual de desarrollo e insisten en proponer un SSI-DOS. A la fecha, esa propuesta no ha sido adoptada por la comunidad de desarrollo ni por los proveedores de Servicios en la Nube. Esta tesis propone también una variante en la forma diferente de desarrollar un SSI-DOS que lo hace adecuado para integrarse con las tecnologías de virtualización y brindar Servicios en la Nube. Algunos ejemplos de DOS históricos son: V-System [63], Amoeba [64], Sprite [65], Chorus [66] y Locus [67]. Los DOS más actuales son: MOSIX [68], OpenSSI [69], Kerrighed [70], XtremOS [71], GNU Mach [72], y FOS [73].

Generalmente, los DOS están constituidos por programas que suelen estar fuertemente acoplados. Aun así, se podrían separar funcionalmente sus componentes en al menos dos capas, como se muestra en la Fig. 13.

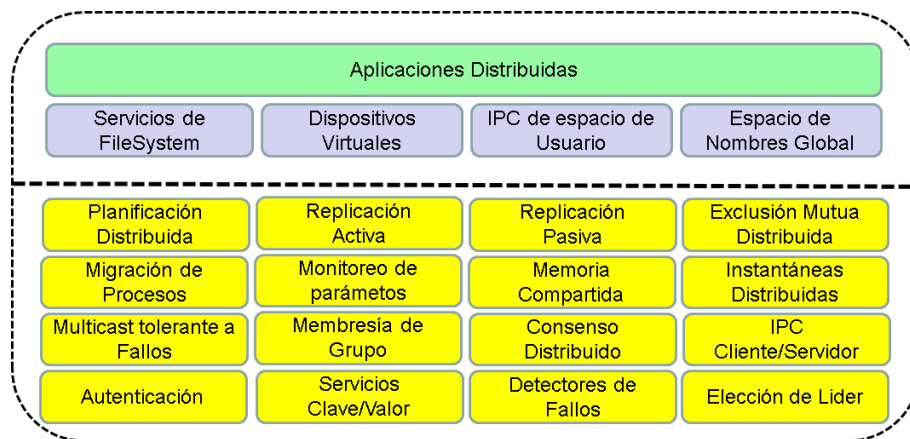


Figura 13. Componentes de un Sistema Operativo Distribuido.

La capa inferior provee servicios comunes para ser utilizados por otros servicios de la capa superior o de la misma capa. La capa superior ofrece los recursos abstractos a las aplicaciones, las Llamadas al Sistema, los Sistemas de Archivos, los espacios de nombres, etc.

Esta separación de responsabilidades y funciones es importante para el modelo de arquitectura de DVS propuesto el cual desempolva varios tópicos ya resueltos por los DOSs.

2.2.2. Unikernel

Generalmente se asocia a cada VM un OS-Guest quien ofrece a las aplicaciones servicios de archivos, IPC y red entre otros. Las aplicaciones se ejecutan por sobre el OS-Guest utilizando los servicios de éste. Varios años atrás, se propuso otra forma de procesamiento de aplicaciones mediante exokernel [74]. En ella las aplicaciones, los servicios de un Sistema Operativo de Biblioteca (Library Operating System) y los Gestores de Dispositivos se compilan en un único programa binario que se

ejecuta utilizando servicios del exokernel. Pueden ejecutarse múltiples aplicaciones sobre un mismo exokernel el cual es responsable de la gestión de recursos (CPU, memoria, E/S, etc.). Una de los mayores inconvenientes que presenta ésta tecnología es las incompatibilidades del Library-OS con la diversidad que presenta el hardware. De todos modos, este aspecto está parcialmente resuelto porque gran parte de las bibliotecas utilizadas para construir el exokernel son las mismas que para construir a un OS ordinario o sus aplicaciones.

En los unikernels no existe el concepto de Llamada al Sistema o espacio de usuario o modo usuario, todos los servicios del sistema están disponibles a la aplicación directamente invocando a una función y todo ejecuta con niveles de privilegios de kernel o supervisor (Fig. 14).

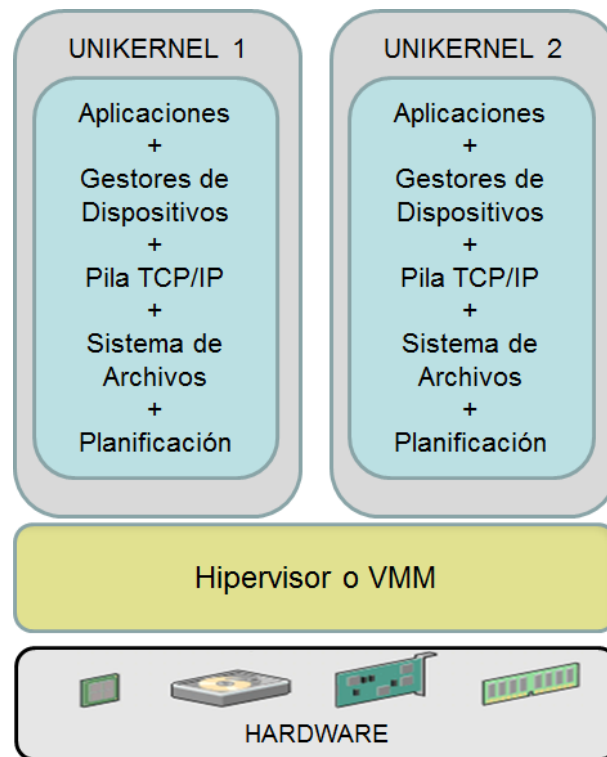


Figura 14. Unikernels ejecutando sobre Hipervisor.

Un unikernel [75] es equivalente a un exokernel, pero que no opera directamente sobre el hardware, sino que utiliza los servicios de un VMM (a modo paravirtualizado) logrando así una mayor portabilidad. De todos modos, en la actualidad existen varios unikernels que pueden ejecutar sobre el hardware directamente (como un exokernel), sobre un VMM (como virtualización de hardware o Paravirtualización) o sobre un OS con interface POSIX (como virtualización de OS). Esta forma de ejecutar aplicaciones contenidas dentro de una VM es muy eficiente en cuanto a utilización de recursos, seguridad y gran escalabilidad.

En definitiva, un unikernel es un kernel monolítico minimalista, que se construye a medida para la aplicación que lo utilizará, la que se integra en el propio unikernel junto al resto de sus componentes requeridos. Se dice que es minimalista porque solo se incluyen en su código los componentes requeridos para ejecutar tanto los componentes de servicios (Gestores de Dispositivos, Sistemas de Archivo, pila de protocolos de red, etc.) como las aplicaciones. Si bien esto reduce la superficie de ataque ante amenazas de seguridad, también se debe considerar que el unikernel se ejecuta con privilegios de kernel (incluidas las aplicaciones), por lo que un error o vulnerabilidad en una aplicación puede afectar al unikernel por completo y si éste se comunica con otros unikernels (lo que es habitual en la Nube) puede inducir errores, defectos e inconsistencia en ellos (contaminación).

Actualmente los productos que son referentes en este tipo de arquitectura son Nemesis, Singularity, ClickOS, Clive, Rumprun, MirageOS, entre otros.

2.2.3. Sistemas Operativos Multiservidor o de Microkernel

Desde sus orígenes hasta la actualidad varios OSs fueron y son implementados con un kernel monolítico. Esto significa que la mayoría de las funciones conforman un único programa (el kernel), el cual gestiona CPUs, memoria, procesos, Sistemas de Archivos, Gestores de Dispositivos, protocolos de red, etc. Esta arquitectura define las siguientes características:

- Todos los componentes del kernel (código de rutina de servicios de interrupciones, Gestores de Dispositivos, Sistemas de Archivos, gestión de memoria, etc.) comparten el mismo espacio de direcciones, lo que significa que comparten variables globales y funciones, por lo que todos los componentes tienen acceso total y completo a ellas. Esta característica le otorga un alto rendimiento porque desde un componente dado se puede acceder a cualquier variable o función del kernel simplemente mencionándola o invocándola.
- Las Llamadas al Sistema se implementan utilizando trampas o compuertas de la CPU.
- Todo el código del kernel se ejecuta en modo kernel o modo supervisor, lo que significa que éste dispone de todos los privilegios para acceder a cualquier área de memoria o controlar cualquier dispositivo, como así también para ejecutar cualquier instrucción de la ISA.
- Resulta más complejo mantener un código enorme, y que al modificar un componente no se afecte al funcionamiento de otro componente.
- Se puede integrar código adicional al kernel a través de módulos cargables.

-
- Al compartir el mismo espacio de direcciones entre módulos diferentes se pone en riesgo la seguridad y protección. Esta característica que es una ventaja en cuanto al rendimiento se transforma en una debilidad en seguridad. Por ejemplo, si un Gestor de Dispositivo tiene un defecto de programación, éste podría afectar a cualquier otro componente del sistema tal como el planificador, el Sistema de Archivos o los protocolos de red. La debilidad en la protección no solo comprende a cuestiones relacionadas con la seguridad frente a ataques por parte de personas maliciosas, sino también a fallos en funcionamiento y de rendimiento. Es decir, en un OS monolítico el aislamiento entre los componentes del kernel es débil, con una gran TCB, o lo que es equivalente a presentar una superficie de ataque o de fallos cuyo tamaño comprende a todos los componentes del kernel. Cualquier error o defecto en uno de los componentes del kernel puede causar una inhabilitación o caída del OS completo con todos sus procesos y aplicaciones.

Los OS de microkernel fueron desarrollados en la década del 80 como respuesta a los problemas de los OS monolíticos. La estrategia utilizada fue minimizar el kernel e implementar servicios como servidores en modo usuario. En el microkernel se implementan componentes responsables de la gestión básica de memoria, de los mecanismos de IPC, de gestionar interrupciones, fallos y excepciones, y de realizar una planificación de corto plazo de los procesos. Como los servidores tienen sus propios espacios de direcciones, sus objetos (variables y funciones) se encuentran protegidas del acceso por parte de otros módulos. Los microkernels tienen los siguientes beneficios:

- El OS se hace más flexible y extensible.
- Una interface de microkernel fuerza a una estructura más modular del OS.
- Un kernel pequeño es más fácil de mantener y es menos propenso a errores, lo que resulta en un OS más confiable y robusto.
- Se reduce la TCB.

La principal desventaja de los OS de microkernel es que una simple Llamada al Sistema, tal como *getpid()*, puede requerir al menos un par de transferencias de mensajes. Una Llamada al Sistema más compleja, tal como *read()* o *write()*, puede requerir una serie de transferencias de mensajes entre el programa de aplicación y uno o más servidores, lo que impacta negativamente en el rendimiento. Cada transferencia de mensaje implica a su vez, varios cambios de contextos, inserción y remoción de

procesos de las colas de procesos en estado *Listos para Ejecución*, múltiples invocaciones al planificador, copias de bloques de datos y mensajes entre diferentes espacios de direcciones, etc.

En la Fig. 15, se muestra un modelo de OS multiservidor en el que las funcionalidades de un OS son factorizadas en varios servidores independientes que se ejecutan en modo usuario. Al ejecutar en modo usuario, un fallo en uno de los componentes no necesariamente provocaría un fallo en el sistema completo. Solo el microkernel ejecuta en modo kernel, adquiriendo así todos los privilegios sobre la plataforma.

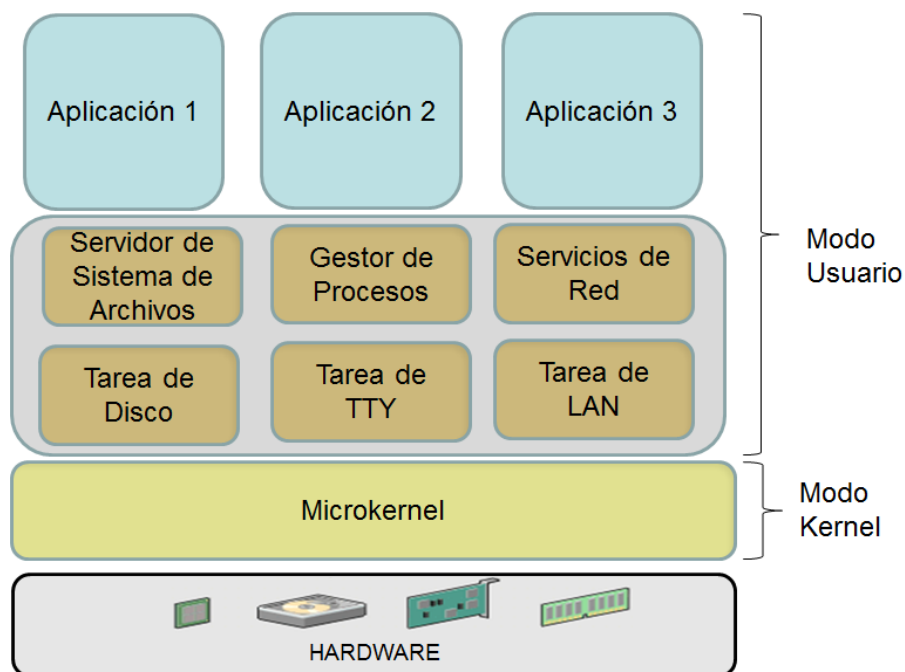


Figura 15. Sistema Operativo Multiservidor o de Microkernel.

Cada uno de los servidores ejecuta en modo usuario en su propio espacio de direcciones y tiene asignada su propia cuota de tiempo de CPU como cualquier otro proceso, incluso los procesos de aplicaciones de usuario. Como los procesos necesitan comunicarse, el microkernel provee mecanismo de IPC. Los datos y funciones de un servidor no son accesibles desde otro servidor o proceso, la información debe transferirse utilizando IPC. Esta característica de aislamiento asemeja sus propiedades a la de las tecnologías de virtualización. Existen numerosos artículos de investigación que analizan y comparan características de hipervisores vs. OS de microkernel [27, 28, 29].

Las ventajas en el rendimiento de los OS monolíticos frente a los OS de microkernel recae en aprovechar las facilidades que ofrece la memoria compartida y la coherencia de cache de los SMP que hoy son comunes tanto en servidores, computadores de escritorio o dispositivos móviles. En esos sistemas SMP, la consistencia del cache es mantenida por el hardware, descargando al software de esa

tarea. Los objetos del kernel tales como las estructuras de datos globales y las funciones residen en memoria compartida. El hecho de compartir variables y funciones habilita a que cualquiera de sus componentes puede acceder al estado de otro componente simplemente referenciando una variable compartida o invocando una función lo que mejora el rendimiento.

Actualmente, hay disponibles chips con varias decenas de núcleos, pero se espera que en menos de una década la cantidad total de núcleos sea de varios cientos (*many-cores*). A medida que se incrementa el número de núcleos, se dificulta mantener la consistencia o coherencia de cache dado que resulta en un cuello de botella para el bus. De este razonamiento es que surgen los OS multikernel [9, 162], que se presentan como un nuevo modelo de programación que trata a una máquina *many-core* como un sistema distribuido. No se asume la existencia de memoria compartida, pero sí que se dispone de mecanismos de IPC para transferencia de mensajes. Con tantos núcleos, se pueden asignar un proceso de sistema (servidor de sistema de archivos, gestor de red, Gestor de Dispositivo de disco, etc.) a cada núcleo y no realizar cambios de contexto.

Los OS de microkernel son muy versátiles y flexibles, como sólo dependen de los mecanismos de IPC, pueden fácilmente adaptarse a los sistemas *many-core* tal como se demuestra en [76], o convertirse en SSI-DOSs [77, 78], o transformarse en sistemas de tiempo real como Minix4RT [79] o en hipervisores tal como Mhyper [40]. Existen varios OS de microkernel, pero los más utilizados en la actualidad son: L4 [80], QNX Neutrino [81], HelenOS [82] y Minix [41].

2.2.4. Comunicaciones de Grupo

Un sistema de virtualización cuyo objetivo es brindar servicios de tipo IaaS con calidad de proveedor debe contemplar en su diseño mecanismos que le permitan incrementar la disponibilidad de los servicios. Como un sistema distribuido que es, el DVS propuesto debe soportar las características dinámicas de los clusters donde permanentemente se incorporan y se remueven nodos. En un Datacenter ocurren fallos en los computadores, en los procesos, en la red, y de operación; los cuales deben estar contemplados en el diseño del sistema. El tratamiento de fallos es un tema complejo y hay numerosas investigaciones al respecto [83, 84, 85]. Existen diversas alternativas de software tales como el consenso distribuido [86, 87] y la difusión (multicast) tolerante a fallos [88] que pueden utilizarse para resolver el problema de la replicación de servicios.

Un Sistema de Comunicaciones de Grupo (GCS: Group Communication System) permite el intercambio de mensajes mediante mecanismos de difusión tolerante a fallos entre los procesos miembros de un grupo. Generalmente están disponibles diferentes formas de entrega en los mensajes

tal como FIFO, Causal, Orden Total, etc. de tal forma que le permitan al programador seleccionar aquella modalidad que se adecua mejor a su aplicación. Además, como los cluster son dinámicos, un GCS debe incluir todos los aspectos relacionados con la membresía de nodos que conforman el grupo. Esto quiere decir, que al grupo pueden incorporarse nuevos nodos o removerse administrativamente. También pueden ocurrir fallos de procesos que requiere que los restantes miembros tomen en consideración ese hecho y eventualmente tomen las acciones correspondientes.

Birman, señala en [83]: “*se debería usar un GCS por consideraciones de estandarización, complejidad, y rendimiento*”. A estas consideraciones se le deberían adicionar aspectos tales como el soporte en cambios de conformación del cluster (membresía) y la detección de fallos. Además de simplificar el desarrollo de una aplicación tolerante a fallos, el uso de un GCS permite desacoplar la aplicación distribuida de los mecanismos de comunicaciones de grupo y de la implementación de detectores de fallos. Adicionalmente, un GCS brinda las herramientas necesarias para resolver otros problemas típicos que se presentan en los sistemas distribuidos como es el caso de la replicación de servicios tales como Servidores de Archivos, Servidores de Almacenamiento, Servidores de Clave-Valor, colas de mensajes distribuidas, registros distribuidos, por enumerar algunos.

2.2.5. Migración de Procesos, Máquinas Virtuales y Contenedores

La migración de procesos es otra temática ampliamente estudiada en los sistemas distribuidos [89, 90, 91]. Esta permite transferir procesos (y su carga de procesamiento y uso recursos que representan) desde un nodo a otro de un cluster.

Las técnicas y razones para la migración de Procesos, VMs y Contenedores son muy similares, por esta razón en esta sección solo se hará referencia a procesos para facilitar la lectura, mencionando a VMs o Contenedores cuando se haga una mención particular para estos.

La migración de procesos puede ser:

-Estática: Se dice que es migración estática cuando se decide que la ejecución de un proceso se realizará en otro nodo (destino) diferente al nodo (origen) donde se está ejecutando el proceso que lanza la ejecución del nuevo proceso. Esta modalidad es la más sencilla de implementar dado que no hay movimiento de proceso desde un nodo a otro y el nuevo proceso se ejecutará desde su inicio en el nodo destino.

-Dinámica: Se dice que es migración dinámica cuando la misma procede durante la ejecución del proceso. Esta modalidad presenta varias cuestiones que deben resolverse, algunas de muy alta complejidad.

Las razones para la migración dinámica suelen ser:

-Distribución Dinámica de Carga: cuando un nodo del cluster supera un límite de carga, podría ser conveniente transferir su ejecución a otro nodo que se encuentre descargado.

-Ahorro Energético: cuando el conjunto de nodos de un cluster está operando por debajo de un límite dado, el administrador puede considerar conveniente concentrar o consolidar una cierta cantidad de servicios (procesos) en una menor cantidad de nodos, de tal forma de proceder a la desconexión de aquellos que fueron descargados de procesos.

-Mantenimiento: cuando el hardware o el software de algún nodo requiere mantenimiento, los procesos que en él ejecutan podrían desplazarse hacia otros nodos de tal forma de descargar completamente al nodo en cuestión y proceder a su reparación o actualización.

Un sistema de migración dinámica de procesos debe cumplir con ciertos requerimientos [92] que lo hacen adecuado para la tarea, estos son:

-Transparencia: Cuando se migra un proceso, éste no debe percibir que ha cambiado, por lo que debe haber transparencia en lo que *acceso a los recursos* y Llamadas al Sistema se refiere. Por ejemplo, si el proceso realizó un *open()* exitoso de un archivo y luego es migrado, ese archivo debe aparecer como abierto después que el proceso ha migrado. De igual forma, si un proceso se comunica con otro usando una tubería o *pipe*, después de migrar deberá poder acceder a esa misma tubería.

-Mínima interferencia: Se deberá minimizar el tiempo durante el cual el proceso no está ejecutando en el nodo origen ni en el nodo destino.

-Dependencias residuales mínimas: La migración de un proceso no debe dejar ningún recurso asignado a éste en el nodo origen. De dejar dependencias residuales en el nodo origen, le impondrá a este una carga adicional, aun cuando el proceso ya no se encuentre ejecutando en ese nodo. Por otro lado, una falla en el nodo origen

provocaría una falla en el proceso, aunque éste se encuentre ejecutando en el nodo destino, reduciendo la disponibilidad del proceso. Por otro lado, si el proceso es sometido a varias migraciones por diferentes nodos, dejaría una cadena de dependencias residuales que disminuirían más aún su disponibilidad.

-Eficiencia: Para realizar la migración dinámica de procesos deben transferirse su estado y su espacio de direcciones de memoria desde el nodo origen al nodo destino. El tiempo de transferencia representa la mayor fuente de ineficiencia. Se debe evaluar en la planificación de la migración de un proceso la conveniencia de la misma considerando estos tiempos y el consumo de recursos (de procesamiento y de red) que demanda la misma.

-Robustez: La falla en un nodo distinto del que está ejecutando un proceso no debe afectar la ejecución de ese proceso ni su accesibilidad a los recursos que tiene asignados (de hardware o de software).

-Comunicación entre procesos relacionados: Generalmente, existen grupos de procesos que se comunican entre sí para llevar a cabo tareas coordinadas. Estos procesos suelen usar mecanismos de comunicación (IPC) provistos por el OS. Generalmente, el OS implementa estos mecanismos de IPC utilizando su propia memoria para contener las estructuras de datos que componen el recurso abstracto. Si de un conjunto de procesos relacionados entre sí se migra alguno de ellos a otro nodo, los mecanismos de IPC del nodo origen no son los mismos que en el nodo destino, por lo que deben implementarse mecanismos de comunicación que utilicen protocolos de red. Esto hace que aumente la latencia de las comunicaciones, el uso de CPU y el factor de utilización de la red. Por esta razón, suele ser conveniente migrar de un nodo a otro del cluster, no a un proceso solitario sino a todo un conjunto de procesos relacionados. Esta condición se cumple básicamente al migrar VMs y Contenedores.

Prácticamente, todos los sistemas de virtualización de hardware que están preparados para brindar servicios con calidad de proveedor, disponen de mecanismos de migración de VMs. Una VM para un hipervisor es equivalente a un proceso para un OS, por esta razón se utilizan las mismas técnicas de migración que para estos últimos. Migrar VMs [93] suele ser menos complejo que migrar procesos porque las dependencias entre las VMs y el hipervisor suelen ser menores.

En los sistemas de virtualización basados en OS, también se han desarrollado técnicas para la migración de Contenedores [91, 94]. La migración de Contenedores (con sus procesos contenidos)

también suele ser más sencilla que la migración de procesos en solitario, porque la tecnología de virtualización establece una capa de aislamiento entre los procesos dentro del Contenedor y el OS. Como los espacios de nombre (*namespaces*) que comprenden a UIDs, IPC, PIDs, puntos de montajes de sistemas de archivos, etc. forman parte del Contenedor, se deben volver a mapear estos nombres a los recursos locales del OS destino. Por esta razón, los Contenedores tiene mejores características de aislamiento que un proceso individual, por lo que su migración de un nodo a otro presenta menos inconvenientes.

Generalmente en los SSI-DOS, cada proceso ejecuta dentro de su propio entorno aislado conformando lo que se conoce como un vPROC (virtual PROCess) [95]. De esta forma es posible implementar PIDs (y otros recursos) independientes del nodo donde se ejecute el proceso, lo que permite utilizar herramientas de gestión para todos los procesos del cluster tales como *ps*, *ptrace*, *kill*, etc. De igual forma, los vPROCs facilitan la migración de los procesos entre nodos del cluster.

Como se mencionó más arriba la migración de procesos, VMs o Contenedores comprende tanto a su estado como a su espacio de direcciones de memoria. Este último es el que representa el mayor volumen de datos a transferir entre nodos y es el factor principal a considerar cuando se quiere reducir la interferencia. Existen varias técnicas para transferir el espacio de direcciones desde un nodo a otro tratando de reducir el tiempo de interferencia, que depende del tamaño del espacio de direcciones. A los efectos de simplificar las descripciones, referiremos a los objetos a migrar como procesos:

❑ **Congelamiento Total**

En esta técnica (Fig. 16-A), el proceso se detiene por completo en el nodo origen, se realiza la transferencia y luego continúa su ejecución en el nodo destino. Es una técnica simple, solo transfiere por la red el volumen del espacio de direcciones, pero es la técnica que presenta el mayor tiempo de interferencia.

❑ **Pretransferencia**

Una vez que se ha decidido migrar un proceso, se comienza con la transferencia de las páginas que componen su espacio de direcciones (Fig. 16-B). En el nodo origen, cada página transferida se protege en modo “*solo lectura*”. Una vez finalizada la Pretransferencia, algunas de las páginas que componen el espacio de direcciones en el nodo origen pudieron haber sido modificadas dado que la ejecución no se detuvo, pero las mismas son detectadas por el OS (o VMM) origen al protegerlas contra escritura. Luego, se detiene al proceso y se comienza a transferir solo las páginas modificadas.

De esta forma se reduce el tiempo de interferencia dado que éste es proporcional a la cantidad de modificadas, pero el volumen de datos total transferido por la red es mayor o igual al espacio de direcciones del proceso.

□ *Transferencia por Referencia*

En esta modalidad (Fig. 16-C), se detiene la ejecución del proceso en el nodo origen y sólo se transfiere su estado desde el nodo origen al destino. Cuando en el nodo destino se comienza a ejecutar, el espacio de direcciones está vacío, por lo cual la CPU del nodo destino produce un *fallo de página* dado que la página requerida no se encuentra en memoria. El OS (o VMM) del nodo destino solicita al OS (o VMM) del nodo origen la transferencia de esa página en particular. A medida que el proceso va ejecutando, se van haciendo referencia a otras páginas no transferidas, las que van produciendo sucesivos fallos de página. El tiempo de interferencia solo comprende el tiempo de transferir el estado, lo que lo hace más imperceptible. Pero el tiempo total para transferir todo el espacio de direcciones se divide de acuerdo al número de páginas que lo conforman por lo que la ejecución del proceso presentará eventuales demoras intermitentes. Esta técnica presenta un inconveniente al dejar dependencias residuales (páginas no referenciadas) en el nodo origen, disminuyendo así la disponibilidad del proceso que ahora se hace dependiente del nodo origen y del nodo destino mientras se realizan las transferencias. Una variante para reducir esta dependencia es transferir las páginas requeridas del nodo origen al destino, y en paralelo transferir otras páginas no requeridas hasta completar el total del espacio de direcciones.

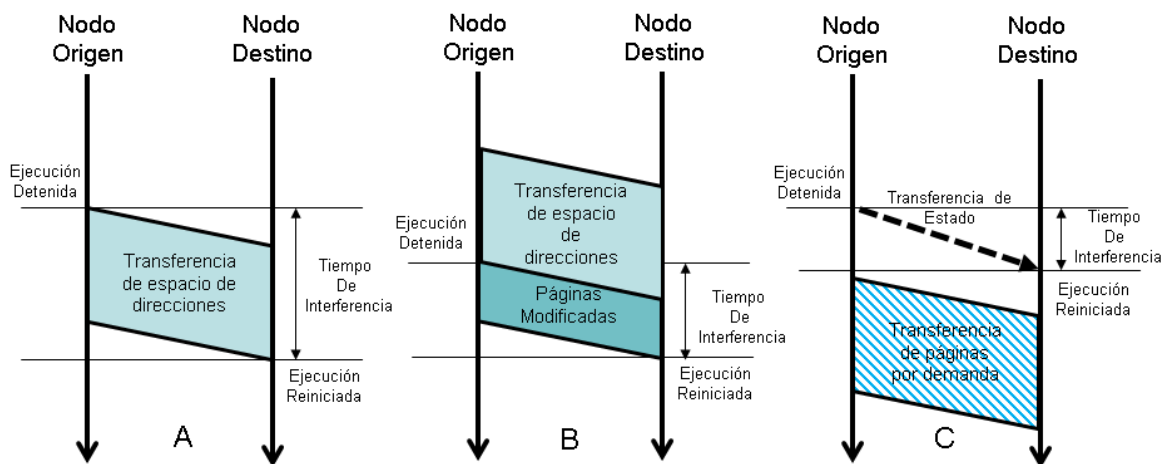


Figura 16. Modos de transferencias del espacio de direcciones de procesos, VMs o Contenedores.

Otro de los aspectos fundamentales que debe resolverse cuando se migran procesos, VMs o Contenedores refiere a los mecanismos a utilizar para mantener las comunicaciones preestablecidas cuando el proceso está migrando o ya ha migrado [96]. Al migrar de nodo se modifican, por ejemplo,

las direcciones MAC de las interfaces de red, las sesiones TCP, etc. La solución representa todo un desafío para superar las fuertes dependencias entre el proceso migrado y el nodo origen. En el caso del prototipo del DVS, el mecanismo de IPC provisto contempla la migración transparente de procesos, es decir las comunicaciones entre procesos se mantienen aun cuando uno de ellos haya migrado.

Conforme se pretende que el modelo de arquitectura de DVS propuesto permita construir una tecnología apta para brindar Servicios en la Nube, se requiere disponer de la facilidad de migrar procesos y/o Contenedores. Para la migración de procesos en Linux existen herramientas tales como CRIU (Checkpoint and Restore In Userspace) [97], para la migración de Contenedores Linux se dispone de CMT (Containers Migration Tool) [98] y LXD que incluye a CRIU.

2.2.6. Redundancia y Replicación

Una tecnología que pretende ser adoptada para brindar Servicios en la Nube debe disponer de mecanismos que le permitan soportar fallos y ocultarlos para ofrecer servicios con alta disponibilidad que puedan ser utilizados para aplicaciones críticas.

Los mecanismos que habitualmente se utilizan para incrementar la disponibilidad son la redundancia y la replicación. Se pueden aplicar a elementos estáticos como lo hacen las tecnologías de almacenamiento de tipo RAID (Redundant Array of Independent Disk), a procesos como lo hacen los mecanismos de replicación de servicios, o a comunicaciones utilizando enlaces redundantes o protocolos de inundación. Los dos métodos que han marcado el desarrollo de aplicaciones redundantes se describen a continuación.

❑ Replicación Pasiva o Primario-Respaldo [99]

Un proceso elegido como Primario replica los comandos de las actualizaciones a los procesos Respaldo utilizando difusión de mensajes (multicast). Esta difusión debe ser tolerante a fallos y debe mantener idéntico orden (orden Total o Atómico) para todas las réplicas. Dado que el Primario es el que impone el orden a las peticiones de actualizaciones que se envían a las otras réplicas, solo hace falta que la difusión cumpla con ordenamiento FIFO. Cuando una réplica detecta que el Primario ha fallado, se elige un nuevo Primario entre las réplicas de tipo Respaldo, y el servicio continúa brindándose. Los clientes ahora dirigen sus peticiones al nuevo Primario o éste adquiere idéntica personalidad que el Primario fallido (dirección IP, puerto TCP, MAC address, etc.). Las réplicas de

tipo Respaldo no prestan servicio alguno a los clientes (son pasivas), dado que los clientes solo se comunican con el Primario.

❑ **Replicación Activa o de Máquina de Estado [100]**

En este caso todas las réplicas son activas y se comunican entre sí mediante un protocolo de difusión tolerante a fallos. El orden de la difusión debe ser Total o Atómico, de tal forma que todas las réplicas procesen las peticiones en idéntico orden, por lo que el sistema distribuido donde se ejecuta este protocolo de difusión no puede ser de tipo Asincrónico [101]. Como todas las réplicas ejecutan el mismo código, todas parten con el mismo conjunto de datos y todas ejecutan las mismas peticiones determinísticas en el mismo orden. Una nueva petición de actualización provocará idénticos resultados y, por ende, idéntico estado resultante en todas las réplicas. La desventaja en el rendimiento que implica la difusión con Orden Total se puede compensar con la distribución de carga que se puede realizar sobre las distintas réplicas, dado que los clientes pueden someter sus peticiones a cualquiera de ellas. Si las peticiones son de lectura o no alteran el estado, entonces la réplica la resuelve en forma local sin comunicarlo al resto del grupo.

El modelo de DVS propuesto debe facilitar la implementación de procesos replicados por cualquiera de estos dos métodos. Por ejemplo, en el caso de Replicación Primario-Respaldo hace transparente a las modificaciones las comunicaciones entre los clientes y un nuevo Primario después de fallar el predecesor. Para el caso de Replicación por Máquina de Estados, soporta la distribución de cargas que imponen los clientes entre las distintas réplicas.

3. MODELO DE ARQUITECTURA PROPUESTO

El modelo de arquitectura de DVS propuesto en esta tesis [102, 103], ante todo resuelve las limitaciones en cuanto a capacidad de cómputo, memoria y almacenamiento que tienen las VMs y los Contenedores. Conforme su característica modular admite la incorporación de nuevos servicios que facilitan el desarrollo, despliegue y mantenimiento de Servicios en la Nube. Un DVS integra conceptos y enfoques de varias tecnologías a saber:

- 1) *Sistemas Operativos Multiservidor o de Microkernel [41, 80, 81]*: A diferencia de los OS monolíticos, los OS multiservidor basados en microkernel descomponen al sistema en múltiples capas y múltiples procesos aislados que ejecutan en modo usuario. Estos procesos se comunican entre sí mediante mecanismos de IPC que provee el microkernel. El microkernel brinda servicios básicos a las capas superiores y es el único componente que ejecuta en modo kernel o modo supervisor.
- 2) *Sistemas Operativos Distribuidos con Imagen de Sistema Única [69, 70, 73, 104]*: Estos OSs distribuyen la ejecución de procesos en diferentes nodos de un cluster ofreciendo al usuario una imagen única de sistema. Algunos de estos sistemas permiten la migración de procesos para distribución y/o balanceo de cargas.
- 3) *Virtualización de Sistema Operativo - Sistemas Operativos Virtuales [17, 51, 52, 55]*: Un OS ofrece servicios y abstracciones a las aplicaciones a través de su interface superior, y se comunica con el hardware de la plataforma a través de su interface inferior. Al igual que un OS, un VOS ofrece servicios y abstracciones a las aplicaciones a través de su interface superior, pero utiliza los servicios de un OS subyacente a través de su interface inferior. El funcionamiento es similar al de un OS ejecutando sobre un sistema paravirtualizado, excepto que utiliza Llamadas al Sistema (*syscalls*) en vez de Llamadas al Hipervisor (*hypercalls*).

-
- 4) *Virtualización basada en Sistema Operativo con Contenedores [20, 22]*: El kernel del OS-Host particiona sus recursos en instancias aisladas en espacio de usuario. El sistema de virtualización construye un Contenedor que confina a un conjunto de aplicaciones y a su entorno de ejecución. Esta tecnología se basa en la virtualización de Llamadas al Sistema. Si bien todas estas instancias se ejecutan en sus propios entornos, todas comparten el mismo OS.
- 5) *Sistema de Comunicaciones Grupales (GCS) [83]*: Como el DVS es un sistema distribuido, se requiere de mecanismos de comunicación para sus diferentes componentes, y una alternativa es utilizar un GCS. Dentro de las posibilidades que ofrecen los sistemas distribuidos se encuentra la replicación; al utilizar GCS ésta debe ofrecer diferentes opciones de entrega de mensajes tales como FIFO, Causal, Orden Total, etc. Por otro lado, como los sistemas distribuidos que ejecutan en un cluster, la falla de uno de los nodos del mismo puede provocar una falla completa del sistema. Para reducir esta posibilidad, el sistema de GCS debe ofrecer a las aplicaciones mecanismos de tolerancia a fallos.

La descomposición en múltiples tareas y servidores comunicándose mediante mecanismos de IPC como proponen los OS de microkernel brinda el aislamiento requerido por un sistema de virtualización y permite la distribución de los mismos en diferentes computadores siempre y cuando se puedan comunicar mediante IPC y GCS.

Actualmente los servicios de infraestructura proveen recursos tales como CPUs, memoria, disco y red, pero el usuario carece de una visión integrada tal como la que ofrece un OS multikernel [9, 162] debiendo gestionar explícitamente sus recursos en la Nube. El modelo de arquitectura de DVS propone una gestión integrada de recursos y transparencia para las aplicaciones. Estas características, simplifican la programación, el despliegue y mantenimiento de aplicaciones en la Nube.

Es probable que en la próxima década sean comunes los chips con cientos de procesadores (many-cores) [105], pero también es probable que no utilice el paradigma de memoria compartida, al menos como lo utilizan los procesadores actuales, debido a la contención del bus por actualizaciones o invalidaciones de cache. Intel ha liberado para proyectos investigación su *Nube en Computador de un único chip* [106] (SCC: Single-chip Cloud Computer). Básicamente, el SCC es una cluster en un chip constituido por 24 nodos de 2 núcleos, unidos por una red conmutada de 256 GB/s y con mecanismo de pasaje de mensajes implementado en Hardware. Los OS que utilizan IPC se adaptan

perfectamente a este tipo de arquitectura de hardware [76, 73]. El paradigma de desarrollo basado en transferencia de mensajes, aplica tanto para la comunicación entre procesos en el mismo chip como para procesos en diferentes nodos de un cluster. Aún con la potencia de cómputo provista por estos chips, seguramente la demanda de procesamiento superará los límites impuestos por esa tecnología, por lo que los DOSs deberán extenderse a través de las redes más allá de las fronteras del chip. Esta es otra razón por la cual un sistema de virtualización debe ser distribuido.

3.1. Topología de un cluster DVS

El modelo de arquitectura de DVS cumple con los requerimientos de los proveedores de Servicios en la Nube tales como alta disponibilidad, replicación, elasticidad, balanceo de carga, gestión de recursos y migración de procesos entre otros. Su objetivo principal es aumentar el rendimiento asignando recursos de un subconjunto de nodos a cada instancia de un VOS (agregación de recursos), pero permitiendo que múltiples VOS compartan los mismos nodos (partición de recursos). Cada VOS se ejecuta en un entorno aislado denominado Contenedor Distribuido (DC), el cual puede abarcar desde uno a todos los nodos del cluster DVS. Un DC es un entorno de ejecución aislado y distribuido que ofrece recursos computacionales, virtuales y abstractos a un conjunto de procesos. En la Fig. 17, se presenta un ejemplo de topología de un cluster DVS que cuenta con cinco nodos unidos por una red de alta velocidad.

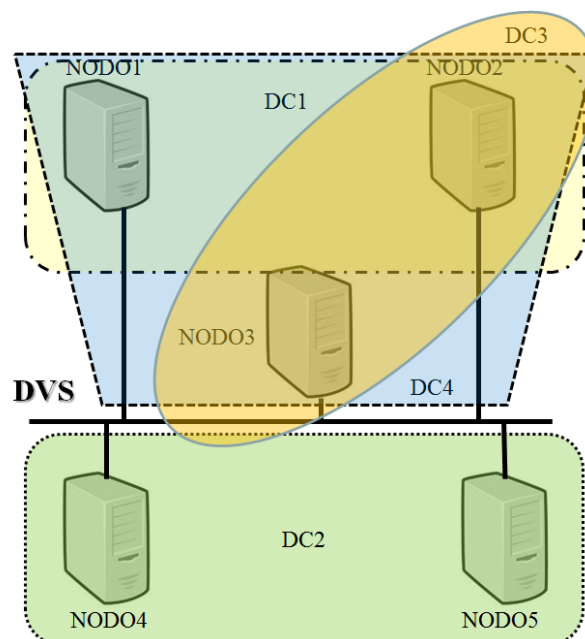


Figura 17. Ejemplo de Topología de un cluster DVS.

Sobre esos 5 nodos se definen 4 DCs asignados de la forma en que muestra la Tabla III. Por ejemplo, el NODO1 está compartido por los DC1 y DC4 (partición de recursos) en tanto que el DC3 cubre los nodos NODO2 y NODO3 (agregación de recursos).

TABLA III. EJEMPLO DE ASIGNACIÓN DE NODOS A CONTENEDORES DISTRIBUIDOS

	DC1	DC2	DC3	DC4
NODO1	X			X
NODO2	X		X	X
NODO3			X	X
NODO4		X		
NODO5		X		

Un DVS puede ejecutar un VOS con componentes distribuidos que, complementado con el aislamiento de un DC, le permite expandirse por fuera de los límites de un único computador o nodo. A través de la agregación de recursos de múltiples nodos un VOS puede obtener más potencia de cómputo y más recursos para así alcanzar mayores niveles de rendimiento, escalabilidad y elasticidad en su configuración. Como algunos de los procesos y recursos (físicos o abstractos) de un VOS pueden replicarse, también es posible obtener mayor disponibilidad y robustez.

3.2. Abstracciones

El modelo de DVS considera las siguientes abstracciones a saber:

- DVS*: Es el Sistema de Virtualización Distribuida que abarca un cluster de *Nodos*.
- Nodo*: Es un computador que conforma el cluster de virtualización y en el cual se ejecuta un OS-host que incluye al módulo del Núcleo de Virtualización Distribuida (DVK: Distributed Virtualization Kernel), sus bibliotecas y servicios asociados.
- Proxies*: Son procesos que se encargan de transferir mensajes y bloques de datos entre nodos. Utilizan APIs del DVK para obtener los mensajes y datos enviados por procesos *Locales*^{Nota1} a transferir hacia otros nodos, y para entregar mensajes y datos provenientes de otros nodos destinados a procesos *Locales*. Los *proxies* se registran

Nota1: Se dice que un objeto es Local cuando desde el punto de vista de un nodo, este objeto se encuentra definido en ese nodo. Se dice que un objeto es Remoto cuando desde el punto de vista de un nodo, este objeto se encuentra definido en otro nodo.

en el módulo local del DVK de cada nodo, especificando a que nodos remotos representa.

-*Contenedores*: Son entornos aislados de ejecución de procesos limitados a un solo nodo. Son abstracciones ofrecidas por el OS-host de cada nodo.

-*Contenedores Distribuidos*: Son conjuntos coordinados de Contenedores individuales que conforman un entorno de ejecución aislado distribuido. No se admite la transferencia de mensajes o de datos (IPC) entre procesos de diferentes *DCs* (aislamiento), pero si se admite IPC entre procesos del mismo *DC* ejecutando en diferentes nodos (transparencia de localización).

-*Procesos*: Son tareas que se ejecutan en forma local en un nodo, o en forma remota en otro nodo del cluster DVS. Una tarea del OS-host Local se convierte en un Proceso Local cuando se registra ante el DVK en un *DC* dado. Un Proceso Remoto se registra en el módulo del DVK del nodo local en un *DC* dado para un Proceso que se ejecuta en otro nodo. Cada Proceso tiene asociado un *endpoint* que lo identifica en el *DC*.

-*Endpoint*: Un *endpoint* es un número (entero) que identifica a un Proceso (o a varios procesos relacionados localizados en diferentes nodos) en todo el *DC*. Existen varios tipos de *endpoints*:

○*Endpoint Local*: En un nodo dado, un *endpoint* es Local si existe una tarea en el OS-host local que tiene asignado en forma exclusiva ese *endpoint* de tipo Local en todo el *DC*.

○*Endpoint Remoto*: En un nodo dado, un *endpoint* es Remoto si existe una tarea en el OS-host de otro nodo perteneciente al mismo *DC* que tiene asignado ese *endpoint* como de tipo Local.

○*Endpoint Respaldo*: En un nodo dado, un *endpoint* es tipo Respaldo si existe una tarea en el OS-host del nodo local que tiene asignado ese *endpoint* y otra tarea en el OS-host de otro nodo con el mismo *endpoint* pero de tipo Local perteneciente al mismo *DC*. Pueden existir múltiples *endpoints* de tipo Respaldo con el mismo identificador en diferentes nodos pertenecientes al mismo *DC*, todos ellos referenciando al nodo del único *endpoint* con ese identificador de tipo Local. El *endpoint* Respaldo tiene el mismo

comportamiento para el IPC que un *endpoint* tipo Remoto. Se usa para registrar Procesos de tipo Respaldo cuando se utiliza el esquema de replicación Primario-Respaldo [99] y para migrar un proceso de tipo Remoto desde un nodo Remoto al nodo Local. Su semántica se ejemplifica en la [Sección 3.2.2.](#)

○*Endpoint Réplica*: Un *endpoint* es tipo Réplica cuando pueden existir múltiples Procesos con el mismo *endpoint* tipo Réplica en varios nodos del mismo DC. Se utiliza para registrar Procesos que van a utilizar un esquema de replicación Activa o de Máquina de Estados [100]. Los mensajes y datos transferidos entre Procesos locales a un nodo y ese *endpoint* Réplica del mismo nodo, serán transferidos al Proceso local. También, se pueden utilizar los *endpoints* Réplica para realizar distribución de cargas. Su semántica se ejemplifica en la [Sección 3.2.2.](#)

○*Endpoint de Proxy*: El par de tareas o procesos que conforman un Proxy también tienen sus propios *endpoints* que los distinguen del resto de los *endpoints* utilizados para Procesos ordinarios. Cuando un par de tareas se registran como Proxy adquieren los privilegios para tomar los mensajes y datos del nodo local para luego poder transferirlos a un nodo remoto (o a varios). También adquieren los privilegios para entregar a los Procesos locales los mensajes y datos provenientes de otros nodos. Los *endpoint* de *proxies* no pueden mencionarse como origen ni como destino en las APIs de IPC, son de uso interno del DVK.

El *endpoint* de un proceso finalizado debe de-registrarse del DC utilizando las APIs de gestión que ofrece el DVK. Ese *endpoint* liberado podrá ser reasignado luego a otro proceso.

3.2.1. Relaciones entre las Abstracciones

Para describir al modelo de arquitectura DVS se requiere establecer las relaciones entre las abstracciones que lo conforman. En la Fig. 18 se presenta un diagrama de estas relaciones.

Un cluster *DVS* comprende un conjunto de *Nodos* el cual puede compartirse entre múltiples DCs (agregación). Cada DC está compuesto por uno o más *Contenedores* individuales que se

localizan en diferentes *Nodos*. Los *Nodos* se comunican entre sí utilizando *Proxies* que son *Procesos* que no tiene *DC* asignado y se definen a nivel del módulo local del DVK en cada *Nodo*.

En el DVK de cada *Nodo*, un *DC* tiene registrados *Procesos* de tipo Local, de tipo Respaldo y/o de tipo Réplica que se ejecutan como *Tareas* locales dentro de un *Contenedor*, pero también tiene registrados *Procesos* de tipo Remoto que son *Tareas* que se ejecutan en otros *Nodos*.

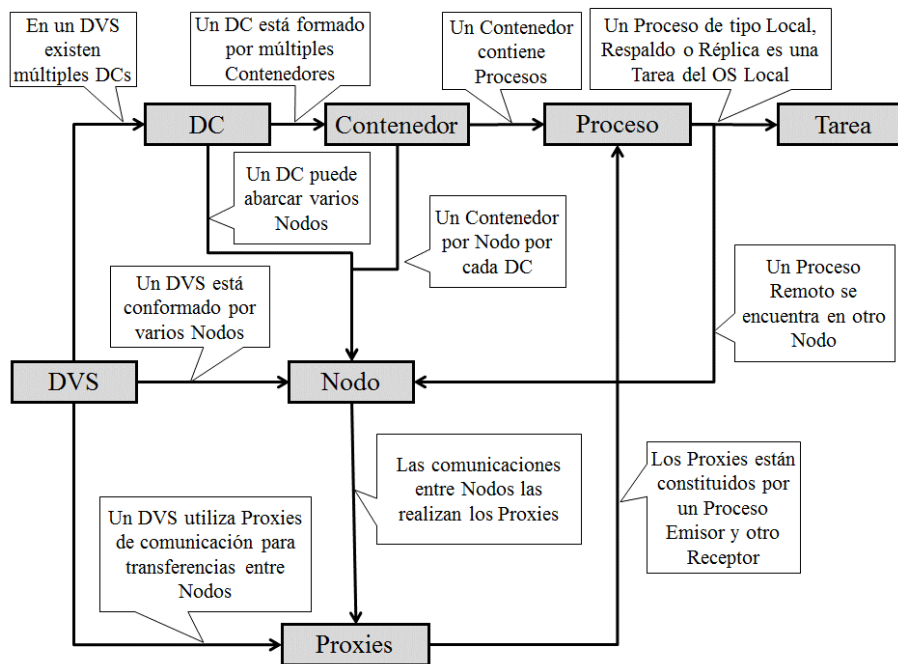


Figura 18. Relaciones entre Abstracciones.

Desde el punto de vista topológico, se pueden distinguir dos niveles de componentes. El nivel de DVS en el que los *Nodos* y *Proxies* especifican la constitución del cluster (los nodos que lo componen y como se comunican entre sí). El nivel de *DC*, en el que se mapean los *Nodos* del cluster DVS que abarcará cada *DC*. Cada *DC* está constituido por un conjunto de *Contenedores* individuales que son los entornos de ejecución para los VOS con sus *Procesos*.

3.2.2. Semántica del IPC

De acuerdo al tipo de *endpoint* de cada proceso en cada nodo, las transferencias de mensajes y datos (IPC en general) serán aceptadas o rechazadas entre procesos de un mismo *DC*.

En la Fig. 19, se presentan dos casos con tres procesos pertenecientes al mismo *DC-A* el cual está conformado por el *Contenedor-1* del *Nodo-1* y el *Contenedor-2* del *Nodo-2*. En cada *Contenedor* se visualiza la tabla de *endpoints* registrados en el módulo local del DVK. El proceso con

$endpoint=10$ y el proceso con $endpoint=100$ ejecutan en el *Contenedor-1* del *Nodo-1*. En tanto que el proceso con $endpoint=200$ ejecuta en el *Contenedor-2* del *Nodo-2*.

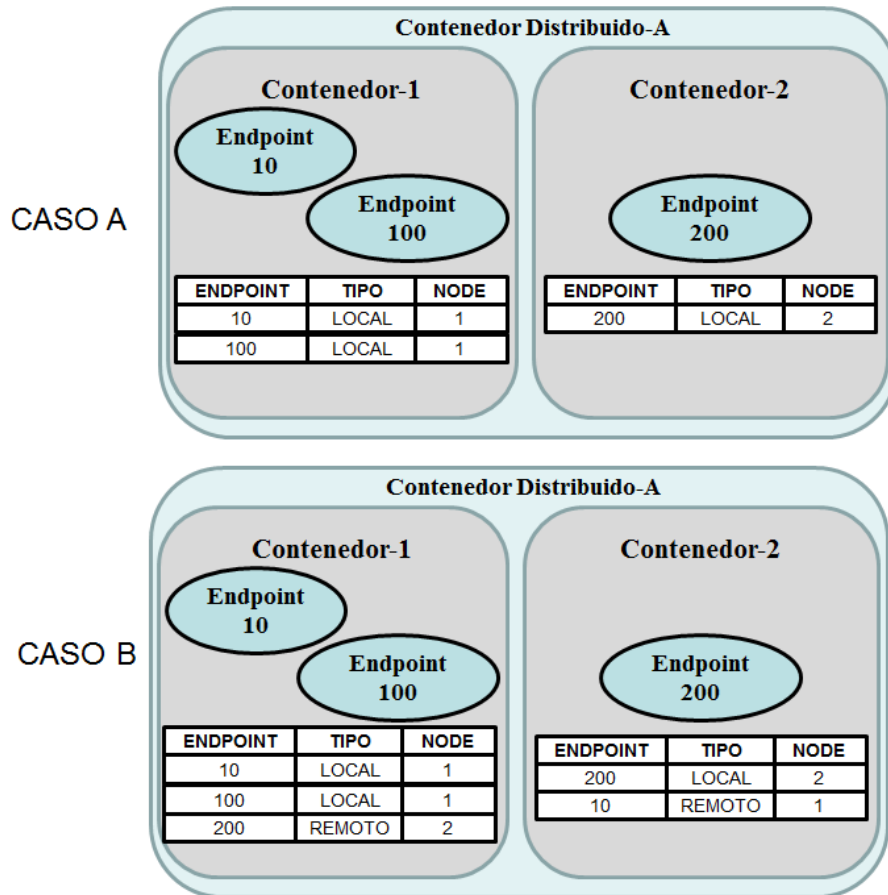


Figura 19. Endpoints Locales y Remotos.

En el Caso-A (Tabla IV), el proceso con $endpoint=100$ puede comunicarse con el proceso con $endpoint=10$ porque ambos están registrados en el mismo nodo (*Nodo-1*).

TABLA IV. SEMÁNTICA DE IPC PARA EL CASO-A

<i>Endpoint</i> Origen	<i>Endpoint</i> Destino	Resultado
100	10	Se efectúan transferencias en ambos sentidos
10	100	
200	10	Se rechazan las transferencias en ambos sentidos en el nodo Origen
10	200	
200	100	Se rechazan las transferencias en ambos sentidos en el nodo Origen
100	200	

Si el proceso con $endpoint=200$ quisiera comunicarse con el proceso con $endpoint=10$ o con $endpoint=100$ no lo podría hacer porque el módulo DVK de su nodo (*Nodo-2*) no los tiene

registrados. Además, el DVK del *Nodo-1* no tiene registrado al proceso con *endpoint=200*, por lo que se rechazaría cualquier transferencia de mensajes o datos proveniente de él.

Para el Caso-B (Tabla V), si en el *Contenedor-1* se registra el *endpoint=200* tipo *Remoto*, y en el *Contenedor-2* se registra el *endpoint=10* tipo *Remoto*, entonces se podrán comunicar los procesos con *endpoints* 10 y 200 entre sí, y también los de *endpoints* 10 y 100 entre sí. El proceso con *endpoint=100* podrá enviarle mensajes o datos al proceso con *endpoint=200* pero el modulo del DVK del *Nodo-2* rechazará esos mensajes, dado que no tiene registrado al *endpoint=100* como tipo *Remoto*.

TABLA V.SEMÁNTICA DE IPC PARA EL CASO-B

<i>Endpoint</i> Origen	<i>Endpoint</i> Destino	Resultado
100 10	10 100	Se efectúan transferencias en ambos sentidos
200 10	10 200	Se efectúan transferencias en ambos sentidos
200 100	100 200	Se rechaza la transferencia en el nodo Origen Se rechaza la transferencia en el nodo Destino

En la Fig. 20, se presentan dos casos con cuatro procesos registrados en el mismo *DC-A*.

Para el Caso-C (Tabla VI), el proceso con *endpoint=10* tipo *Local* y el proceso con *endpoint=100* tipo *Local* ejecutan en el *Contenedor-1* del *Nodo-1*. En tanto que el proceso con *endpoint=10* tipo *Respaldo* y el proceso con *endpoint=200* tipo *Local* ejecutan en el *Contenedor-2* del *Nodo-2*. Ambos *Contenedores* y sus procesos pertenecen al mismo *DC-A*.

El proceso con *endpoint=100* puede comunicarse con el proceso con *endpoint=10* porque ambos están registrados en el mismo nodo. El proceso con *endpoint=200* podrá comunicarse con el proceso con *endpoint=10* del *Nodo-1* a pesar de que existe un proceso con *endpoint=10* en su propio nodo, pero éste es ignorado porque es de tipo *Respaldo* del que se encuentra en *Nodo-1* (el *Primario*).

El comportamiento del *endpoint=10* en el *Nodo-2* es análogo a un *endpoint* de tipo *Remoto*, la diferencia está en que existe una tarea local que lo representa, pero la misma no recibirá mensajes ni podrá enviarlos hasta que no sea convertida (registrado su *endpoint*) en tipo *Local*, por ejemplo, como consecuencia de un fallo en el proceso *Primario*.

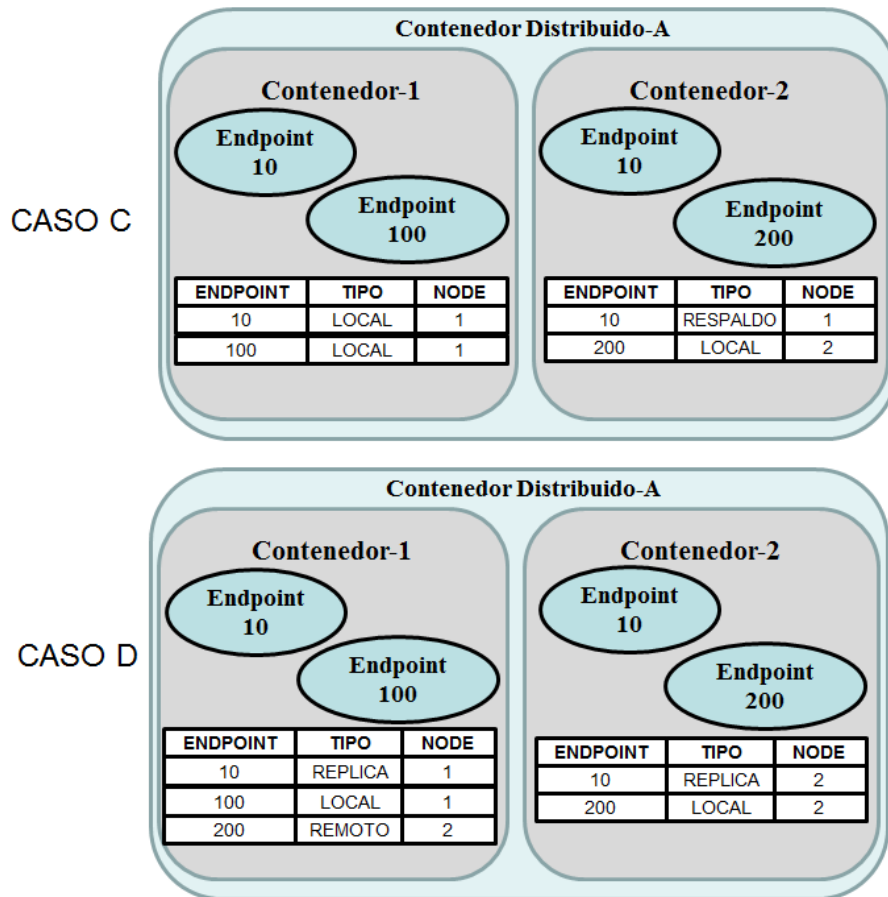


Figura 20. Endpoints tipo Respaldo y Réplica.

La conversión de tipo de *endpoint* no es realizada en forma automática por el DVK, sino que debe existir una aplicación de gestión con suficientes privilegios (puede ser el mismo VOS) que ejecute las APIs del DVK que la efectúa.

TABLA VI. SEMÁNTICA DE IPC PARA EL CASO-C

Endpoint Origen	Endpoint Destino	Resultado
100 10	10 100	Se efectúan transferencias en ambos sentidos
200 10	10 200	Se efectúan transferencias en ambos sentidos entre el <i>Endpoint=200</i> del <i>Nodo-2</i> y el <i>Endpoint=10</i> del <i>Nodo-1</i>
200 100	100 200	Se rechaza la transferencia en el nodo Origen Se rechaza la transferencia en el nodo Destino

Para el Caso-D (Tabla VII), en el *Contenedor-1* se registra el *endpoint=10* como tipo Réplica y en el *Contenedor-2* también se registra el *endpoint=10* como tipo Réplica. Entonces se podrán

comunicar los procesos con $endpoints=100$ y $endpoints=10$ del *Nodo-1* entre sí, como así también los $endpoints=10$ y $endpoints=200$ del *Nodo-2* entre sí.

TABLA VII. SEMÁNTICA DE IPC PARA EL CASO-D

Endpoint Origen	Endpoint Destino	Resultado
100 10	10 100	Se efectúan transferencias en ambos sentidos entre el $Endpoint=100$ del <i>Nodo-1</i> y el $Endpoint=10$ del <i>Nodo-1</i>
200 10	10 200	Se efectúan transferencias en ambos sentidos entre el $Endpoint=200$ del <i>Nodo-2</i> y el $Endpoint=10$ del <i>Nodo-2</i>
200 100	100 200	Se rechaza la transferencia en el nodo Origen Se rechaza la transferencia en el nodo Destino

En la Fig. 21, se presenta una forma de utilizar los *endpoints* de tipo Réplica para distribuir cargas. El proceso que se encuentra en el *Contenedor-2* del *Nodo-2* con $endpoint=10$ y el proceso que se encuentra en el *Contenedor-3* del *Nodo-3* con $endpoint=10$ son de tipo Réplica. Cuando el proceso con $endpoint=100$ del *Contenedor-1* del *Nodo-1* quiera comunicarse con el $endpoint=10$, lo hará con el proceso que se encuentra en el *Contenedor-2* del *Nodo-2* con $endpoint=10$, dado que lo tiene configurado como de tipo Remoto en *Nodo-2*. Cuando el proceso con $endpoint=400$ del *Contenedor-4* del *Nodo-4* quiera comunicarse con el $endpoint=10$, lo hará con el proceso que se encuentra en el *Contenedor-3* con $endpoint=10$ del *Nodo-3*, dado que lo tiene configurado como de tipo Remoto en ese nodo.

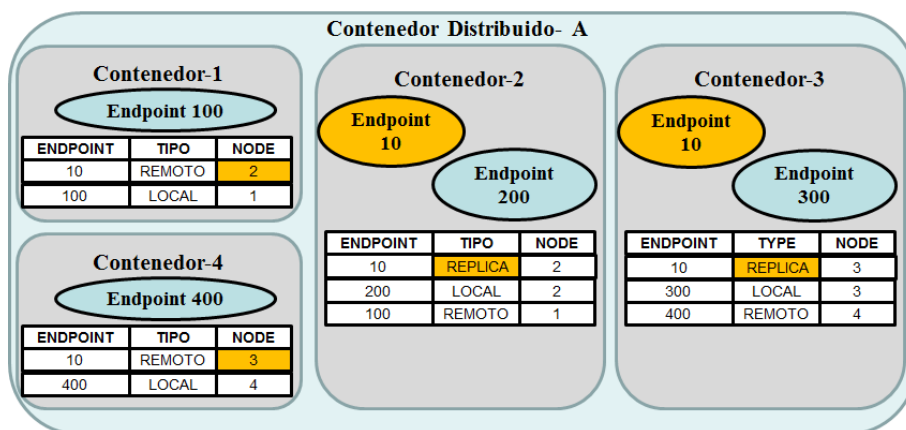


Figura 21. Distribución de cargas mediante *endpoints* tipo Réplica.

De esta forma, y mediante un algoritmo de distribución de cargas se puede conmutar la localización del proceso destino (que generalmente será un servidor) en los diferentes nodos que abarca el *DC*.

Este mecanismo de IPC que utiliza *endpoints* independiza de sus localizaciones a las aplicaciones que se comunican entre sí. La gestión de la localización de procesos y la registración de sus *endpoints* en el DVK la realizan procesos externos a la aplicación o el propio VOS.

Por ejemplo, cuando se decide migrar un proceso, otro proceso con privilegios de administrador informa al DVK (mediante el uso de sus APIs de gestión) que un *endpoint* dado perteneciente a un DC, migrará hacia otro nodo. El DVK detiene las comunicaciones que involucran a este *endpoint*, y retorna el control al proceso que controla la migración. Una vez finalizada la migración, se le informa al DVK (mediante otra API) que modifica las tablas de *endpoints* para reflejar la nueva localización del proceso migrado. No se consideran entonces recursos tales como direcciones IPs o puertos TCP/UDP que relacionan al proceso con su ubicación, lo que facilita su migración sin dejar dependencias residuales en el host origen.

Para ejemplificar la utilización de los tipos de *endpoints* se presenta las secuencias de operaciones a realizar para los casos de migración de un proceso, de replicación Pasiva y de replicación Activa. Se debe considerar para estos ejemplos que tanto los procesos como los *endpoints* refieren al mismo DC.

Para el caso de una migración de un proceso:

- a. En todos los nodos se detienen las transferencias de mensajes o datos que tengan al *endpoint* del proceso a migrar como origen o destino de dichas transferencias.
- b. Se realiza la migración de la tarea.
- c. En el nodo local al proceso (origen), se convierte su *endpoint* de tipo Local a tipo Remoto.
- d. Se crea (o convierte) el *endpoint* del nodo destino a tipo Local.
- e. En el resto de los nodos, los *endpoints* de tipo Remoto o Respaldo (si el proceso migrado fuese Primario) modifican el nodo del proceso migrado.
- f. Se reinician las transferencias de mensajes o datos hacia/desde ese *endpoint*.

Para el caso de Replicación Pasiva (Primario- Respaldo):

- a. Se crea un *endpoint* de tipo Local en un nodo que será el proceso Primario.
- b. Se crean múltiples *endpoints* de tipo Respaldo en otros nodos.
- c. Si se produce un fallo en el nodo o proceso Primario, en todos los nodos se detienen las transferencias de mensajes o datos que tengan al *endpoint* del proceso fallido como origen o destino de dichas transferencias.

-
- d. Se elige entre los procesos con *endpoint* de tipo Respaldo el nuevo proceso Primario.
 - e. En el nodo del Primario, se modifica el *endpoint* del proceso Primario de tipo Respaldo a tipo Local.
 - f. En el resto de los nodos con ese *endpoint* de tipo Respaldo o de tipo Remoto, se modifica el *endpoint* haciendo referencia al nuevo nodo donde se ejecuta el proceso Primario.
 - g. Se reinician las transferencias de mensajes o datos hacia/desde ese *endpoint*.

Para el caso de Replicación Activa (Máquina de Estados):

- a. En cada uno de los nodos del *DC* donde se ejecutan tareas replicadas se crea un *endpoint* de tipo Réplica.
- b. Si se produce un fallo en la réplica local de un nodo, se cambia el *endpoint* a tipo Remoto especificando el nodo de alguna de las réplicas que permanece activa.
- c. En todos los nodos que tenían como a la tarea fallida con *endpoint* de tipo Remoto, se cambia el valor del nodo para referir alguna de las réplicas que permanece activa.
- d. El resto de los nodos activos con *endpoints* tipo Réplica no se modifican.

Para separar privilegios entre tareas componentes del VOS y tareas componentes de aplicaciones, existen dos tipos de *endpoints*: Privilegiados y No Privilegiados. Los *endpoints* Privilegiados pueden realizar tareas de administración del *DC* utilizando las APIs de gestión del *DVK*, utilizar las APIs de IPC y realizar Llamadas al Sistema del OS-Host. Entre esas APIs que pueden ejecutar los procesos con *endpoints* Privilegiados están aquellas que permiten la registración y de-registración de procesos a un *DC*. Esto se utiliza generalmente, para que un proceso privilegiado pueda registrar a sus procesos hijos dentro del mismo *DC*. Vale aclarar que los procesos hijos de procesos registrados en un *DC* no heredan la registración, debe registrárselos en forma explícita.

Las tareas con *endpoints* No Privilegiados solo pueden utilizar las APIs de IPC. Cualquier servicio adicional que la aplicación requiera deberá solicitarlo al VOS, el cual eventualmente, podrá actuar de pasarela (Gateway) y/o de intermediario hacia algún servicio ofrecido por el *DVK*.

Es responsabilidad del VOS registrar los *endpoints* correctamente en cada nodo. También es responsabilidad del VOS la asignación de *endpoints* con sus correspondientes privilegios a sus tareas como así también a las aplicaciones que se ejecutan sobre él.

3.3. Modelo de Arquitectura de DVS

El modelo de arquitectura de *DVS* está conformado por tres capas (Fig. 22).

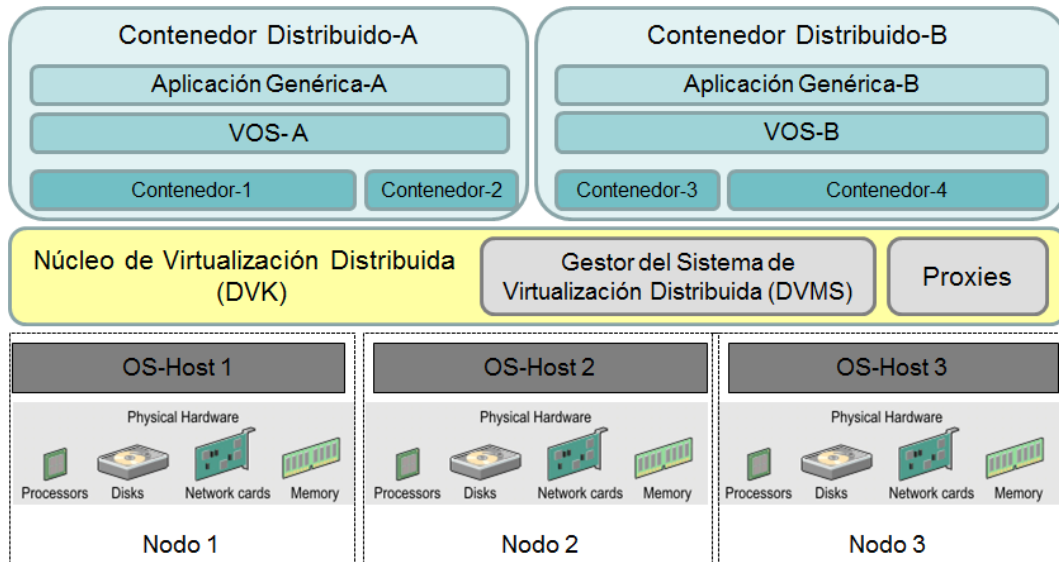


Figura 22. Modelo de Arquitectura del *DVS*.

En su capa inferior se encuentra el hardware, con su OS-host que tiene embebido un módulo que conforma uno de los componentes del *DVK*. La capa intermedia la conforma el *DVK*, el Sistema de Gestión de Virtualización Distribuida (*DVMS*: Distributed Virtualization Management System) y los *proxies*. El *DVK* está compuesto por una serie de servicios que se ofrecen a la capa superior a través de sus APIs, y los cuales utilizan servicios de la capa inferior a través del módulo embebido. El *DVMS* dispone de las APIs, comandos, bibliotecas, interfaces gráficas o interfaces web para gestionar el *DVS* y todos sus componentes.

Los *proxies* son los procesos que transfieren mensajes y datos entre los nodos del cluster del *DVS*. Finalmente, la capa superior la conforman los *DCs*, los Contenedores, los VOS y las aplicaciones.

3.4. Descripción de los Componentes del DVS

En las siguientes subsecciones se detalla la funcionalidad de cada uno de los componentes del modelo de arquitectura del *DVS*.

3.4.1. Núcleo de Virtualización Distribuida (DVK)

El *DVK* es la capa de software principal que integra los recursos del cluster, gestiona y limita los recursos asignados a cada *DC*. Ofrece a las capas superiores las interfaces necesarias para utilizar los protocolos y servicios de bajo nivel, los cuales pueden ser utilizados por los VOS, tales como Planificación Distribuida, Replicación Activa/Pasiva, Exclusión Mutua Distribuida, mecanismos de migración de procesos, monitoreo de parámetros de rendimiento, Memoria Compartida Distribuida, Instantáneas Distribuidas, Multicast tolerante a Fallos, Membresía de Grupos, Consenso Distribuido, IPC Cliente/Servidor, servicios de clave/valor, autenticación, detección de fallos, elección de Lider, etc. (Fig. 23). Estos protocolos y servicios son similares a los existentes en un DOS, con la diferencia que en el *DVK* los mismos son autónomos, y que pueden ser utilizados para el desarrollo tanto de los VOSs como de aplicaciones centralizadas o distribuidas. El enfoque es similar al de la modalidad *Sin Servidor* (Serverless), en este caso las aplicaciones y los VOSs hacen uso de los servicios que provee el *DVK*.

Cada servicio del *DVK* que se ofrece a las capas superiores (GCS, exclusión mutua distribuida, consenso distribuido, etc.) debe tener en consideración que los procesos que involucran deben pertenecer al mismo *DC*, tal como lo hace el mecanismo de IPC. Es función del *DVK* controlar el aislamiento entre *DCs* para todos los servicios.

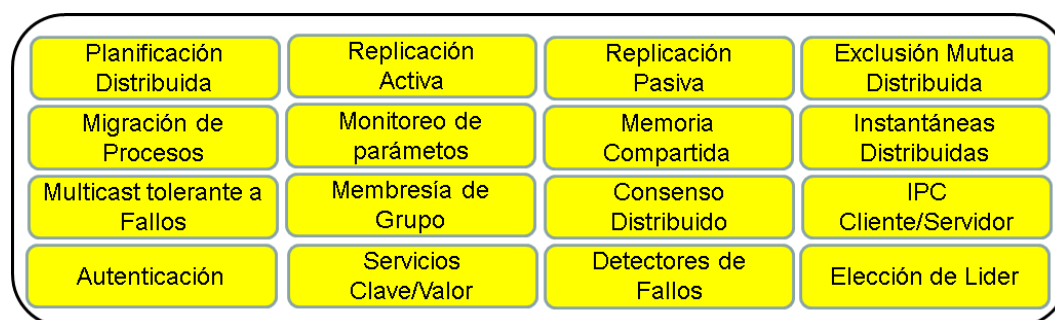


Figura 23. Servicios del Distributed Virtualization Kernel (*DVK*).

El *DVK* además provee las interfaces para gestionar todos los recursos del DVS tales como *Nodos*, *Proxies*, *Contenedores*, *DCs* y *Procesos*. La gestión de procesos le permite al administrador del *DVS* asignar procesos a *DCs* y asignarle nodos al *DC*. El nodo donde ejecuta un proceso puede cambiar cuando se produce una migración o cuando un proceso (por ejemplo, un Primario) es reemplazado por otro proceso de tipo Respaldo. El mecanismo de IPC utilizado debe ocultar los cambios en la localización de un proceso a los otros procesos dentro del mismo *DC*, es decir debe brindar transparencia en la localización.

3.4.2. Representante de Comunicaciones de Nodos Remotos (Proxies)

El cluster donde se ejecutará el *DVS* se define básicamente utilizando nodos y *proxies*. Los *proxies* de comunicaciones son los procesos responsables de transferir mensajes y bloques de datos entre un nodo Local y uno o más nodos remotos. La existencia de *proxies* se oculta al resto de los componentes del *DVS* tales como *DCs*, Contenedores y procesos registrados. Este modelo de uso de *proxies* es también utilizado por algunos DOS, por ejemplo FOS [73].

Los *proxies* no consideran el *DC* o proceso destinatario de un mensaje o de un bloque de datos para realizar su tarea (aunque lo pueden hacer), solo requieren conocer el nodo destino. En un *DVS*, todos los nodos deben estar totalmente conectados (full mesh) entre sí en forma lógica. Por ejemplo, pueden estar en diferentes redes IP, pero un proxy de un nodo en una red debe poder comunicarse en forma directa con el Proxy del nodo de la otra red utilizando por ejemplo protocolo TCP.

En la Fig. 24, se presenta un cluster conformado por 3 nodos (NODO0, NODO1, NODO2), unidos por *proxies* totalmente conectados. Como se puede ver, a través de los enlaces lógicos entre los *proxies* pueden transferirse mensajes y datos pertenecientes a procesos de diferentes *DCs* (distinguidos por diferentes colores).

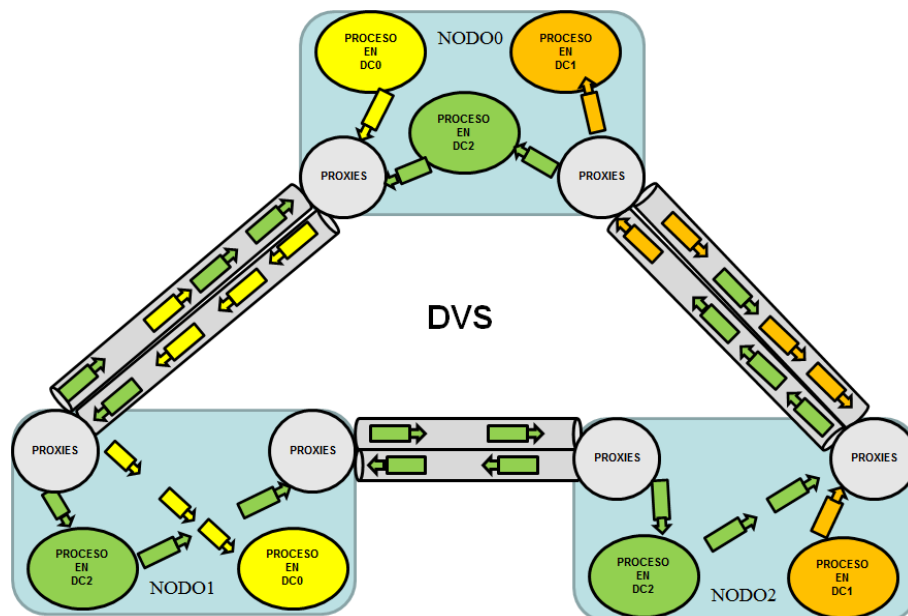


Figura 24. *Proxies* de Comunicaciones

El modelo plantea dos tipos de procesos que conforman una entidad Proxy; un proceso proxy Emisor responsable de enviar mensajes o datos salientes del nodo Local, y un proceso proxy Receptor responsable de recibir mensajes o datos destinados a procesos del nodo Local. Los procesos o hilos que conforman el par Emisor/Receptor deben registrarse ante el *DVK* indicando cuáles son los nodos

que representa. En el caso de mensajes y datos entrantes, el Proxy Receptor a través de las APIs que ofrece el *DVK*, encola el mensaje o bloque de datos en el descriptor del proceso destino. Para el caso de mensajes o datos salientes, el proceso emisor encola el mensaje en el descriptor del proceso Proxy Emisor y este se encarga de enviarlo al nodo Remoto donde reside el proceso (*endpoint*) destinatario del mensaje o del bloque de datos.

El modelo de DVS no establece ninguna restricción en lo que refiere a la implementación de los *proxies*. Los mismos pueden implementarse tanto en modo usuario o integrarse al kernel de cada uno de los nodos, pueden utilizar procesos o utilizar hilos (threads). Por otro lado, tampoco hay restricciones respecto a los protocolos a utilizar, al cifrado o compresión de datos y mensajes, etc. Es posible utilizar un único par de procesos Emisor/Receptor para las transferencias de mensajes con el resto de los *proxies* del cluster, o usar un par de procesos Emisor/Receptor por nodo o cualquier combinación.

3.4.3. Contenedores

Los Contenedores son entornos de ejecución aislados que brinda el OS y que ya fueron presentados en la [Sección 2.1.3](#). Entre sus características significativas para su utilización en el *DVS* se destaca el aislamiento y el espacio de nombres (*namespace*) propio. El aislamiento de rendimiento se logra limitando ciertos recursos del OS asignados a cada Contenedor tales como tiempo de CPU, buffers, caches, reserva de ancho de banda de la red, E/S de dispositivos, memoria, etc. El aislamiento de seguridad y de fallos está relacionado en parte, con la asignación del espacio de nombres propio donde los procesos dentro de un Contenedor no pueden acceder a objetos fuera de éste, reduciendo así las posibilidades de que un defecto en un proceso o un ataque de seguridad puedan afectar a otros Contenedores o al propio OS.

Generalmente, los Contenedores se utilizan para aislar las aplicaciones entre sí. Dentro de cada Contenedor se ejecutan aplicaciones que, directamente o a través de bibliotecas, utilizan las Llamadas al Sistemas del OS-host para solicitar sus servicios. Es decir, las aplicaciones acceden a los servicios del OS-host sin restricciones. En el modelo de arquitectura de *DVS* se propone que las aplicaciones no accedan directamente a los servicios del OS-hosts, sino que utilicen los servicios de un VOS intermediario (Fig. 25). Las aplicaciones entonces utilizan los servicios del VOS, y es el VOS el que utiliza los servicios y recursos del OS-host a modo de paravirtualización.

De esta forma, la arquitectura no restringe sus servicios únicamente a los que ofrece el OS-host, sino que se pueden ofrecer diferentes VOS. Particularmente, un VOS distribuido (DVOS:

Distributed Virtual Operating System) podría obtener un mejor aprovechamiento, una mayor escalabilidad, disponibilidad y elasticidad de los recursos del cluster.

Por otro lado, estableciendo privilegios y capacidades diferentes para los procesos que conforman el VOS respecto a los procesos de las aplicaciones, se logra un nivel adicional de aislamiento a costa de un nivel adicional de indirección.

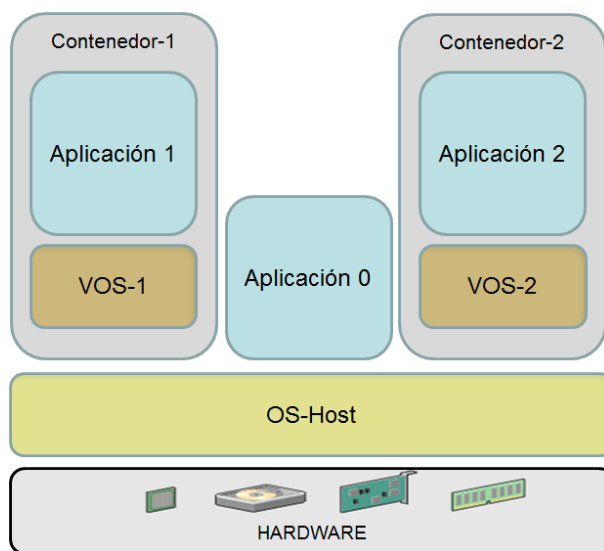


Figura 25. Utilización de Contenedores propuesta.

En la Fig. 25, Aplicación-0 es nativa del OS-host y ejecuta fuera de un Contenedor. En tanto que la Aplicación-1 ejecuta dentro del Contenedor-1 utilizando los servicios del VOS-1, y la Aplicación-2 ejecuta dentro del Contenedor-2 utilizando los servicios del VOS-2. Aplicación-1 y Aplicación-2 están aisladas entre sí, y éstas respecto a Aplicación-0 y no tiene acceso a ningún recurso por fuera del Contenedor.

3.4.4. Contenedores Distribuidos

Actualmente, las aplicaciones desarrolladas para ejecutar en la Nube utilizan un conjunto de VMs o Contenedores para obtener mayor rendimiento, escalabilidad y elasticidad. Su despliegue y gestión lo basan en Gestores de Contenedores (Containers Managers) o Sistemas Operativos para la Nube (Cloud Operating Systems). Este conjunto de aplicaciones ejecutando en Contenedores aislados se comunican entre sí utilizando protocolos diversos que tratan de imponerse como estándares. Uno de los aspectos más críticos de los sistemas distribuidos es la coordinación y sincronización de acciones, aún en presencia de fallos. La programación de este tipo de aplicaciones es una tarea compleja que requiere mucho conocimiento por parte de los desarrolladores [84]. Esta solución no

presenta a los programadores ni a los administradores una visión integrada de la aplicación, a pesar de que se dispone de múltiples herramientas que ayudan a estas tareas.

El modelo de arquitectura de *DVS* propone una extensión a la abstracción de los Contenedores que se denomina Contenedor Distribuido (*DC*). Un *DC* también es un conjunto de Contenedores dispersos en diferentes nodos de un cluster, pero que permite comunicar los procesos de un Contenedor con los procesos de otro Contenedor en diferente nodo utilizando un IPC transparente a la localización. El mecanismo de IPC solo admite la comunicación entre procesos que se ejecutan en un mismo *DC*, estén estos localizados en el mismo o en diferentes nodos del cluster de virtualización. A cada Contenedor individual que conforma el *DC*, se le asignan recursos del propio nodo, de la misma forma que lo hacen actualmente las aplicaciones desarrolladas para la Nube.

Un *DC* está constituido por un conjunto de Contenedores y puede abarcar desde uno a todos los nodos del cluster. Si el *DC* abarca más de un nodo, entonces habrá un Contenedor por *DC* por cada nodo. Cada *VOS*, centralizado o distribuido ejecuta junto a sus aplicaciones dentro de un *DC*. Si se ejecuta un *DVOS*, sus componentes se pueden ejecutar distribuidos en los diferentes Contenedores que conforman el *DC*, lográndose así agregar las ventajas de los sistemas distribuidos (mayor rendimiento, escalabilidad, disponibilidad y elasticidad), las ventajas de los SSI (simplificación en programación, despliegue, operación y mantenimiento) y las ventajas de la virtualización (aislamiento, migración, portabilidad y clonación). Los *DCs* proveen a los administradores del *DVS* de un control fino y detallado de los recursos del cluster de virtualización para partirlos de manera precisa e inteligente y así aumentar la eficiencia en su utilización.

Pero, ¿por qué razón se necesita un DC en el modelo de arquitectura de DVS? Porque un Contenedor sencillo limita el acceso a recurso del propio OS-host. Como el *DVK* ofrece servicios de IPC entre procesos locales entre sí o con procesos ejecutando en diferentes nodos, los mensajes pueden atravesar las fronteras de un nodo para ser entregados en otro nodo. Para completar el aislamiento de aplicaciones de la arquitectura, un proceso perteneciente a un *DC* no debe poder comunicarse mediante IPC con otro proceso que se encuentre fuera del propio *DC*. El *DC* brinda aislamiento de seguridad entre procesos de diferentes *DCs* y permite la comunicación entre procesos del mismo *DC*, aunque se encuentren en diferentes nodos. El aislamiento de performance se basa en la configuración de cada Contenedor que compone el *DC* dado que afecta al OS-host.

3.4.5. Sistema Operativo Virtual

Como se mencionó en la [Sección 2.1.5](#), un VOS ofrece servicios a las aplicaciones como un OS típico, pero no opera directamente con el hardware. Utiliza los servicios y recursos abstractos de un OS-host. La complejidad de desarrollar un VOS no es la misma que desarrollar un OS típico. Como un VOS no interactúa con el hardware no requiere de Gestor de Dispositivos, pero sí puede incorporar Gestores de Dispositivos Virtuales para realizar operaciones de Entrada/Salida utilizando los recursos virtuales que ofrece el *DVK*.

El desarrollo de un DVOS se facilita si utiliza los protocolos y servicios que ofrece el *DVK*, tales como exclusión mutua distribuida, servicio de elección de líder, servicio de sincronización distribuida, migración de procesos, etc. Generalmente, las aplicaciones desarrolladas para la nube despliegan un conjunto de VOSs de tipo Unikernel que requieren de la sincronización entre procesos, de consenso distribuido, de comunicaciones grupales, etc. Todos estos servicios están disponibles en el *DVK*, simplificando así el desarrollo de las aplicaciones y reduciendo sus requerimientos de memoria. Esta característica es similar al actual modelo de desarrollo de aplicaciones para ejecutar en la Nube tipo *Sin Servidor*, donde servidores de Complementarios o de Backend provistos por el *DVK* son utilizados por las aplicaciones, y en este caso por los VOS.

Pero, ¿porque se necesita de un VOS para la arquitectura? Porque la utilización de VOS tiene varios beneficios:

- Las aplicaciones no pueden realizar Llamadas al Sistema al OS-host en forma directa. Esta característica reduce la cantidad de dependencias respecto al nodo donde ejecuta, minimizando los recursos del OS-host asignados a la aplicación, lo que facilita el proceso de migración.
- Cada VOS provee su propio conjunto de Llamadas al Sistema, recursos abstractos, APIs, bibliotecas, etc. sin considerar lo ofrecido por el OS-host.
- Un VOS ofrece un nivel adicional de aislamiento de fallo y de seguridad porque ejecuta en modo-usuario.
- Diferentes VOS pueden ejecutar en el mismo *DVS*, lo que no es posible realizar con Contenedores ordinarios donde las aplicaciones utilizan los servicios del único OS-host.

A continuación, se analizarán diferentes alternativas de VOS que pueden ejecutar dentro de un *DC*, mencionando sus características, ventajas e inconvenientes.

❑ VOS Monolítico

Un VOS monolítico es aquel en el que su kernel constituye un único programa ejecutable en un único espacio de direcciones. Como el *DVS* es un sistema distribuido, un VOS monolítico no parecería obtener rédito en la su ejecución en un *DVS*.

Para que un VOS monolítico pueda expandirse más allá de los límites del nodo que lo contiene, debería aplicarse una estrategia similar a los sistemas de virtualización actuales que utilizan, por ejemplo, discos virtuales de una SAN. Es decir, se deberían desagregar aquellos componentes que puedan ejecutarse en forma autónoma en otros nodos diferentes al del kernel del VOS.

En la Fig. 26, un Servidor de Sistema de Archivos y un Servidor de Disco o Almacenamiento ejecutan en un nodo diferente del que ejecuta el VOS-monolítico. Para ello, el VOS monolítico debe instalar nuevos módulos para comunicarse con sus servidores.

Esta configuración no parece ser diferente de un OS monolítico ejecutando dentro de una VM en un nodo dado, accediendo a los servicios de un Servidor de Sistema de Archivo tal como NFS y de un Servidor de Disco tal como NBD o DRBD ejecutando en otro nodo. Existen varias diferencias entre la configuración mencionada y la ejecución de un VOS monolítico en el *DVS*. Estas son:

- Tanto el NFS como el NBD/DRBD utilizan sus propios protocolos para la comunicación con el OS, mientras que para el VOS monolítico la programación se simplifica porque utiliza los mecanismos de IPC que ofrece el *DVK* para comunicarse con sus servidores, el cual es transparente a la localización de los procesos.
- Tanto el Servidor de Sistema de Archivos como el Servidor de Disco pueden ejecutar replicados en otros nodos para adquirir mayor disponibilidad incluso ante ocurrencia de fallos, siendo esta replicación transparente para el VOS monolítico. Los servidores replicados utilizan los servicios ofrecidos por el *DVK* para mantener sus réplicas sincronizadas.
- Tanto el Servidor de Sistema de Archivos como el Servidor de Disco pueden migrar a otros nodos, siendo esta migración transparente para el VOS monolítico. Los servidores migrados utilizan los servicios ofrecidos por el *DVK* para realizar la migración y para mantener sus identificaciones (*endpoints*) aun cuando hayan cambiado de nodo.

-Tanto el servidor NFS como el servidor NBD/DRBD requieren de una VM completa ejecutando con su propio OS-Guest. En tanto que el Servidor de Sistema de Archivos y el Servidor de Disco ejecutan dentro de un *DC* utilizando los servicios del *DVK* y del OS-Host.

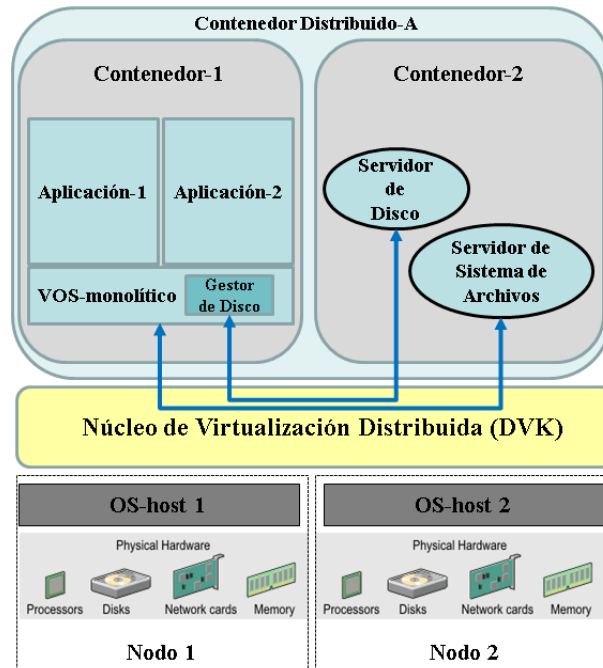


Figura 26. Ejecución de un VOS monolítico en un DVS.

Aunque los mayores beneficios de un *DVS* se obtienen con la descomposición o factorización del VOS y de las aplicaciones, un VOS monolítico puede mejorar su desempeño desagregando algunos de sus componentes para obtener mayor rendimiento, elasticidad y disponibilidad.

□ VOS Unikernel

Un VOS-Unikernel está constituido por un único programa ejecutable en un único espacio de direcciones. A ese programa lo conforman los códigos de las bibliotecas que gestionan los dispositivos tales como disco e interfaces de red, proveen protocolos tales como TCP/IP, brindan servicios de archivos, etc. y por la aplicación. Un Unikernel ordinario generalmente obtiene servicios de un hipervisor, pero un VOS-Unikernel utiliza los servicios que ofrece el *DVK*.

Al igual que lo que sucede con un VOS-monolítico, a-priori un VOS-Unikernel no parecería obtener rédito en la su ejecución en un DVS. Pero, de igual forma, éste puede expandirse más allá de los límites del nodo que lo contiene aplicando una estrategia similar a los sistemas de virtualización actuales que utilizan, por ejemplo, discos virtuales de una SAN. Es decir, se deberían desagregar

aquellos componentes que puedan ejecutarse en forma autónoma en otro nodo diferente al del VOS-Unikernel.

Generalmente, las aplicaciones en para ejecutar en la Nube son desarrolladas bajo MSA o SOA, despliegan sus procesos en enjambre (swarm) de Unikernels en diferentes VMs. Como todos procesos distribuidos relacionados, necesitan coordinar sus tareas, intercambiar datos y sincronizar sus operaciones. Estos se comunican utilizando protocolos ad-hoc, para lo cual deben configurar direcciones IPs de las VMs, habilitar de puertos y direcciones en los filtros de los firewalls que los protegen, mecanismos de autenticación y autorización, etc. Además, el administrador no tiene una visión integrada de la aplicación y sus recursos.

Una solución más elegante y simple es desplegar los diferentes VOS-Unikernel en diferentes nodos de un cluster *DVS* como procesos de un mismo *DC* (Fig. 27).

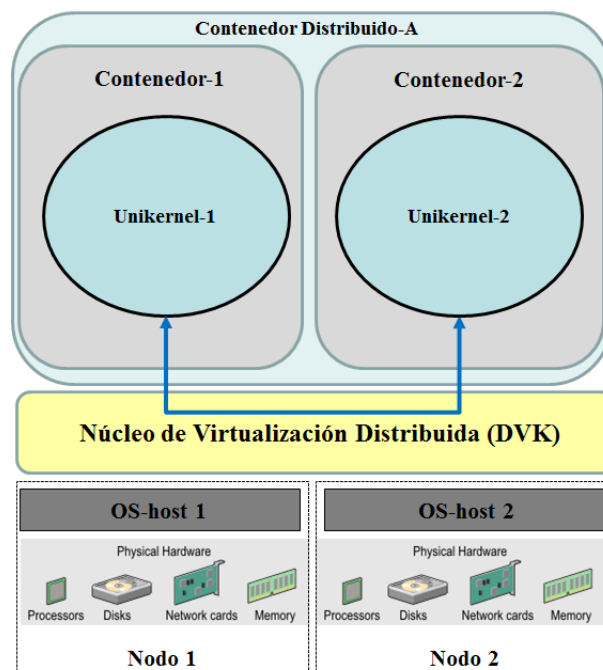


Figura 27. Ejecución de Unikernels en DCs.

Cada VOS-Unikernel tiene asignado su propio *endpoint* (o más de uno si fuese el caso), que le permite comunicarse con los otros VOS-Unikernel mediante mecanismos simples de IPC. Como el *DVS* soporta migración y replicación de procesos, se hace más sencillo incorporar estas características como parte de la solución para la ejecución de la aplicación.

El *DVK* provee los mecanismos de IPC, GCS, detección de fallos, dispositivos virtuales, etc. que pueden ser utilizados tanto por los VOS-Unikernel, como por las aplicaciones que se ejecutan integradas en cada uno de ellos.

❑ VOS Distribuido

Diseñar e implementar un OS es una tarea compleja que involucra dar tratamiento a las interrupciones, fallos y excepciones, a gestionar procesos, memoria, archivos, interfaces con las aplicaciones, interfaces de texto y gráficas, etc. Diseñar e implementar un DOS es una tarea aún más compleja porque no se dispone de un temporizador común a todos los nodos de un cluster que facilite el ordenamiento de los eventos mediante estampillas de tiempo (timestamps), ni tampoco de una memoria común que permita compartir estructuras de datos. Adicionalmente, se debe considerar que los nodos del cluster se encuentran enlazados por una red que generalmente no da garantías de entrega, de orden de entrega, ni de tiempo de entrega.

Los DOS deben resolver problemas adicionales a los OS centralizados tales como la exclusión mutua distribuida, la replicación de datos y procesos, la planificación distribuida, la detección de fallos, la distribución de cargas, la migración de procesos, etc. Si bien un DVOS comparte las características de un DOS y debe resolver los mismos problemas, su diseño e implementación se ven simplificados porque muchos de ellos se resuelven utilizando los servicios que ofrece el *DVS* a través de su *DVK*.

En la Fig. 28, se presenta un ejemplo de DVOS ejecutando en DC-A, el cual está constituido por el Contenedor-1 del Nodo-1 y el Contenedor-2 del Nodo-2. Los componentes del DVOS se encuentran distribuidos comunicados entre sí mediante los mecanismos de IPC cliente/servidor que ofrece el *DVK*.

En este ejemplo, la Tarea del Sistema encuentra replicada en ambos Contenedores de ambos nodos, uno atendiendo las peticiones de los procesos del Contenedor-1 y otro las peticiones de los procesos del Contenedor-2. Estos procesos replicados deben coordinar sus acciones para evitar conflictos, inconsistencias o condiciones de competencia.

Por ejemplo, si un proceso del Contenedor-1 realiza una llamada al sistema *fork()* (creación de un proceso hijo), el DVOS debe asignarle un PID único a ese nuevo proceso. Si en forma simultánea a ese *fork()*, otro proceso del Contenedor-2 realiza su propio *fork()*, también a ese nuevo proceso debe asignársele un PID. Los PIDs son números enteros secuenciales que se inician con el valor 1 (uno). Generalmente, el DVOS mantiene el último número asignado o el próximo a asignar, en este caso ambas réplicas mantienen el mismo valor para esta variable. Si las réplicas acceden a sus propias variables replicadas en forma concurrente, ambas podrían obtener el mismo número, por lo que se asignaría el mismo PID a diferentes procesos generando una inconsistencia.

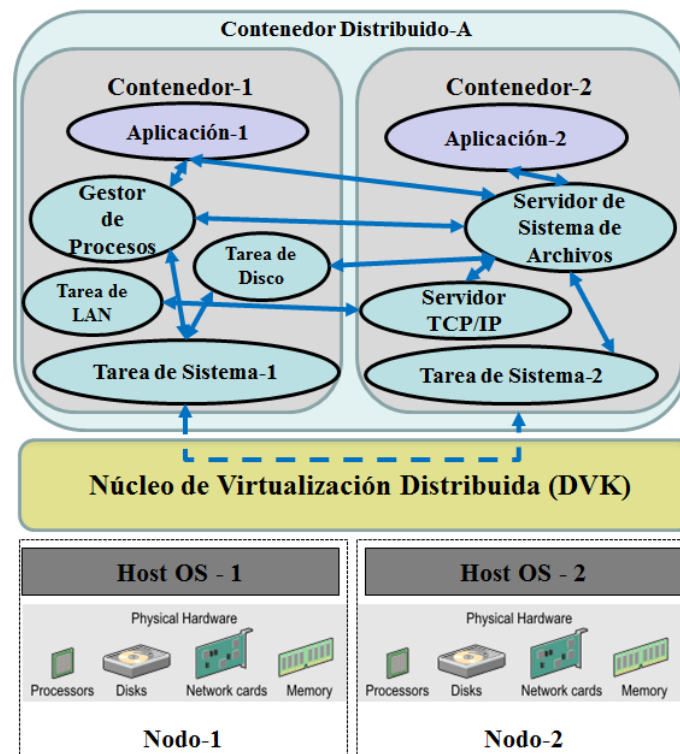


Figura 28. VOS Distribuido.

Para evitar el acceso concurrente a los recursos compartidos o replicados, las Tareas del Sistema pueden utilizar las facilidades de GCS que ofrece el *DVK* que les permitirá acceder en forma secuencial a la variable que conserva el valor del próximo PID a asignar. Estas facilidades y otras requeridas por un DVOS están disponibles en el *DVK* como servicios, facilitando así su desarrollo.

Se hace evidente que los DVOS se adaptan perfectamente a la arquitectura de un *DVS* porque sus procesos pueden distribuirse en diferentes nodos expandiendo su capacidad de procesamiento y almacenamiento, replicarse para obtener tolerancia a fallos o mejores tiempos de servicio y migrarse para distribuir las cargas. Todos estos beneficios se complementan con las características de aislamiento, consolidación y partición de recursos que ofrecen las tecnologías de virtualización.

3.4.6. Relaciones entre Procesos y Endpoints

Todos los procesos y tareas que conforman los VOSs y las aplicaciones deben estar registrados ante el *DVK* para poder utilizar sus servicios. Los *endpoints* identifican a los procesos como pertenecientes a un *DC*, por lo que en un mismo nodo pueden existir varios procesos con el mismo *endpoint*, pero en diferentes *DCs*. Generalmente la relación entre procesos ordinarios y *endpoints* es biunívoca dentro del *DC*, con las excepciones de los *endpoints* de tipo Respaldo y

Réplica. Cada *DC* tiene configurado un número máximo de *endpoints*. La asignación de un *endpoint* a cada proceso es realizada por el VOS que ejecuta en ese *DC*, aunque lo puede hacer cualquier proceso con los privilegios adecuados.

Las APIs de IPC utilizan los *endpoints* como destinatarios de las transferencias de mensajes o bloque de datos, o como origen de mensajes esperados para ser recibidos. Existen *endpoints* especiales que permiten identificar un conjunto de procesos tal como *ANY*, que se utiliza para recibir mensajes de cualquier *endpoints* origen y no de uno en particular. Por otro lado, los *endpoints* pueden ser Privilegiados o No Privilegiados, siendo éste un atributo que habilita o no al proceso a utilizar servicios del *DVK* adicionales a IPC. El mapeo entre los diferentes *endpoints* y los privilegios que éste posea son específicos de la implementación del *DVK*.

Como se vió en la [Sección 3.2](#), el tipo de un *endpoint* dado en un nodo puede ser diferente del tipo de *endpoint* en otros nodos. Por ejemplo, un proceso con *endpoint=10* del *DC-A* en el *NODO1*, se encuentra registrado en el módulo de *DVK* de dicho nodo como Local, pero como tipo Remoto en el módulo de *DVK* del *NODO2*.

3.5. Otros Componentes Requeridos

El modelo de arquitectura de *DVS* fue diseñado para expandir el alcance del entorno aislado de ejecución de aplicaciones por fuera de los límites de un único host. Los proveedores de Servicios en la Nube son los grandes usuarios de las tecnologías de virtualización requiriendo de éstas determinadas características que les permitan brindar servicios de mayor calidad y confiabilidad. Estas características incluyen mecanismos de tolerancia a fallos para lograr mayor disponibilidad, balanceo de carga para obtener mayor rendimiento de la infraestructura, elasticidad para ahorro energético y provisión de servicios, facilidad en la gestión de servicios y mecanismos de control de utilización para la facturación de los mismos, entre otros.

Por su característica modular un *DVS* es posible incorporar servicios al *DVMS* y/o al *DVK* que permiten a los administradores, VOS y aplicaciones hacer uso de ellos. A continuación, se presentan algunas características que un *DVS* posee para cumplir con los requerimientos de los proveedores de Servicios en la Nube.

3.5.1. Replicación Primario-Respaldo

La replicación es un mecanismo utilizado para tolerar fallos y mejorar el rendimiento, una de sus variantes es la Replicación Pasiva o Primario-Respaldo [99]. En el cluster se conforma un grupo de procesos servidores (uno en cada nodo) denominados Réplicas. Mediante los servicios provistos por el *DVK*, se elige a una de esas réplicas como *Primario*, el resto son réplicas *Respaldo*.

Los clientes del servicio solo se comunican con la réplica Primario, mientras que ésta se comunica con el resto de las réplicas utilizando GCS. Generalmente la réplica Primario utiliza difusión tolerante a fallo con entrega con orden FIFO para enviar mensajes a las réplicas Respaldo, lo que hace que la sobrecarga y la latencia producida por el protocolo de difusión sean bajas.

Cuando se produce un fallo en la réplica Primario, mediante detectores de fallo que posee el GCS, alguna de las réplicas tipo Respaldo lo detecta e invoca a un algoritmo de elección de líder para designar la nueva réplica Primario. El nuevo líder designado se transforma en Primario y atenderá los requerimientos de los clientes.

Como se debe mantener el servicio con transparencia de fallos para los clientes, es necesario que éstos no detecten el fallo del Primario original y que el mismo ha sido reemplazado. Para ello se requiere que el nuevo Primario adopte la misma dirección que el original y continúe con los servicios a clientes tal como que no hubiese ocurrido un fallo. En el *DVS*, los mecanismos de IPC que provee el *DVK* permiten hacer transparente el cambio de Primario a las aplicaciones clientes. El *DVK* permite definir el *endpoint* del Primario como tipo Local a un nodo, el cual será el proceso activo que recibirá las peticiones de los clientes, y múltiples *endpoints* tipo Respaldo en otros nodos que se comunican con el Primario a través de un GCS, pero que no intercambian información con los clientes.

Cuando el Primario falla, es detectado por el detector de fallos informando (a través del *DVK*) al resto de los nodos de esta situación, los cuales eligen la nuevo Primario. En el nodo donde ejecuta el Primario se modifica el tipo de *endpoint* del proceso Respaldo elegido a tipo Primario, el resto de los nodos modifican en los *endpoints* de los Respaldo el nodo donde reside el nuevo Primario. Mientras tanto las comunicaciones entre los clientes y el (nuevo) Primario se encuentran suspendidas a la espera de estas modificaciones, solo impactando en la latencia percibida por los clientes. Una vez finalizada las operaciones, las comunicaciones entre los clientes y el Primario se reestablecen.

Es importante destacar que la *Transparencia de Fallos* que ofrece el *DVK* en sus mecanismos de IPC facilita el desarrollo de aplicaciones que utilizan el esquema de replicación pasiva.

3.5.2. Replicación de Máquina de Estados

La replicación Activa [100] o de Máquina de Estados (State Machine) es otra variante que se puede utilizar para realizar replicación. En ésta todas las réplicas están activas, es decir se comunican con los clientes mediante IPC y entre sí utilizando un GCS. El protocolo de difusión requerido debe ser tolerante a fallos con entrega Atómica o de Orden Global. Si se produce un fallo en alguna de las réplicas, simplemente el grupo lo detecta y reduce su número de miembros. Los clientes que se comunicaban con la réplica fallida deberán ahora comunicarse con alguna otra réplica que les brinde servicios. El *DVK* dispone de la posibilidad de designar múltiples réplicas con el mismo *endpoint* (tipo Réplica). Esto permite que los clientes ejecutando en el mismo nodo que la réplica utilizarán los servicios de ésta mientras esté activa. Con esto se logra una distribución de las cargas entre los clientes y los servidores.

Para aquellos clientes que no residen en el mismo nodo que una réplica, ante un fallo de la réplica que los servía, el VOS podrá cambiar el parámetro que identifica al nodo del *endpoint* Remoto que asignaba la réplica fallida a una nueva réplica residente en otro nodo. La *Transparencia de Fallos* que brinda el *DVK* cuando se utiliza replicación Activa facilita el desarrollo de aplicaciones con alta disponibilidad.

3.5.3. Migración de Procesos

La migración de procesos es una técnica que se utiliza para balancear la carga entre los diferentes nodos de un cluster, y para la consolidación de servicios de tal forma de reducir la cantidad de nodos activos, lo que redundaría en un ahorro energético.

El *DVK* debe contemplar la posibilidad de que un proceso asociado a un *endpoint* pueda ser migrado desde un nodo origen a un nodo destino, sin que por ello se deban modificar las aplicaciones (ni el proceso migrado ni los procesos que se comunican con él), a lo que se denomina *Transparencia de Migración*.

Antes de la migración, el VOS debe informar al *DVK* la intención de migrar a un proceso (identificado por su *endpoint*). Durante la migración, las comunicaciones entre el proceso a migrar y otros procesos se suspenden. Si durante la migración se produce un fallo y el proceso finalmente no se migra, el VOS informa al *DVK* que debe retrotraer (Rollback) la migración y las comunicaciones se reestablecen entre el proceso fallidamente migrado y el resto de los procesos. En caso de que la migración resulte exitosa, el VOS informa al *DVK* que debe dar por Confirmada (Commit) la

migración, reestableciendo las comunicaciones entre el proceso migrado (ejecutando en su nodo destino) y el resto de los procesos.

3.5.4. Seguridad y Protección

Si bien la seguridad y la protección son aspectos que fueron considerados en el diseño, las tareas de investigación no se focalizaron en estos temas ya que muchos están implícitos en el propio concepto de virtualización. De todos modos, se utilizaron para el modelo principios básicos de seguridad para el diseño y desarrollo de software tales como la separación de responsabilidades, el otorgamiento de privilegios mínimos para desarrollar las diferentes tareas y operaciones. Del modelo de arquitectura propuesto se desprende que la seguridad del *DVS* depende en gran medida de la seguridad del OS-host y de la implementación del *DVK*.

Los servicios de Autenticación, Autorización, Registros (logging) y control de acceso a los diferentes recursos que pueden ser utilizados por los VOS deben implementarse como componentes del *DVK*, o éste debe actuar como intermediario para el control de acceso hacia los servicios del OS-host.

Para el desarrollo del prototipo del *DVS*, se utilizaron facilidades de seguridad básicas provistas por el OS (Linux), se usaron algoritmos de cifrado para las comunicaciones entre *proxies* (de uso opcional) y se implementó el control de privilegios (*capabilities*) en cada invocación a las APIs del *DVK*.

3.5.5. Planificación Distribuida

El modelo de *DVK* no propone ningún mecanismo de planificación particular. La planificación de corto plazo (en cada uno de los nodos del cluster) la lleva a cabo el OS-host. Cada proceso de cada *DC* tendrá asignada una prioridad de ejecución y de despacho establecida por su VOS. Cada VOS dispondrá de un conjunto recursos de acuerdo a la configuración del *DC*, y los recursos asignados al *DC* están conformados por el conjunto de recursos de cada uno de los Contenedores de cada nodo.

El *DVK* puede ofrecer servicios de monitoreo de parámetros de rendimiento del cluster de virtualización, tales como la utilización de cada nodo (CPU, RAM, disco, red, etc.) y de los enlaces entre nodos. Los datos obtenidos por estos mecanismos pueden ser utilizados por los diferentes VOS,

por ejemplo, para aplicar sus políticas de planificación distribuida, de distribución de cargas o de migración de procesos.

4. PROTOTIPO DE DVS

Prácticamente, en forma simultánea al diseño del modelo de arquitectura de *DVS*, se comenzó con el desarrollo de un prototipo con el fin de confirmar la factibilidad de la propuesta, detectar errores de diseño y proponer mejoras.

Como consecuencia de las limitaciones de tiempos y de recursos, uno de los principios para su desarrollo fue “*utilizar, en lo posible, todo el software previamente desarrollado sea propio o de disponibilidad pública*”. Por esta razón, se eligió al OS Linux como plataforma para el prototipo, por la gran disponibilidad de software que podía complementar el desarrollo, la disponibilidad de Contenedores en su kernel y la gran comunidad de soporte.

En el año 2011, el autor desarrolló MoL [55] del cual surgieron las primeras ideas del *DVS*. MoL no tenía un rendimiento atractivo porque básicamente utilizaba los mecanismos de IPC de Linux (sockets). Para mejorar su rendimiento se propuso incorporar como un co-kernel los mecanismos de IPC del microkernel de Minix 3, denominándolo M3-IPC. M3-IPC en principio permitía la transferencia de mensajes y datos entre procesos dentro del mismo computador, luego fue extendido con la capacidad de transferir mensajes y datos, en forma transparente, a procesos de otros computadores de un cluster.

El prototipo desarrollado comprende los siguientes componentes:

-*DVK*: En él se implementan, por un lado, los mecanismos de IPC (y sus APIs) similares a los de Minix 3, pero extendidos para transferir mensajes y datos entre nodos de un cluster, para soportar migración y esquemas de replicación. Por otro lado, se implementaron los mecanismos de gestión del *DVS* tales como arrancar y detener el *DVS*, configurar, arrancar y detener *DCs*, registrar y de-registrar nodos y procesos *proxies*, registrar y de-registrar procesos ordinarios dentro de un *DC*, obtener información de estado y estadística de cada componente del *DVS*.

-
- *GCS*: Para desarrollar las aplicaciones y los VOSs que implementarían replicación de ciertos componentes se requería disponer de una herramienta tal como un mecanismo de consenso distribuido o un sistema de comunicaciones grupales. Se analizaron varias alternativas, tales como Paxos [86], ISIS [83], HORUS [107], ISIS2 [108], Corosync/Heartbeat [109], etc., y se decidió utilizar el Spread Toolkit [110, 111] por su facilidad de uso, abundante documentación, buen soporte, programas fuentes de varias implementaciones y porque disponía de todas las APIs requeridas para los desarrollos proyectados. En el *DVK* no se incorporan comunicaciones grupales que utilizan las APIs de Spread Toolkit, aunque en el modelo propuesto se sugiere hacerlo. De todos modos, el *DVK* actuaría como una pasarela o intermediario que controlaría los accesos de los procesos de un *DC* a los recursos y APIs del Spread Toolkit.
 - *VOS-multiservidor*: Como se disponía de la base del desarrollo de MoL, se convirtieron sus componentes en los componentes de un VOS-multikernel, los cuales podrían ejecutarse distribuidos y eventualmente, replicados en diferentes nodos del cluster.
 - *VOS-unikernel*: Como base para el desarrollo de un VOS-unikernel se tomó el software LwIP [134], el cual dispone de una pila de protocolo TCP/IP completa, un servidor web básico con página estática en RAM, e interfaces de LAN virtual para utilizar las vNICs que se disponen en el kernel de Linux denominada TAP (Terminal Access Point).
 - *MoL-FS*: MoL-FS es el servidor de sistema de archivos de Minix 3 cuyo código fuente fue portado a MoL. Es utilizado tanto como componente del VOS-multiserver, como componente opcional del VOS-unikernel o accesible directamente desde aplicaciones ordinarias Linux. Se incluyó para esto último un módulo FUSE (FileSystem in USErspace) de modo que dichas aplicaciones pudieran utilizarlo de forma autónoma a través del mecanismo de montaje. También se desarrolló MoL-FAT que es otro servidor de sistemas de archivos de tipo FAT el cual utiliza las funciones disponibles en *FatFs*, una biblioteca que permite acceder a dispositivos con este formato de sistema de archivos.
 - *MoL-VDD*: Es un Gestor de Disco Virtual (VDD: Virtual Disk Driver), es decir acepta operaciones de lectura y escritura en bloques de disco. MoL-VDD acepta diferentes tipos de dispositivos tales como archivos de dispositivos, archivos ordinarios, archivos

remotos, etc. Además, admite la configuración en redundancia utilizando el esquema Primario/Respaldo, comportándose en forma equivalente a un RAID-1 distribuido.

-*MoL-INET*: Es el servidor de la pila de protocolos TCP/IP de Minix 3 cuyo código fuente fue portado a MoL.

-*MoL-ETH*: Es una tarea de interface virtual ethernet de Minix 3 cuyo código fuente fue portado a MoL, pero que no utiliza hardware real sino TAP como virtual NIC.

-*Nweb*: Es un servidor web muy básico que utiliza páginas web en RAM, el cual fue modificado para acceder a páginas web residentes en archivos.

- *Módulo Webmin*: Si bien el prototipo puede ser gestionado mediante Interface de Línea de Comandos (CLI: Command Line Interface), o utilizando las APIs del *DVMS* que forman parte del *DVK*, se desarrolló un módulo *Webmin* que permite gestionar el DVS mediante una interface web (la versión actual solo permite visualizar el estado, la configuración y las estadísticas de los componentes).

-*Otras utilidades*: En base a *Nweb* (servidor web) y a *urlget* (cliente web de línea de comandos) se crearon un servidor web y un cliente web que utilizan protocolo M3-IPC en lugar de HTTP. El objetivo era demostrar cómo se pueden convertir programas de red que utilizan sockets usando las APIs de M3-IPC en su reemplazo. También, se desarrolló un proxy entre los protocolos HTTP y M3-IPC de tal forma que un cliente M3-IPC pueda acceder a un server HTTP a través de dicho proxy. En este caso, el objetivo era demostrar que, si no es posible modificar el código del servidor web, se puede utilizar un proxy que actúa como pasarela entre protocolos.

El estado actual de desarrollo del prototipo contiene suficientes componentes como para considerarlo una prueba de concepto. Aun así, el desarrollo sobre el mismo continúa tanto en su mejora como en la incorporación de nuevas facilidades. El repositorio del prototipo se encuentra en <https://github.com/PabloPessolani/DVS>.

4.1. Núcleo de Virtualización Distribuida (DVK)

El *DVK* se desarrolló como un módulo del kernel de Linux, complementado por bibliotecas para enlazar con los programas que constituyen las aplicaciones y los VOSs que utilizaría sus

servicios. Si bien el GCS no está integrado al *DVK* en la versión actual del prototipo, se incluye su descripción en esta sección porque el modelo de *DVS* lo considera como parte de él.

4.1.1. Módulo para el Kernel de Linux

Para contemplar el *DVS* con todas sus componentes (Nodos, Contenedores, *DCs*, Procesos, *proxies*, etc.) se desarrolló un módulo del kernel de Linux que se carga mediante el comando *insmod*.

La versión de kernel de Linux utilizada para desarrollar el módulo es la 2.6.32 para Intel arquitectura x86. Previo al proceso de compilación del módulo se requiere aplicar unos parches (patch) al código fuente que preparan el kernel para luego enlazar con el módulo. Generalmente la aplicación de parches suele ser un problema a medida que cambian las versiones, pero en este caso se tuvo que realizar por la forma elegida para implementar las Llamadas al Sistema del *DVS*.

Es conveniente aclarar la terminología utilizada en Linux, en Minix (base de MoL) y en el *DVK* referida a procesos y tareas resulta confusa. En el kernel de Linux, a los procesos e hilos de modo usuario de los denomina tareas y se los define por estructura de datos *struct task_struct* del archivo */include/linux/sched.h*, pero por otro lado la información de esas tareas se puede encontrar en el pseudo sistema de archivos */procs*, lo que da una idea de que son procesos.

En Minix, todas las entidades ejecutables (a las que se les asigna un *endpoint*) se las denomina procesos, pero luego se las clasifica según la capa del sistema donde se ejecutan y los privilegios que tienen. Las Tareas son procesos que se ejecutan con privilegios que le permiten gestionar de dispositivos (en forma indirecta a través de Task Calls). Los Servidores son procesos que ofrecen a los procesos de usuarios servicios (System Calls) tales como Sistema de Archivos, Gestión de Memoria, Información del Sistema, Protocolos de Red, etc. En el *DVK* se adopta una terminología similar a la de Minix 3, en donde los procesos de usuarios, tareas, servidores y proxies son todos procesos a los que se los distingue por sus funciones y privilegios.

Habitualmente, para la arquitectura Intel x86, Linux establece la dirección del punto de entrada para sus Llamadas al Sistema en el descriptor 0x80 en hexadecimal (128 decimal) de la tabla de descriptores de interrupciones. Las versiones más actuales del kernel de Linux utilizan la instrucción de máquina *syscall*. Una forma de implementar los servicios del *DVS* sería incluirlos como nuevas Llamadas al Sistema de Linux. Esta opción fue descartada previendo que actualizaciones de Linux futuras puedan utilizar los números de Llamadas al Sistema elegidos para el *DVS*.

Con el objetivo de evitar estos problemas a futuro se instaló un descriptor propio para dar tratamiento a las Llamadas al Sistema del *DVS*, razón por la cual se requiere del parche (*patch*) al código fuente de Linux. De esta forma, las modificaciones permanentes que se hicieron y que se hagan sobre el código del *DVK* no deberían afectar el resto del sistema. Además, tal como se planteó en el modelo, el *DVK* se puede considerar como un co-kernel, es decir el kernel de otro OS compartiendo recursos con el kernel de Linux. El *DVK* toma como base a los mecanismos de IPC de Minix, pero mejorándolo y extendiéndolo a comunicaciones con procesos de otros nodos, soportando migraciones de procesos, replicación activa, replicación pasiva e implementando el aislamiento requerido entre *DCs*. Además, se implementan en él todas las operaciones relacionadas con la gestión de las abstracciones de un *DVS*.

Otra opción hubiese sido utilizar la Llamada al Sistema *ioctl()* de tal forma de considerar al *DVS* como un dispositivo, pero esto forzaría a que cada invocación al *DVK* deba atravesar previamente parte del gestor del sistema de archivos de Linux y eso provocaría una reducción del rendimiento, particularmente en IPC. Esta opción no se descarta, sino que merece un análisis más profundo dado que se reduciría el número de líneas de código fuente del parche del *DVK* a aplicar al kernel de Linux y por lo tanto es menos intrusivo y más fácilmente actualizable.

A continuación, se describirán algunas de las estructuras de datos utilizadas para la implementación del prototipo de *DVS*. Se presentan aquí solamente aquellos campos que permiten describir el funcionamiento y no se incluyen campos complementarios requeridos para la programación. Es importante aclarar que estas estructuras tienen carácter *local*, es decir que la misma estructura que describe al mismo componente del *DVS* en nodos diferentes puede contener valores diferentes en algunos de sus campos.

La abstracción básica del *DVK* es el *DVS*, en el cual se especifican un conjunto de parámetros requeridos para su funcionamiento y configuración. La estructura de datos que lo describe es *dvs_usr*.

```

struct dvs_usr {
    int d_nr_dcs;           // Número máximo de DCs en el DVS
    int d_nr_nodes;       // Número máximo de nodos en el DVS
    int d_nr_procs;       // Número máximo de procesos en un DC
    int d_nr_tasks;       // Número máximo de tareas en un DC
    int d_nr_sysprocs;    // Número máximo de procesos privilegiadas en un DC
    int d_max_copybuf;    // Número máximo de bytes de datos a transferir
                        // en una copia simple
    int d_max_copylen;    // Número máximo de bytes de datos a transferir
                        // en una copia compuesta (N * d_max_copybuf)
    unsigned long int d_dbglvl; // conjunto de banderas para el nivel de debug
    int d_version;        // version del DVK (inmutable)
    int d_subver;        // sub-version del DVK (inmutable)
};

```

En el campo *d_nr_procs*, se especifica el número máximo de procesos que puede contener un *DC*. Cada proceso tiene un número de proceso *p_nr*, que lo identifica y que tiene un valor ($p_nr \geq 0$). El campo *d_nr_sysprocs* describe cuántos de esos procesos serán *Procesos de Sistema* que adquieren privilegios adicionales a los de los procesos ordinarios. Es decir, los *p_nr* que estarán disponibles para utilizar son ($0 \leq p_nr < d_nr_procs$), de los cuales los de valor ($0 \leq p_nr < d_nr_sysprocs$) serán *Procesos de Sistema*. Los *Procesos de Sistema* son utilizados por el VOS para realizar operaciones privilegiadas utilizando los servicios del *DVK*, y es el VOS el responsable de la administración de los *p_nr* y *endpoints*. Adicionalmente, en cada *DC* se puede disponer de hasta *d_nr_tasks* procesos de tipo Tareas con ($p_nr < 0$) dedicadas al sistema, es decir con privilegios adicionales. Un VOS puede asignar ($(-2) \geq p_nr \geq (-d_nr_tasks)$), dado que el $p_nr=(-1)$ tiene un uso especial.

Por lo tanto, entre tareas y procesos (que incluye a los procesos servidores y procesos de usuario), la cantidad máxima de *p_nr* que podrá contener un *DC* será ($d_nr_tasks + d_nr_procs$). Esta forma de organizar la asignación *p_nr* proviene de Minix 3. A diferencia de los PIDs que no son reutilizables, los *p_nr* pueden volver a utilizarse una vez que el proceso que lo tenía previamente asignado finalizó.

Respecto a *d_max_copybuf* (número máximo de bytes de datos a transferir en una copia simple), habilita al administrador del *DVS* a configurar su valor de acuerdo al ancho de banda de las LANs utilizada en el cluster. Si el valor de *d_max_copybuf* es muy elevado, los *proxies* de los nodos origen y destino de una copia de datos estarían ocupados durante demasiado tiempo enviando esos datos y retrasando el envío de otros mensajes. Para reducir la latencia en las copia de datos, se configura también *d_max_copylen* (número máximo de bytes de datos a transferir de datos en una copia múltiple ($N * d_max_copybuf$)), que es el tamaño máximo que puede tener una copia de datos entre procesos en el *DVS*. De acuerdo al tamaño de la copia pueden realizarse una o múltiples copias parciales de tamaño *d_max_copybuf* hasta completar el tamaño solicitado. Al dividir el proceso de copia, los *proxies* de comunicaciones se liberan al final de cada copia parcial permitiendo que otras transferencias de mensajes o de datos utilicen sus servicios y reduciendo para ellos la latencia por la contención de la red.

Otra estructura de datos fundamental para el *DVS* es el *node_usr*, que permite describir a cada uno de los nodos que conforman el *DVS*.

```
struct node_usr {
int n_nodeid;           // ID del nodo; 0 ≤ n_nodeid < d_nr_nodes
volatile unsigned long n_flags; // banderas que indican el estado del nodo
int n_proxies;         // ID del proxy que se utiliza para el nodo
unsigned long int n_dcs; // mapa de bits de DCs utilizando el nodo
struct timespec n_timestamp; // estampilla de tiempo del último mensaje enviado
```



```

struct timespec n_rtimestamp; // estampilla de tiempo del último mensaje recibido
unsigned long n_pxsent; // cantidad de mensajes enviados al nodo
unsigned long n_pxrcvd; // cantidad de mensajes recibidos del nodo
char n_name[MAXNODENAME]; // nombre del nodo
};

```

Cada nodo tiene un identificador *n_nodeid* y un nombre (*n_name*) para una mejor descripción. El campo *n_proxies* es el número que identifica a los procesos *proxies* que se utilizarán para realizar las comunicaciones entre el nodo local y el nodo descrito por la estructura. El campo *n_dcs*, es un mapa de bits (bitmap) de los *DCs* que están utilizando ese nodo. Actualmente, se utiliza un entero sin signo lo que limita el valor de *n_dcs* a 32 *DCs*. Esto solo se hizo para facilitar la impresión del contenido del bitmap durante el debugging y en el archivo */proc/dvs/nodes*, pero es fácilmente modificable por un vector de enteros.

Para describir un *DC* se utiliza la estructura de datos *DC_usr*.

```

DC_usr {
int dc_dcid; // ID del DC. 0 ≤ dc_dcid < d_nr_dcs
int dc_flags; // Banderas que describen el estado del DC
int dc_nr_procs; // Número máximo de procesos en este DC (≤ d_nr_procs)
int dc_nr_tasks; // Número máximo de tareas en este DC (≤ d_nr_tasks)
int dc_nr_sysprocs; // Número máximo de procesos privilegiadas en este DC
int dc_nr_nodes; // cantidad de nodos a los que se puede expandir el DC
unsigned long int dc_nodes; // mapa de bits de nodos que puede abarcar el DC
int dc_warn2proc; // Endpoint del proceso a informar cuando otro proceso
// ejecuta un exit() implícito.
int dc_warnmsg; // Tipo de mensaje que se le envía al dc_warn2proc
char dc_name[MAXVMNAME]; // Nombre del DC
cpu_set_t dc_cpumask; // Máscara de las CPU de este host para este DC
};

```

Cada *DC* tiene un identificador *dc_dcid* y un nombre (*dc_name*) para una mejor descripción. Como se puede ver en la estructura, también aquí se pueden especificar las cantidades máximas de procesos, tareas y procesos del sistema, pero para cada *DC*. Esos valores máximos estarán siempre limitados a los valores máximos especificados para el *DVS*. También, en esta estructura se especifican la cantidad máxima de nodos a los que se puede expandir el *DC* y un mapa de bits que indica cuáles nodos están siendo utilizados por el *DC*. Actualmente, para *dc_nodes* se utiliza un entero sin signo lo que limita el valor a 32 nodos, pero es fácilmente modificable.

Si bien se mencionó que cada estructura de datos tiene carácter local, algunas de ellas deben contener valores idénticos (replicados) en diferentes nodos, como por ejemplo *dc_nodes*, que es el mapa de bits que especifica cuáles nodos puede abarcar un *DC*. La configuración de estos parámetros en cada uno de los nodos es responsabilidad del administrador o de una aplicación de gestión del *DVS* que así lo configure. Actualmente, esto se lo realiza utilizando scripts, pero a futuro se planea desarrollar una aplicación de gestión distribuida que lo realice utilizando archivos de configuración.

La estructura que describe a los procesos (registrados en el *DVS*) que se ejecutan dentro de un *DC* y a los procesos *proxies* se denomina *proc_usr*.

```

struct proc_usr {
    int p_nr;                // Número de proceso
    int p_endpoint;         // Número de Endpoint
    int p_dcid;             // DC al que pertenece el proceso
    volatile unsigned long p_rts_flags; // banderas de estado
    int p_lpid;             // PID de LINUX si es un proceso Local
    int p_nodeid;          // ID del nodo donde se encuentra ejecutando
                                // el proceso PRIMARIO y el proceso local
                                // es tipo RESPALDO (BACKUP)
    unsigned long p_nodemap; // bitmap de los nodos donde se están
                                // ejecutando otras réplicas
    volatile unsigned long p_misc_flags; // Banderas que definen el tipo de proceso
    char p_name[MAXPROCNAME]; // nombre del proceso
    cpu_set_t p_cpumask;      // Máscara de las CPU de este proceso en este
                                // host
};

```

El campo *p_nr* es el número de proceso y como se indicó anteriormente puede tener valores que van desde $((-d_nr_tasks) \leq p_nr < d_nr_procs)$, donde $p_nr \neq (-1)$ para los procesos ordinarios y $p_nr = (-1)$ para los procesos *proxies*. Existe una tabla o vector con todos los procesos ordinarios de un *DC*, el *p_nr* es el índice de esa tabla.

El campo *p_endpoint* indica cual es el *endpoint* del proceso. El valor de un *endpoint* se calcula en función del *p_nr* y la *generación* de ese *p_nr*, ambos valores son gestionados por el VOS. Se podría utilizar ($p_endpoint = p_nr$), sin considerar la *generación*. La *generación* se utiliza para distinguir a un proceso **A** que tiene $p_nr = X$ y que ha finalizado, de otro nuevo proceso **B** al que se le ha asignado el mismo $p_nr = X$. Ambos procesos ocuparon la misma entrada en la tabla de procesos del *DC*, pero son diferentes porque tienen diferentes *endpoints* (son de generaciones distintas). El campo *p_misc_flags* son banderas que indican de qué tipo de descriptor de proceso se trata.

```

MIS_PROXY           // El proceso es un Proxy Emisor o Receptor
MIS_CONNECTED      // El proxy está conectado
MIS_NEEDMIG        // Se ha solicitado una migración de este proceso.
MIS_RMTBACKUP      // El proceso es de tipo Respaldo (BACKUP)
MIS_KTHREAD        // No es proceso ordinario de usuario, es un hilo de Kernel
MIS_REPLICATED     // Es un proceso Réplica

```

En cuanto a los *proxies*, los mismos se describen con las siguientes estructuras de datos.

```

struct proxies_s {
    proxies_usr_t px_usr; // Otra informacion de proxies
    struct proc px_sproxy; // Proceso proxy EMISOR
    struct proc px_rproxy; // Proceso proxy RECEPTOR
};

```

Como se puede ver, la estructura *proxies_s* contiene dos descriptores de procesos, uno para el proxy Emisor y otra para el proxy Receptor. El campo *px_usr*, es de tipo *proxies_usr_t* que se corresponde con una estructura *proxies_usr_s* con información de interés para el administrador.

```

struct proxies_usr_s {
unsigned int px_id;           // ID de los Proxies
unsigned long int px_flags; // Banderas de estado
int px_max_copybuf;         // Número máximo de bytes que los proxies soportan copiar
char px_name[MAXPROXYNAME]; // Nombre de los Proxies
};
typedef struct proxies_usr_s proxies_usr_t;

```

En la estructura de datos *proxies_usr_s*, el campo *px_id* es el identificador de los *proxies* y el campo *px_name* permite asignarle un nombre para una mejor identificación. El campo *px_flags* son una serie de banderas que describen el estado de los *proxies*. El campo *px_max_copybuf* indica la cantidad máxima de bytes que los *proxies* soportan copiar/transferir, que por supuesto, debe ser menor o igual al especificado en el campo *d_max_copybuf* del *DVS*.

4.1.2. APIs de Gestión del DVS

El *DVK* incluye las funciones de gestión del *DVMS* suministrando las siguientes APIs:

rcode = mnx_dvs_init(nodeid, dvs_usr);	Esta función habilita el modulo local del <i>DVK</i> para conformar el <i>DVS</i> . El parámetro <i>nodeid</i> , es el identificador del nodo local y <i>dvs_usr</i> es la dirección de una estructura de datos donde se asignan los valores al <i>DVS</i> .
rcode = mnx_dvs_end();	Esta función detiene el funcionamiento del <i>DVS</i> en el nodo local, finalizando todos los procesos y <i>DCs</i> que se estuviesen ejecutando en el nodo local y liberando los recursos que este hubiese obtenido durante su funcionamiento.
rcode = mnx_getdvsinfo(dvs_usr);	Esta función permite obtener los valores de configuración del <i>DVS</i> en el nodo Local en una estructura <i>dvs_usr</i> .

Antes de poder iniciar cualquier operación sobre alguno de los componentes del *DVS* (*DCs*, Procesos, *proxies*, etc.) se debe habilitar el *DVS* en el módulo del *DVK* del nodo local. Para ello, se utiliza la función *mnx_dvs_init()*, donde se le pasan como parámetros el ID del nodo local y un puntero a una estructura *dvs_usr* donde se deben especificar los valores de los distintos campos (mutables) que la componen.

Lo recomendable es, antes de finalizar un *DVS*, finalizar previamente todos los procesos, todos los *proxies* y todos los *DCs*. Invocar a la función *mnx_dvs_end()* sin hacer las operaciones recomendadas, provoca la finalización en forma abrupta tanto a procesos ordinarios registrados como de *proxies* y *DCs* que estuviesen ejecutando en el *DVS* del nodo local.

La función *mnx_getdvsinfo()* permite obtener en una estructura *dvs_usr* los parámetros de configuración actual en el nodo local de un *DVS* previamente iniciado.

4.1.3. Conformación del cluster del DVS (nodos unidos por proxies)

Para conformar un cluster de *DVS* deben describirse los nodos que lo componen y los *proxies* de comunicaciones que permiten las transferencias de mensajes y datos entre ellos. En cada uno de los módulos *DVKs* de cada nodo componente del *DVS*, deben describirse el resto de los nodos y los *proxies* que el nodo local utilizará para comunicarse con cada uno de ellos.

El *DVK* ofrece las siguientes APIs para gestión de *proxies*:

rcode = mnx_proxies_bind(name, px_id, spid, rpid, maxcopybuf);	Esta función permite registrar un par de procesos que serán el proxy Emisor y proxy Receptor. Como parámetros se indican: el nombre del par de <i>proxies</i> (<i>name</i>), el número que identifica a los <i>proxies</i> (<i>px_id</i>), el PID del Emisor (<i>spid</i>), el PID del Receptor (<i>rpid</i>) y número máximo de bytes que los <i>proxies</i> soportan copiar (<i>maxcopybuf</i>).
rcode = mnx_proxies_unbind(px_id);	Esta función permite de-registrar a un par de <i>proxies</i> especificando su ID (<i>px_id</i>)
rcode = mnx_proxy_conn(px_id, status);	Esta función permite especificar el estado (<i>status</i>) de cada proxy en lo referente a la conexión.
rcode = mnx_getproxyinfo(px_id, sproc_usr, rproc_usr);	Permite obtener información de estado y estadística de los procesos <i>proxies</i> .
rcode = mnx_get2rmt(header, payload); rcode = mnx_get2rmt_T(header, payload, to_ms);	Estas funciones son utilizadas por el proxy Emisor. Si lo que se recibe del módulo de <i>DVK</i> local es un mensaje, entonces el mensaje estará dentro de la estructura de datos apuntada por <i>header</i> , y el buffer apuntado por <i>payload</i> no se utiliza. Si lo que se recibe es un bloque de datos, entonces en la estructura apuntada por <i>header</i> se especifica la longitud de los datos, y en el buffer apuntado por <i>payload</i> , los datos a copiar. Si no hay mensajes para enviar, el proxy Emisor se bloquea hasta transcurridos 30 segundos, luego de lo cual retorna un error <i>EDVSAGAIN</i> que indica al programador que debe volver a intentar. Se puede cambiar el período de espera utilizando el parámetro <i>to_ms</i> en milisegundos.
rcode = mnx_put2lcl(header, payload);	Esta función la utiliza el proxy Receptor para enviarle al módulo de <i>DVK</i> local los mensajes y datos provenientes del nodo remoto. Para ellos, la estructura de datos <i>header</i> debe estar correctamente completada, y si fuesen datos los transferidos, los mismos deberán estar apuntados por <i>payload</i> .

Una vez en ejecución los procesos *proxies*, deben registrarse como Emisor y Receptor mediante *mnx_proxies_bind()*. Esto aún no habilita a los *proxies* a intercambiar mensajes y datos con el nodo remoto especificado.

Una vez registrados, cada proxy del nodo local (Emisor/Receptor) establece la comunicación con el proxy remoto (Receptor/Emisor). Cada uno de los *proxies* en forma individual verifica la

conexión con el nodo remoto y si considera que la misma se estableció correctamente, se lo informa al *DVK* usando *mnx_proxy_conn()*, indicando el estado de la conexión (*CONNECT_SPROXY*, *CONNECT_RPROXY*) correspondiente. Inmediatamente después, el proxy Emisor ya se encuentra en condiciones de obtener mensajes y datos del *DVK* para ser enviados hacia el nodo remoto. De igual forma, el proxy Receptor se encuentra en condiciones de entregarle mensajes y datos provenientes del nodo remoto especificado.

El proxy Emisor utiliza *mnx_get2rmt()* para obtener los mensajes y datos del módulo del *DVK* local para luego enviarlos al nodo remoto. El proxy Receptor utiliza *mnx_put2lcl()* para enviarle al módulo del *DVK* local los mensajes y datos recibidos del nodo remoto.

En caso de un fallo en las comunicaciones de alguno de los *proxies*, se le debe comunicar al módulo de *DVK* local respecto a esta situación utilizando *mnx_proxy_conn()* especificando la situación en el parámetro *status* (*DISCONNECT_SPROXY*, *DISCONNECT_RPROXY*). Para finalizar los *proxies*, se utiliza *mnx_proxies_unbind()*.

El modelo de *DVK* no establece ninguna condición particular respecto al protocolo que debe utilizarse para los *proxies*, de hecho, el prototipo cuenta con una amplia variedad como se muestra en la Tabla VIII:

TABLA VIII. PROXIES DE COMUNICACIONES DEL PROTOTIPO DE DVS

Protocolo	Procesos/Hilos	Nivel	Compresión
TCP	Procesos	Usuario	NO
UDP	Procesos	Usuario	NO
TIPC	Procesos	Usuario	NO
TCP	LwIP/Hilos	Usuario	NO
TIPC	Procesos	Usuario	LZ4
UDT	Procesos	Usuario	NO
TCP	Hilos	Usuario	NO
RAW Ethernet	Procesos/Hilos	Usuario	NO
TCP	Hilos	Kernel	NO

Para transferir un mensaje (estructura *message*) o un bloque de datos desde un nodo a otro se requiere transferir una estructura tipo *cmd_s*. En ella se especifican el tipo de comando o encabezado que transporta, el DC destino, el *endpoint* origen, el *endpoint* destino, y algunos campos que pueden ser utilizados por los *proxies* para implementar su propio protocolo de transferencia (como se hizo en el caso de los *proxies* RAW Ethernet).

```

struct cmd_s {
int c_cmd;      // tipo comando que transporta
int c_dcid;    // ID del DC
int c_src;     // Endpoint Origen
int c_dst;     // Endpoint Destino
int c_snode;   // Nodo Origen

```

```

int c_dnode;    // Nodo Destino
int c_rcode;    // Código de retorno
int c_len;      // Tamaño en bytes de la carga trasportada (para copias de datos)
unsigned long c_flags; // Banderas genéricas a utilizar por los proxies
unsigned long c_snd_seq; // nro de secuencia de envío a utilizar por proxies
unsigned long c_ack_seq; // nro de secuencia de reconcimiento a utilizar por proxies
struct timespec c_timestamp; // Estampilla de tiempo
union {
    vcopy_t cu_vcopy; // estructura de datos requerida para una copia remota
    message cu_msg; // mensaje transportado
}c_u;
};

```

Para el caso de transportar bloques de datos, se especifican su tamaño, dirección virtual en el proceso origen y dirección virtual en el proceso destino.

Transferir paquetes pequeños uno a uno desde un nodo a otro puede resultar ineficiente desde el punto de vista de las comunicaciones. A los efectos de mejorar la eficiencia, se puede transferir un lote de mensajes con una sola transferencia. Si bien no está implementado aún, los *proxies* utilizando lotes de mensajes se encuentran en proceso de desarrollo. De igual forma, tanto los datos como los metadatos pueden cifrarse, pero aún no se desarrollaron *proxies* de ese tipo.

Los *proxies* pueden utilizar cualquier protocolo, pueden implementarse en modo usuario o en modo kernel, puede utilizarse compresión, puede utilizarse cifrado, pueden utilizarse los mismos para *proxies* para intercambiar datos con todos los nodos del cluster o puede utilizarse *proxies* individuales por cada nodo o cualquier combinación. Esto da mucha flexibilidad para conformar el cluster de *DVS*, dejando a la decisión del administrador que *proxies* utilizar y de qué forma.

El *DVK* ofrece las siguientes APIs para gestión de Nodos:

rcode= mnx_node_up(node_name,node_id, px_id);	Esta función permite registrar a un nodo remoto en el módulo del <i>DVK</i> especificando su nombre (<i>node_name</i>), su identificador ($0 \leq \textit{node_id} < d_nr_nodes$) y el ID del par de <i>proxies</i> (Emisor y Receptor) que conectarán al nodo local con el otro nodo.
rcode = mnx_node_down(node_id);	Esta función permite de-registrar a un nodo remoto del módulo del <i>DVK</i> especificando su <i>node_id</i> .
rcode = mnx_getnodeinfo(node_id, node_usr);	Esta función permite obtener en una estructura de datos tipo <i>node_usr</i> , la información del nodo especificando su <i>node_id</i> .

El *DVK* tiene su propio protocolo para transferir mensajes y copiar datos entre procesos de un mismo *DC*. Este protocolo consiste en un comando o encabezado (*header*) con todos los parámetros requeridos (estructura *cmd_s*) y eventualmente una carga de datos (*payload*). Estos componentes son encapsulados por el protocolo que los *proxies* utilizan para transportarlos hacia otros nodos.

En la Fig. 29 se presentan diagramas del protocolo para diferentes comandos utilizados para realizar las operaciones de transferencias que a continuación se describen.

Cuando un proceso del nodo Local envía un mensaje con *mnx_send()* a un proceso remoto (Fig. 29-A), se genera en el *DVK* local un comando *CMD_SEND_MSG*, mientras tanto el proceso Emisor queda bloqueado. El comando *CMD_SEND_MSG* es extraído por el Proxy Emisor Local utilizando *mnx_get2rmt()* y enviado al Proxy Receptor Remoto. Este inserta el comando en el *DVK* remoto utilizando *mnx_put2lcl()*. El *DVK* remoto intenta entregar el mensaje al destinatario. Si el destinatario estaba esperando por el mensaje, entonces el *DVK* remoto genera un mensaje de tipo *CMD_SEND_ACK*, para indicar que el receptor recibió el mensaje y lo inserta en la cola de comandos del Proxy Emisor Remoto. Este lo extrae usando *mnx_get2rmt()* y lo envía al nodo Local. Una vez recibido por el Proxy Receptor Local, ejecuta *mnx_put2lcl()* desbloqueando al Emisor del mensaje. Si el mensaje no pudo entregarse por alguna otra razón, también se desbloquea al Emisor indicando el código del error ocurrido.

Cuando un proceso del nodo Local envía un mensaje de notificación con *mnx_notify()* a un proceso remoto (Fig. 29-B), se genera en el *DVK* un comando *CMD_NTFY_MSG*, mientras tanto el proceso Emisor queda bloqueado hasta que el Proxy Emisor Local haya extraído el comando (*mnx_get2rmt()*) para enviarlo al nodo destino. Por lo descripto, no hay garantía de entrega para este tipo de mensajes a nivel de *DVK*.

Cuando un proceso del nodo Local envía un mensaje con *mnx_sendrec()* a un proceso remoto (Fig. 29-C) y luego espera un mensaje de respuesta, se genera en el *DVK* un comando *CMD_SNDREC_MSG*. Mientras tanto, el proceso Emisor queda bloqueado. El comando *CMD_SNDREC_MSG* es extraído por el Proxy Emisor Local (*mnx_get2rmt()*) y enviado al Proxy Receptor Remoto, quien inserta el mensaje en el *DVK* remoto (*mnx_put2lcl()*). El *DVK* remoto intenta entregar el mensaje al destinatario, si estaba esperando por el mensaje lo entrega y desbloquea al destinatario.

Después de realizar cierto procesamiento el proceso remoto envía la respuesta al proceso local mediante *mnx_send()*, tal como se mencionó antes, y como el proceso local estaba esperando por esa respuesta, el *DVK* local lo desbloquea y le envía un *CMD_SEND_ACK* al proceso remoto para desbloquearlo.

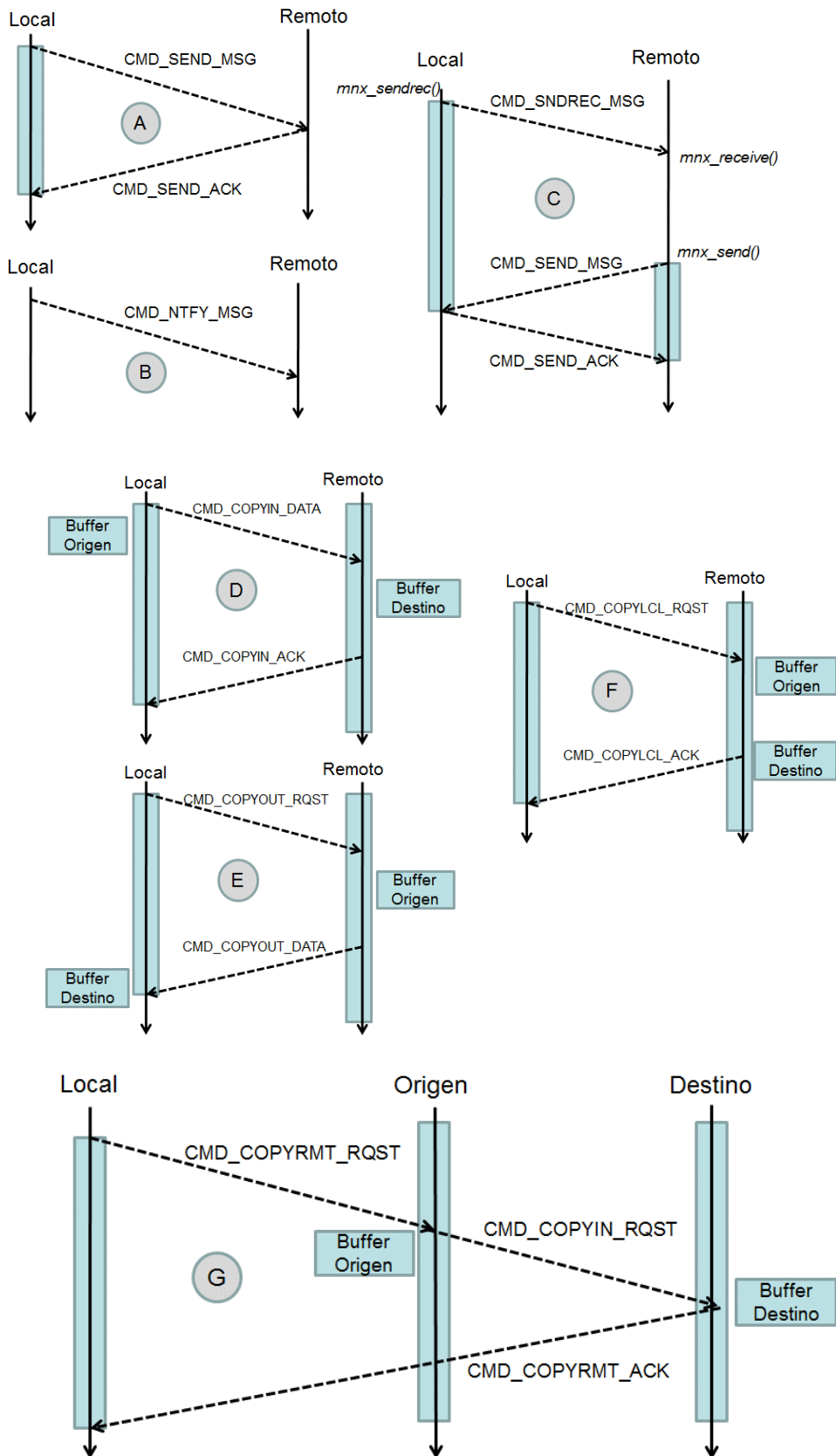


Figura 29. Protocolo para Transferencias de Mensajes y Copias de Datos Remotas.

Cuando un proceso invoca la función *mnx_vcopy()* se dice que es el *Solicitante*. Si un proceso del nodo Local quiere realizar una copia de datos utilizando *mnx_vcopy()* desde un buffer del proceso local a un buffer del proceso remoto (Fig. 29-D), el proceso remoto debe estar bloqueado. Se genera en el *DVK* local un comando *CMD_COPYIN_DATA* como encabezado, acompañado de una carga (*payload*) que contiene los datos a copiar. Una vez extraídos por el Proxy Emisor Local (*mnx_get2rmt()*), los envía al Proxy Receptor Remoto, quien lo entrega al *DVK* remoto (*mnx_put2lcl()*) para copiar los datos en el buffer del proceso destino. Luego, el *DVK* remoto genera un comando *CMD_COPYIN_ACK* y lo inserta en la cola de comandos del Proxy Emisor Remoto. Cuando éste lo extrae (*mnx_get2rmt()*), lo envía al Proxy Receptor Local para que lo inserte en el *DVK* local (*mnx_put2lcl()*). Hecho esto, el *DVK* local desbloquea al proceso *Solicitante*.

Si un proceso del nodo Local quiere realizar una copia de datos utilizando *mnx_vcopy()* desde un buffer de un proceso remoto a un buffer propio (Fig. 29-E), el proceso remoto origen de la copia debe estar bloqueado. Se genera en el *DVK* local un comando *CMD_COPYOUT_RQST*, el cual es extraído por el Proxy Emisor Local (*mnx_get2rmt()*), los envía al Proxy Receptor Remoto para lo entregue al *DVK* (*mnx_put2lcl()*). El *DVK* remoto genera un comando *CMD_COPYOUT_RQST* con los datos a copiar como carga y los inserta en la cola del Proxy Emisor Remoto. Cuando éste lo extrae (*mnx_get2rmt()*), lo envía al Proxy Receptor Local, entregándolo al *DVK* local (*mnx_put2lcl()*) para que copie los datos en el buffer del proceso local. Hecho esto, el proceso *Solicitante* es desbloqueado.

Cuando un proceso del nodo Local quiere realizar una copia de datos utilizando *mnx_vcopy()* desde un buffer de un proceso remoto a un buffer de otro proceso remoto del mismo nodo (Fig. 29-F), ambos procesos remotos deben estar bloqueados. El proceso *Solicitante* también quedará bloqueado hasta finalizar la operación. Se genera en el *DVK* local un comando *CMD_COPYLCL_RQST*, el cual es extraído por el Proxy Emisor Local (*mnx_get2rmt()*), lo envía al Proxy Receptor Remoto para lo entregue al *DVK* remoto (*mnx_put2lcl()*). El *DVK* remoto copia los datos desde el buffer del proceso origen al buffer del proceso destino y luego genera un comando *CMD_COPYLCL_ACK* y lo inserta en la cola del Proxy Emisor Remoto. Cuando éste lo extrae (*mnx_get2rmt()*), lo envía al Proxy Receptor Local, quien lo entrega al *DVK* local (*mnx_put2lcl()*) para desbloquear el proceso *Solicitante*. En este caso no hay transferencia de los datos copiados por la red.

Cuando un proceso del nodo Local quiere realizar una copia de datos utilizando *mnx_vcopy()* desde un buffer de un proceso remoto a un buffer de otro proceso remoto localizado en otro nodo (Fig. 29-G), ambos procesos remotos deben estar bloqueados. El proceso *Solicitante* también quedará bloqueado hasta finalizar la operación. Se genera en el *DVK* local un comando *CMD_COPYRMT_RQST*, el cual es extraído por el Proxy Emisor Local (*mnx_get2rmt()*), lo envía al

Proxy Receptor Remoto del nodo origen para que lo entregue al *DVK* (*mnx_put2lcl()*). El *DVK* del nodo origen genera un comando *CMD_COPYIN_RQST* conjuntamente con los datos a copiar como carga y lo inserta en la cola del Proxy Emisor Remoto Origen. El Proxy Emisor Remoto Origen, extrae el comando (*mnx_get2rmt()*) y lo envía al Proxy Receptor Remoto Destino. Cuando el Proxy Receptor Remoto Destino extrae el comando lo inserta en el *DVK* destino (*mnx_put2lcl()*). El *DVK* destino copia los datos que vienen en el comando como carga al buffer del proceso destino y genera un comando *CMD_COPYRMT_ACK* destinado al *Solicitante*. El Proxy Emisor Remoto Destino obtiene el comando (*mnx_get2rmt()*) y lo envía al Proxy Receptor Local del *Solicitante* quien lo inserta en el *DVK* local (*mnx_put2lcl()*). Finalmente, el *DVK* local desbloquea al proceso *Solicitante*.

4.1.4. APIs de Gestión de Contenedores Distribuidos

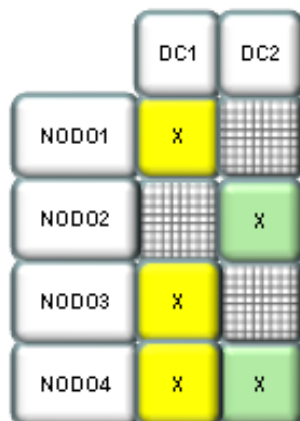
En cada nodo, en donde un *DC* pueda expandirse, se debe registrar ante el módulo del *DVK* de ese nodo. El *DVK* ofrece las siguientes APIs para gestión de *DCs*:

<code>node_id = mnx_dc_init(dc_usr);</code>	Esta función permite registrar un nuevo <i>DC</i> en el módulo del <i>DVK</i> del nodo local especificando sus características en la estructura <i>dc_usr</i> de tipo <i>DC_usr</i> . Si el valor de retorno es mayor o igual a 0, identifica al nodo local. Si es menor que cero es un código de error.
<code>rcode = mnx_dc_end(dc_id);</code>	Esta función permite de-registrar a un <i>DC</i> del módulo del <i>DVK</i> especificando su <i>dc_id</i> .
<code>rcode = mnx_getdcinfo(dc_id, dc_usr);</code>	Esta función permite obtener en una estructura de datos <i>dc_usr</i> tipo <i>DC_usr</i> , la información del <i>DC</i> especificando su <i>dc_id</i> .
<code>rcode = mnx_add_node(dc_id, node_id);</code>	Esta función permite registrar en el módulo del <i>DVK</i> del nodo local a un nodo remoto (<i>node_id</i>) en donde un <i>DC</i> (<i>dc_id</i>) puede expandirse.
<code>rcode = mnx_del_node(dc_id, node_id);</code>	Esta función permite de-registrar del módulo del <i>DVK</i> del nodo local a un nodo remoto (<i>node_id</i>) en donde un <i>DC</i> (<i>dc_id</i>) se había expandido.

Con los nodos y los *DCs* creados, los mismos deber relacionarse entre sí utilizando la API *mnx_add_node()* en cada uno de los nodos que el *DC* cubra. Por ejemplo, supóngase el *DC1* cubre *NODO1*, *NODO3*, *NODO4*, y *DC2* cubre *NODO2* y *NODO4*, es decir el mapeo entre *DCs* y Nodos sería el que se muestra la Tabla IX.

Una vez registrados los *proxies* que conectan a todos los nodos entre sí y registrados los nodos en el *DVS* (en el módulo del *DVK* de cada uno de los nodos), se debe proceder a mapear los *DCs* a los nodos en cada uno de los nodos del *DVS*.

TABLA IX.EJEMPLO DE MAPEO ENTRE DCs Y NODOS



El siguiente es un ejemplo de mapeo entre nodos y DCs que se realiza utilizando las APIs del *DVS*, y así conformar el mapa de la Tabla IX.

<p>En NODO1</p> <pre> mnx_dc_init(dc1_usr); mnx_add_node(DC1, NODO3); mnx_add_node(DC1, NODO4); </pre>	<p>En NODO2</p> <pre> mnx_dc_init(dc2_usr); mnx_add_node(DC2, NODO4); </pre>	<p>En NODO3</p> <pre> mnx_dc_init(dc1_usr); mnx_add_node(DC1, NODO1); mnx_add_node(DC1, NODO4); </pre>	<p>En NODO4</p> <pre> mnx_dc_init(dc1_usr); mnx_dc_init(dc2_usr); mnx_add_node(DC1, NODO1); mnx_add_node(DC1, NODO3); mnx_add_node(DC2, NODO2); </pre>
---	---	---	---

Actualmente, todo este conjunto de operaciones se realiza utilizando scripts, pero es más adecuado para la operación disponer de una aplicación distribuida (con sus APIs) en todos los nodos del *DVS*. La aplicación debería permitir registrar un *DC* ante su componente local en cualquiera de los nodos, y automáticamente el *DC* se registraría en el resto de los nodos que lo componen. Se deja el desarrollo de esta aplicación de gestión distribuida del *DVS* para trabajo futuro.

4.1.5. APIs de Gestión de Procesos

Cada tarea Linux que se ejecute bajo el dominio de un *DC* debe registrarse como proceso ante el módulo del *DVK* del nodo local.

El *DVK* ofrece las siguientes APIs para gestión de procesos:

<pre>proc_ep = mnx_bind(dc_id,endpoint);</pre>	<p>Esta función permite registrar en un <i>DC</i> (<i>dc_id</i>) al proceso que lo invoca especificando <i>endpoint</i> a asignar. Se registra ante el módulo del <i>DVK</i> del nodo local como <i>endpoint</i> de tipo Local. Como un <i>endpoint</i> puede adquirir valores negativos, solo en el caso que el valor de retorno <i>proc_ep</i> sea menor a 300 (<i>EDVSERRCODE</i>) se trata de un código de error, de</p>
--	--

	lo contrario el valor de retorno debería ser igual a endpoint . Esta función solo puede ser utilizada por procesos con privilegios.
thread_ep = mnx_tbind(dc_id,endpoint);	Idem <i>mnx_bind()</i> , pero para un hilo.
proc_ep = mnx_lclbind(dc_id,pid,endpoint);	Idem <i>mnx_bind()</i> , pero registra a otro proceso local cuyo PID es <i>pid</i> .
proc_ep = mnx_rmtbind(dc_id, name, endpoint, node_id);	Esta función permite registrar en un <i>DC (dc_id)</i> a un proceso remoto especificando el endpoint y el node_id donde está ejecutando. Se registra ante el módulo del <i>DVK</i> del nodo local como endpoint de tipo Remoto.
proc_ep = mnx_bkupbind(dc_id, pid, endpoint, node_id);	Esta función permite registrar en un <i>DC (dc_id)</i> a un proceso local de tipo Respaldo especificando su endpoint y el node_id donde está ejecutando el proceso de tipo Primario.
proc_ep = mnx_replbind(dc_id, pid, endpoint);	Esta función permite registrar en un <i>DC (dc_id)</i> a un proceso local de tipo Réplica especificando su endpoint y el PID (<i>pid</i>) del proceso.
proc_ep = mnx_unbind(dc_id, endpoint);	Esta función permite de-registrar de un <i>DC (dc_id)</i> a proceso especificando su endpoint
ret = mnx_getprocinfo(dc_id, endpoint, proc_usr);	Esta función permite obtener información de estado y estadística de un proceso registrado en un <i>DC (dc_id)</i> dado su endpoint . En caso de no haber errores (<i>ret</i> = 0), la información del proceso se almacena en la estructura <i>proc_usr</i> .
proc_ep = mnx_wait4bind(); proc_ep = mnx_wait4bind_T(to_ms);	Esta función permite bloquear a una tarea Linux que la invoca hasta tanto la misma sea registrada en un <i>DC</i> (generalmente lo hace otro proceso o hilo). Como resultado se retorna su endpoint (proc_ep > EDVSERRCODE) . En su versión temporizada se puede especificar un tiempo de espera de <i>to_ms</i> en milisegundos.
proc_ep = mnx_wait4bindep(endpoint); proc_ep = mnx_wait4bindep_T(endpoint, to_ms);	Esta función permite bloquear al proceso registrado en un <i>DC</i> que la invoca hasta tanto se registre en el módulo del <i>DVK</i> local el endpoint especificado en el mismo <i>DC</i> . Como resultado se retorna su endpoint (proc_ep > EDVSERRCODE) . En su versión temporizada se puede especificar un tiempo de espera de <i>to_ms</i> en milisegundos.
ret = mnx_wait4unbind(endpoint); ret = mnx_wait4unbind_T(endpoint, to_ms)	Esta función permite bloquear al proceso registrado en un <i>DC</i> que la invoca hasta tanto se de-registre del módulo del <i>DVK</i> local el endpoint especificado en el mismo <i>DC</i> . En su versión temporizada se puede especificar un tiempo de espera de <i>to_ms</i> en milisegundos.

Una característica que el programador debe considerar, es que si un proceso registrado en un *DC* crea un nuevo proceso mediante *fork()* de Linux, el proceso hijo no hereda los atributos del *DC* de su padre. Es decir, si proceso hijo va a ser incluido en el mismo *DC* del padre, se deberá registrar ante el *DVK* asignándole un **endpoint (mnx_lclbind())**. En la siguiente porción de código fuente, un proceso padre invoca a *fork()* para crear un proceso hijo y registrarlo ante el *DVK* local con

endpoint=CHILD_EP. El proceso hijo entra en loop hasta que es registrado por su padre (o hasta que se produce un error).

```

if ((child_lpid = fork()) == 0) {
    // CHILD
    do {
        child_ep = dvk_wait4bind_T(FORK_WAIT_MS);
        if(child_ep < EDVSERRCODE) exit(EXIT_FAILURE);
    }while(child_ep == EDVSTIMEDOUT);

    // CHILD has child_ep endpoint number.
}else{
    // PARENT
    child_ep = mnx_lclbind(dc_id, child_lpid, CHILD_EP);
    // parent code
}

```

En el caso del VOS-multiserver que forma parte del prototipo, esta operación la hace automáticamente al realizar el *fork()* del VOS y no el *fork()* del Linux. Si un proceso dentro de un *DC* invoca la Llamada al Sistema *fork()*, ésta es interceptada por la coraza de protección y se invoca a la función *do_fork()* del VOS quien invoca al *fork()* de Linux y realiza la registración del proceso hijo asignándole un *endpoint*.

4.1.6. APIs de Gestión de Privilegios

Cada proceso registrado en el módulo local del *DVK* tiene asignado una serie de privilegios. Esto se implementa, incorporando a cada descriptor de proceso en el módulo local del *DVK* una estructura de privilegios *priv_usr*.

```

struct priv_usr {
    sys_id_t s_id;        // ID del descriptor de privilegios
    int      s_warn;     // Endpoint del proceso a notificar en exit() implicito
    int      s_level;   // Nivel de privilegios
    short    s_trap_mask; // Máscara de IPCs habilitados
    sys_map_t s_ipc_from; // Máscara de emisores habilitados
    sys_map_t s_ipc_to;  // Máscara de destinatarios habilitados
    long     s_call_mask; // Llamadas al Kernel habilitadas
    multimer_t s_alarm_timer; // temporizador/Alarma
};

```

El tratamiento de los privilegios proviene de Minix 3, pero se agregaron algunos campos en su versión del *DVK*. Algunos campos no son utilizados por la versión actual de M3-IPC, como por ejemplo: la restricción en cuanto a los tipos de IPC que un proceso puede invocar (*s_trap_mask*), o los procesos de los que se pueden recibir mensajes (*s_ipc_from*) o a cuáles procesos se les puede enviar mensajes (*s_ipc_to*). Los controles que se puede hacer utilizando los campos contenidos en la estructura de privilegios permiten mejorar sustancialmente la seguridad de un VOS y sus aplicaciones. De todos modos, la implementación de dichos controles en Minix 3 solo está reducida a un pequeño

número de procesos, por lo que debe investigarse acerca de una mejor forma de representarlos para que resulte escalable a la cantidad de procesos que puede contener un *DC*. Si bien podrían haberse utilizado los mismos controles que Minix 3, este tema aún no fue abordado en el desarrollo del prototipo.

El campo *s_warn* permite indicar el *endpoint* de un proceso al que se le enviará un mensaje de notificación cuando el proceso en cuestión finalice en forma implícita, es decir sin ejecutar la llamada al sistema *exit()*. El campo *s_level* permite distinguir entre procesos de diferentes niveles de privilegios: procesos del kernel, tareas utilizadas por los gestores de dispositivos, servidores del sistema y procesos de usuarios. En la versión actual del *DVK*, solo se distinguen entre procesos del sistema (kernel, tareas, servidores) y procesos de usuario.

El *DVK* ofrece las siguientes APIs para gestión de privilegios:

<code>rcode = mnx_setpriv(dc_id, endpoint, priv_usr);</code>	Esta función permite asignarle una configuración de privilegios (<i>priv_usr</i>) a un proceso (<i>endpoint</i>) registrado en un <i>DC</i> (<i>dc_id</i>).
<code>rcode = mnx_getpriv(dc_id, endpoint, priv_usr);</code>	Esta función permite obtener la configuración actual de privilegios (<i>priv_usr</i>) de un proceso (<i>endpoint</i>) registrado en un <i>DC</i> (<i>dc_id</i>).

4.1.7. M3-IPC

El desarrollo de un sistema distribuido se facilita al disponer de una infraestructura de IPC que provea una semántica uniforme y que no considere la ubicación de los procesos que se comunican. Pero, como además como el *DVS* es un sistema de virtualización, los mecanismos de IPC deben soportar el confinamiento de las comunicaciones entre procesos ejecutando en diferentes *DCs*. Adicionalmente a esas características básicas, hay otros requerimientos para que se pueda construir un *DVS* de calidad clase-proveedor, por ejemplo, que las comunicaciones se mantengan operacionales durante la migración de procesos o ante el fallo de un proceso replicado. Todas estas características deben complementarse con un rendimiento aceptable para las comunicaciones de procesos intra-nodo e inter-nodo que hagan del *DVS* un sistema factible de ser utilizado en producción.

M3-IPC fue desarrollado para comunicar componentes de un VOS, pero que puede ser utilizado como infraestructura para el diseño de aplicaciones distribuidas genéricas. El mecanismo de IPC resultante es una herramienta de propósito general que puede ser utilizado en una amplia gama de aplicaciones, como por ejemplo una aplicación distribuida con una programación más sencilla. En la Fig.30, una aplicación distribuida utiliza M3-IPC para comunicar sus componentes que se ejecutan individualmente en diferentes VMs. En cada VM ejecuta un OS-host con el módulo *DVK*. En este

caso no hay ningún VOS subyacente a la aplicación, sino que ésta utiliza tanto los servicios del OS-host como del módulo *DVK*.

Antes de comenzar con el desarrollo de M3-IPC, se evaluaron varios mecanismos de IPC, pero ninguno de ellos cumplía con todos los requerimientos mencionados. Otra posibilidad que se analizó fue la de realizar modificaciones a alguno de los mecanismos de IPC existentes, de tal forma de incorporarle aquellas características faltantes. La conclusión fue que el mantenimiento se complicaría en la medida que el software base tuviese modificaciones o actualizaciones con cambios de gran impacto. Habiendo evaluado estas opciones, se decidió tomar el camino inverso: construir el mecanismo de IPC que cumpla los requerimientos de un *DVS* tomando como base el código fuente de un IPC de un OS de microkernel.

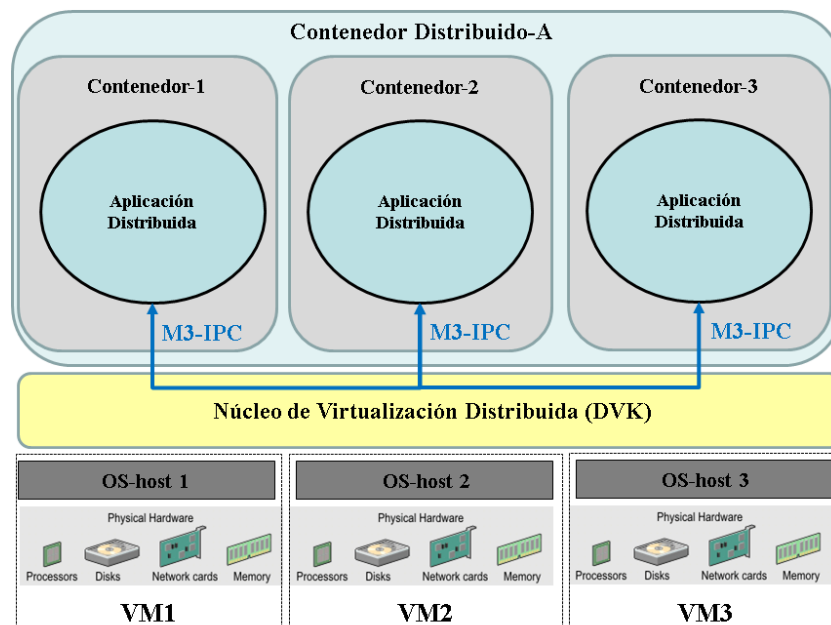


Figura 30. Ejemplo de Aplicación Distribuida utilizando M3-IPC.

Si bien muchos temas relacionados al IPC ya fueron investigados en el pasado [112], la promesa de un resurgimiento de los sistemas distribuidos y los sistemas operativos multikernel [9, 113, 162] destacará el impacto que el IPC tenga en el desarrollo de OSs, VMMs y otros sistemas distribuidos en general.

En el pasado se realizaron múltiples esfuerzos para proporcionar mecanismos de IPC para sistemas distribuidos, algunos de los cuales se integraban como un componente dentro de un DOS clásico [114, 115] y otros como software agregado a través de bibliotecas, parches o módulos [116, 117, 118, 119, 120]. Sus características, semántica y rendimiento se evaluaron previamente y durante las etapas de diseño, implementación y prueba de M3-IPC (Tabla X).

Varios de esos mecanismos de IPC fueron utilizados para contrastar el rendimiento de M3-IPC, tal como se muestra en la [Sección 5](#). No se consideró el URPC [121] porque, aunque puede alcanzar un buen rendimiento, es muy débil en cuestiones relacionadas con la seguridad. Las colas de mensajes, las tuberías, los FIFO y los sockets de Unix no tienen la capacidad de comunicarse con procesos remotos, de todos modos, se utilizaron como mecanismos de contraste para las pruebas de rendimiento comparativo en transferencias locales. DIPC [118] presenta características interesantes, pero carece de mantenimiento. D-Bus [122, 123] también se descartó ya que presenta un rendimiento moderado producto de su diseño. Este se basa en un servidor en modo usuario al que se conectan los procesos mediante sockets Unix, por lo que hay una doble transferencia de los mensajes y se duplica el número de cambio de contextos.

TABLA X.COMPARACIÓN DE CARACTERÍSTICAS DE MECANISMOS DE IPC (PARCIAL)

Características	SIMPL	SRR	DIPC	RPC	TIPC
Semántica Requerida	Si	Si	No	Si	No
IPC Local y Remoto	Si	No	Si	Si	Si
Rendimiento Local	Bajo	Bajo	----	Bajo	Medio
Rendimiento Remoto	Bajo	----	----	Alto	Alto
Protocolo utilizado	TCP	----	RPC	UDP/TCP	IP
Ultimo Release estable	Ene/2012	Ene/2010	No hay	Actualizado	Actualizado

Una de las características más importantes y necesaria para construir un *DVS* es el aislamiento de IPC, que incluye las siguientes propiedades:

- *Confinamiento*: Un proceso que se ejecuta dentro de un *DC* no puede comunicarse con ningún proceso fuera de su propio *DC*.
- *Direccionamiento Privado*: Un **endpoint** asignado a un proceso de un *DC* podría asignarse a otros procesos que se ejecutan dentro de otros *DCs*.
- *Transparencia de Virtualización*: A todo proceso (sin privilegios) se le oculta su pertenencia a un *DC*, el nodo en donde está ejecutando, y los nodos donde ejecutan otros procesos del mismo *DC*. Para permitir la administración de *DVS*, ciertos procesos con privilegios pueden asignar **endpoints** y *DCs* para otros procesos no relacionados. Esta facilidad es utilizada, por ejemplo, para iniciar los procesos que conforman un *VOS*.

Habiendo decidido encarar el desarrollo de M3-IPC, se establecieron los siguientes objetivos de diseño:

- *Fácil de Usar*: Definir, implementar y desplegar aplicaciones usando M3-IPC debe ser sencillo.
- *Transparencia de Ubicación*: Las aplicaciones no requieren considerar la ubicación (nodos) de los procesos involucrados, facilitando así la programación.
- *Confinamiento*: El IPC debe confinarse dentro de cada *DC* para lograr un grado adecuado de aislamiento.
- *Concurrencia*: Debe soportar la utilización de subprocesos o hilos para maximizar el rendimiento y la eficiencia.
- *Soporte Transparente de Migración Dinámica*: Las comunicaciones deben mantenerse activas aun después de la migración de nodo de uno de los procesos involucrados.
- *Soporte para Mecanismos de Tolerancia a Fallos*: Las comunicaciones deben permanecer operativas incluso durante el fallo de un proceso Primario y su reemplazo por alguno de los procesos de Respaldo. Este fallo y cambio de procesos debe ocultarse al resto de los procesos.
- *Rendimiento*: Los objetivos de rendimiento se consideran satisfechos cuando se alcanzan umbrales mínimos establecidos para transferencias locales y remotas. Para ello, deben contrastarse con otros mecanismos de IPC existentes.
- *Escalabilidad*: La escalabilidad se relaciona con la eficiencia en la utilización de los recursos. La eficiencia de las comunicaciones entre procesos ubicados en diferentes nodos debe estar limitado por la capacidad de red (latencia y ancho de banda) y el protocolo de transporte que se utiliza, no por la utilización de CPU.
- *Agnóstico del Protocolo de Comunicación*: Los administradores del *DVS* pueden elegir el protocolo de red/transporte a utilizar en cada uno de los *proxies*. Pudiendo considerar características adicionales como la compresión de datos y el cifrado.

-
- *Modular y Parametrizable*: Su desarrollo debe ser modular de tal forma de poder incluir más funcionalidades a futuro, y parametrizable para adecuarse a diferentes tipos de clusters y usos.
 - *Cliente/Servidor y orientado a RPC*: Debe soportar los patrones de comunicación típicos entre múltiples clientes contra múltiples servidores sin un intermediario.
 - *Seguridad*: Deben aplicarse a su diseño e implementación los principios básicos de seguridad.

M3-IPC se basa en la suposición de un clúster de nodos conectados a una LAN conmutada donde la tasa de pérdida de paquetes es baja y el ancho de banda disponible es alto.

Se tomaron dos decisiones importantes antes de desarrollar M3-IPC: su construcción debía utilizar abstracciones y facilidades de software disponibles en el kernel de Linux, y debía sacar provecho del paralelismo en los sistemas SMP y multi-núcleo. Antes de comenzar el desarrollo, se evaluaron varios mecanismos de sincronización del kernel de Linux y facilidades de exclusión mutua. Los semáforos de kernel y RCU (lectura-copia-actualización) resultaron demasiado lentos, al menos para la cantidad de núcleos por CPU utilizados (4 y 8 núcleos) en las pruebas; las cerraduras giratorias (*spinlocks*) eran algo más lentas que las cerraduras de lectura/escritura (*rwlocks*); finalmente, los *mutexes* se utilizan para exclusión mutua. De todos modos, se habilitaron varias opciones de compilación para que los programadores del sistema puedan seleccionar entre varios mecanismos de sincronización y exclusión mutua que desean utilizar para compilar el módulo del *DVK*.

Otra decisión de diseño se relacionó con el mecanismo de transferencia de datos del espacio de usuario al espacio del kernel y viceversa. Los conectores *Netlink* [124] tienen una inicialización bastante tediosa y requería demasiadas adaptaciones para hacer que cumpla con las necesidades del proyecto *DVS* (por ejemplo, confinamiento). *Efficient Capability-Based Messaging* (ECBM) [125] tiene un alto rendimiento, pero se descuidaron los aspectos básicos relativos a seguridad, por lo que fue descartado. Finalmente, se usaron las funciones *copy_to_user()* y *copy_from_user()* proporcionadas por el kernel de Linux. Para mejorar el rendimiento, se creó una función llamada *copy_usr2usr()* para copiar datos (alineados con el tamaño de página) desde el buffer de espacio de usuario de un proceso origen al buffer de espacio de usuario de un proceso destino.

Las características de la implementación de M3-IPC se resumen a continuación:

-
- Las API admiten hilos y se implementaron utilizando las facilidades de exclusión mutua, colas de tareas (*task queues*) y espera de eventos (*event wait*) provistas por el kernel de Linux. Las colas de tareas y la espera de eventos se utilizaron para la sincronización de procesos; y los Contadores de Referencia (*reference counters*) se utilizaron para contabilizar los procesos que hacen referencia a una estructura de datos.
 - La granularidad de las secciones críticas se maximizó a nivel de proceso/hilo, permitiendo transferencias de mensajes paralelas entre múltiples pares de procesos. Como los *DC* no comparten ninguna estructura de datos entre sí durante las transferencias simultáneas de mensajes entre pares de procesos que pertenecen a diferentes *DC*, se logra un mayor rendimiento por una menor contención de recursos.
 - Las estructuras de datos que se utilizan con frecuencia para los procesos registrados están alineadas con las líneas de caché L1 para reducir el tiempo de acceso.
 - Se implementó un Protocolo de Herencia de Afinidad (similar al conocido Protocolo de Herencia de Prioridades) para minimizar el impacto en el rendimiento del ping-pong de caché. Para ello, se utilizó la facilidad proporcionada en Linux para especificar la afinidad del proceso con un conjunto de procesadores/núcleos. De esta forma, hay una mayor posibilidad de que, por ejemplo, un mensaje permanezca en la memoria caché L1 cuando se conmuta la CPU o núcleo al proceso/hilo destinatario, lo que reduce el tiempo de acceso al mensaje.
 - Cada *DC* puede asignar una máscara de CPU para especificar en qué CPU se pueden ejecutar sus procesos locales (solo significativo dentro de cada nodo).
 - La copia de los bloques de datos y las transferencias de mensajes entre los procesos coubicados se realizan desde el espacio de direcciones del proceso de origen hasta el espacio de direcciones del proceso de destino sin ninguna copia intermedia a través del kernel. La copia es hecha por el kernel de Linux a través del mecanismo de copiar en escritura (*CoW*: copy on write).
 - La información de depuración se envía al buffer de anillo del kernel de Linux y se puede mostrar mediante el comando *dmesg*.

- Los bloques de datos que están alineados con una página de memoria y cuyas longitudes sean mayores o iguales al tamaño de página se copian usando la función *page_copy()* proporcionada por el kernel, que es muy eficiente porque usa instrucciones MMX.
- La información sobre la configuración, el estado y las estadísticas de las estructuras de datos de M3-IPC se presenta como directorios y archivos dentro del sistema de archivos Linux */proc*.

M3-IPC se ha implementado en el lenguaje de programación C en Linux para Intel x86 de 32 bits y se distribuye como un parche de kernel, un módulo kernel y un conjunto de bibliotecas.

❑ Mensajes y Bloques de Datos

Los mecanismos de IPC de M3-IPC soportan las transferencias de mensajes y de bloques de datos. Un mensaje es una estructura de datos de tamaño fijo que soporta múltiples formatos. Esta estructura proviene de Minix 3 (Fig. 31), la cual puede ampliarse para satisfacer algún requerimiento particular de un VOS o de una aplicación, pero manteniendo la compatibilidad con los formatos básicos existentes.

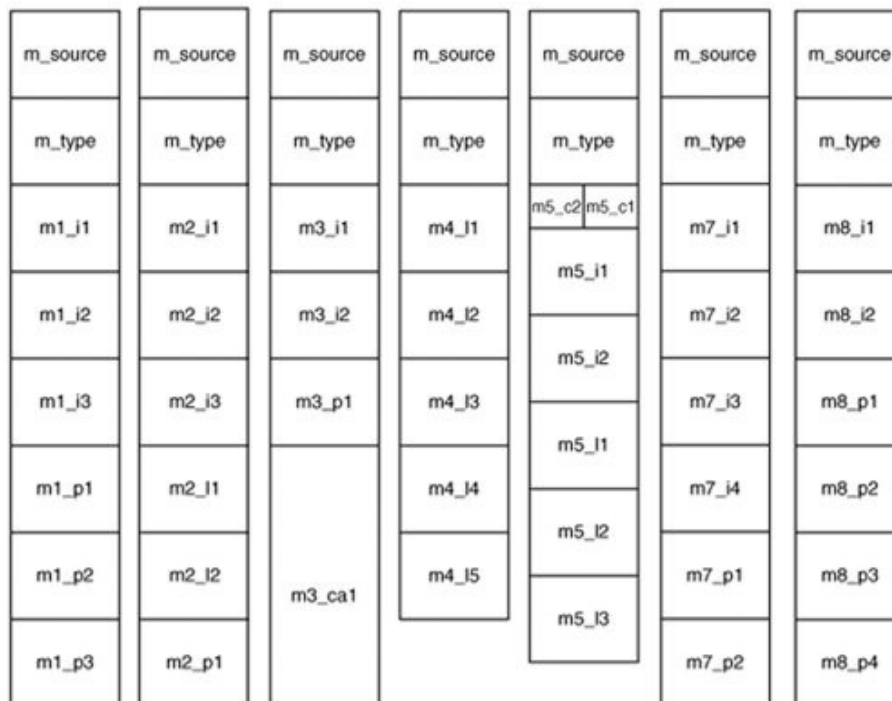


Figura 31. Formatos de Mensajes en Minix 3 (Obtenido de [126])

El formato de un mensaje está definido por la estructura de datos *message*. Su encabezado comienza con el *endpoint* del Emisor (*m_source*) y el tipo de mensaje (*m_type*) que define el formato de los siguientes campos.

```
typedef struct {
    int m_source; /* endpoint del Emisor */
    int m_type;   /* tipo de mensaje   */
    union {
        mess_1 m_m1;
        mess_2 m_m2;
        mess_3 m_m3;
        mess_4 m_m4;
        mess_5 m_m5;
        mess_7 m_m7;
        mess_8 m_m8;
    } m_u;
} message;
```

El tipo de mensaje se acuerda entre los procesos intervinientes, no es establecido por el sistema. Luego le siguen diversos formatos preestablecidos que combinan campos de números enteros, punteros, caracteres, etc. (Fig. 31). La nomenclatura utilizada para nombrar a los campos está dada por el tipo de formato de mensaje (*m1_ a m8_*), luego una letra que identifica al tipo de dato (*i*: integer/entero, *l*: long/largo, *p*: pointer/puntero, *c*: caracter) y finalmente, un número de orden de aparición dentro de la estructura.

En general, cuando se requieren transferir pocos datos de un proceso a otro proceso se lo hace mediante la transferencia de mensajes. Cuando el tamaño de los datos a transferir es más grande, se debería utilizar la copia de bloques de datos. En este caso un bloque de datos está definido por una dirección virtual de inicio del buffer que contiene los datos y la longitud de los datos a transferir. Para realizar una transferencia de datos la misma debe especificarse con la 5-tupla: {*Endpoint_Origen*, *Dirección_Origen*, *Endpoint_Destino*, *Dirección_Destino*, *Tamaño_en_Bytes*}. Cuando se trata de una copia entre procesos que se encuentran en diferentes nodos, esta 5-tupla se encuentra en la estructura de datos *vcopy_s*, la que contiene un campo adicional para el *endpoint* del proceso *Solicitante* (requester) de la copia.

```
struct vcopy_s {
    int v_src;           // Endpoint Origen
    int v_dst;           // Endpoint Destino
    int v_rqtr;          // Endpoint del Solicitante (requester)
    void *v_saddr;       // Dirección virtual de los datos en el origen
    void *v_daddr;       // Dirección virtual de los datos en el destino
    int v_bytes;         // Cantidad de Bytes a copiar
};
```

M3-IPC soporta la copia de bloques de datos entre 2 procesos ordenada por un tercer proceso (el *Solicitante*). Esto se utiliza, por ejemplo, cuando se requieren copiar datos desde un proceso no privilegiado hacia otro proceso no privilegiado. Si uno de los dos procesos no privilegiados intenta la

copia, la misma sería rechazada por no contar con los privilegios para acceder al espacio de direcciones de otro proceso. Para realizar la copia, uno de los procesos no privilegiados debe solicitarle una copia a un proceso privilegiado, mediante la transferencia de un mensaje. El proceso privilegiado puede realizar la copia entre ambos procesos no privilegiados dado que dispone de los privilegios para hacerlo. En otros casos, donde uno de los procesos cuenta con los privilegios suficientes, le permitiría a éste ser origen o destino de la copia de datos y también solicitante. M3-IPC se encarga de realizar la copia sin considerar los nodos donde se encuentran ejecutando los procesos cuyos *endpoints* son especificados en la solicitud. Puede darse el caso que los 3 *endpoints* involucrados (*Solicitante*, *Origen* y *Destino*) se encuentren ejecutando en diferentes nodos.

□ APIs para Transferencias de Mensajes

Las APIs para transferencias del *DVK* son versiones equivalentes, mejoradas y ampliadas de aquellas existentes en el microkernel de Minix 3.

<pre>rcode = mnx_send(dst_ep, message); rcode = mnx_send_T(dst_ep,m,to_ms);</pre>	<p>Estas funciones se utilizan para enviar un mensaje (<i>message</i>) a un proceso con un determinado <i>endpoint</i> (<i>dst_ep</i>). Si el proceso destino no está esperando por un mensaje del emisor, entonces el emisor se bloquea. La variante temporizada soporta un tiempo de espera determinado (<i>to_ms</i>) en milisegundos. Si el receptor estaba esperando, el mensaje se transfiere y ambos procesos continúan su ejecución.</p>
<pre>rcode = mnx_receive(src_ep, message); rcode = mnx_receive_T(src_ep, message, to_ms);</pre>	<p>Estas funciones se utilizan para recibir un mensaje (<i>message</i>) de un proceso con un determinado <i>endpoint</i> (<i>src_ep</i>) o de cualquier proceso, especificándolo con el <i>endpoint</i> especial <i>ANY</i>. Si el proceso emisor no está bloqueado esperando enviar un mensaje al proceso receptor, entonces el receptor se bloquea. La variante temporizada soporta un tiempo de espera determinado (<i>to_ms</i>) en milisegundos. Si el proceso emisor estaba esperando enviar, el mensaje se transfiere y ambos procesos continúan su ejecución.</p>
<pre>rcode = mnx_sendrec(srcdst_ep, message); rcode = mnx_sendrec_T(srcdst_ep,m, to_ms);</pre>	<p>Estas funciones son utilizadas generalmente para realizar peticiones, en donde el proceso que la invoca envía primero una petición y luego queda a la espera de la respuesta. En lugar de hacer invocación <i>mnx_send()</i> seguida de un <i>mnx_receive()</i>, estas funciones concentran ambas operaciones. Un proceso realiza un requerimiento enviando un mensaje (<i>message</i>) a un proceso con un determinado <i>endpoint</i> (<i>srcdst_ep</i>) y luego queda a la espera de la respuesta. Si el proceso destino no está esperando por un mensaje del emisor, entonces el emisor se bloquea. La variante temporizada soporta un tiempo de espera determinado (<i>to_ms</i>) en milisegundos. Si el destinatario estaba esperando, se transfiere el mensaje desde el emisor al receptor y el receptor continúa su ejecución. Mientras tanto el emisor ahora queda</p>

	bloqueado esperando por un mensaje de respuesta. Cuando recibe el mensaje de respuesta (en el mismo buffer del mensaje de la petición), se desbloquea y continúa la ejecución. También, puede desbloquearse si no recibe la respuesta en un tiempo dado en la variante temporizada.
rcode: mnx_rcvrqst(message); rcode: mnx_rcvrqst_T(message, to_ms);	Estas funciones son optimizaciones de <i>mnx_receive()</i> y de su variante temporizada. Equivalen a un <i>mnx_receive(ANY, message)</i> , que son frecuentemente utilizadas por procesos servidores para esperar por peticiones de los procesos clientes. El <i>DVK</i> verifica que el emisor del mensaje de petición quede a la espera de la respuesta, es decir que haya ejecutado <i>mnx_sendrec()</i> o su variante temporizada.
rcode = mnx_reply(dst_ep, message);	Esta función es una optimización del <i>mnx_send()</i> cuando el proceso receptor (generalmente un servidor) conoce que el proceso destinatario con <i>endpoint (dst_ep)</i> está bloqueado esperando por un mensaje (<i>message</i>) de respuesta.
rcode = mnx_notify(dst_ep);	Esta función envía un mensaje con <i>endpoint</i> emisor <i>HARDWARE=(-1)</i> , con un tipo de mensaje en el que se codifica el <i>p_nr</i> del emisor del mensaje. Si el destinatario (<i>dst_ep</i>) no está esperando por el mensaje, el emisor no se bloquea, pero deja indicado en el descriptor del destinatario acerca del mensaje enviado. Luego, cuando el destinatario del mensaje realiza un <i>mnx_receive()</i> , el <i>DVK</i> controla si hay mensajes de tipo NOTIFY pendientes antes de buscar por procesos esperando enviar mensajes. Esta función solo puede ser utilizada por procesos con privilegios del sistema, es decir con $0 \leq \text{endpoint} < \text{nr_sys_procs}$.

Todas estas APIs de IPC permiten construir, entre otras cosas, un sistema de LPC o RPC (llamadas a procedimientos Locales o Remotos). Como el M3-IPC no distingue entre procesos Locales o Remotos, ambas son equivalentes. Para describir su modo de utilización, se presenta un ejemplo en pseudocódigo en el cual un proceso de Cliente realiza una solicitud de tipo *fs_open()* de un archivo a un proceso Servidor de Sistema de Archivos.

CLIENTE	LPC/RPC del CLIENTE
<pre>... rcode = fs_open(pathname, flags, mode); ...</pre>	<pre>int fs_open(const char *pathname, int flags, mode_t mode) { int ret; message msg; // se completan los campos del mensaje msg.type = FS_OPEN; msg.m1_i1 = flags; msg.m1_i2 = mode; msg.m1_p1 = pathname; // se envía el mensaje al servidor FS // y queda a la espera de la respuesta ret = mnx_sendrec(FS, msg); // se retorna el código de retorno return(ret); }</pre>

```

                                SERVIDOR DE SISTEMA DE ARCHIVOS
int fs_server()
{
int ret;
message msg;
    fs_init(FS); // se inicializa el servidor con endpoint FS
    while(TRUE){
        // queda a la espera de recibir peticiones
        ret = mnx_receive(ANY, msg); // podría utilizarse mnx_rcvrqst()

        // se clasifica según el tipo de requerimiento
        switch(msg.mtype){
            ...
            case FS_OPEN:
                ret = do_open(msg); // se efectúa la operación requerida
                break; // obteniendo los parametros del mensaje
            ...
        }
        msg.mtype = ret; // se devuelve el valor de retorno

        // se envía el mensaje de respuesta al Cliente
        ret = mnx_send(msg.m_source, msg); // podría utilizarse mnx_reply()
    }
}

```

Estas APIs son lo suficientemente versátiles como para permitir la construcción de un VOS distribuido completo, con múltiples servidores y tareas como componentes.

❑ APIs para Copias de Bloques de Datos

Minix 3 no incluye entre sus APIs la copia de bloques de datos, pero si la incluye como una Llamada al Kernel (Kernel Call). En el caso del *DVK*, se incluye como una API de M3-IPC.

rcode = mnx_vcopy(src_ep, src_addr, dst_ep, dst_addr, bytes);

Esta función permite copiar un bloque de datos de tamaño dado por *bytes* y localizado en la dirección *src_addr* del proceso origen con *src_ep* a la dirección *dst_addr* del proceso con *dst_ep*.

El rango de valores para el tamaño de bytes es $0 < \text{bytes} \leq d_max_copylen$ (establecido en el DVS).

El proceso que invoca a esta función puede o no ser alguno de los *endpoints* mencionados (*src_ep* o *dst_ep*). En caso de no serlo se lo conoce como el proceso Solicitante (requester).

Si el Solicitante es el origen o el destino, entonces el otro proceso (destino u origen) debe estar bloqueado para poder realizar la copia. Si el Solicitante no es ni origen ni destino, entonces tanto el origen como el destino deben estar bloqueados para llevar a cabo la copia. Todo esto es verificado por el *DVK*.

Para describir su modo de uso, se presenta un ejemplo en el cual un proceso de Cliente realiza una solicitud de tipo *fs_read()* de un archivo a un proceso Servidor de Sistema de Archivos.

CLIENTE	LPC/RPC del CLIENTE
... fd = fs_open(pathname, flags, mode);	size_t fs_read(int fd, void *buf, size_t count)

<pre> if(fd < 0) goto open_error; bytes = fs_read(fd, buf, count); ... </pre>	<pre> { size_t ret; message msg; // se completan los campos del mensaje msg.type = FS_READ; msg.ml_i1 = fd; msg.ml_i2 = count; msg.ml_p1 = buf; // se envía el mensaje al servidor FS // y queda a la espera de la respuesta ret = mnx_sendrec(FS, msg); return(ret); } </pre>
--	---

SERVIDOR DE SISTEMA DE ARCHIVOS

```

int fs_server()
{
int ret, bytes;
message msg;
char fs_buf[SSIZE_MAX];

fs_init(FS); // se inicializa el servidor con endpoint FS

while(TRUE){
// queda a la espera de recibir peticiones
ret = mnx_receive(ANY, msg); // podría utilizarse mnx_rcvrqst()

// se clasifica según el tipo de requerimiento
switch(msg.mtype){
...
case FS_READ:
bytes = do_read(msg, fs_buf); //se colocan los datos en el buffer
if(bytes < 0) goto read_err;

// El cliente (destino) está bloqueado esperando la respuesta
// El servidor (origen) es también el Solicitante de la copia
ret = mnx_vcopy (FS, // Endpoint del Origen
fs_buf, // Dirección del buffer origen
msg.m_source // Endpoint del cliente
msg.ml_p1, // Dirección de buffer destino
bytes); // cantidad de bytes a copiar

break;
...
}
msg.mtype = ret; // se devuelve el valor de retorno

// se envía el mensaje de respuesta al Cliente
ret = mnx_send(msg.m_source, msg); // podría utilizarse mnx_reply()
}
}

```

Las copias de datos entre espacios de direcciones de procesos diferentes (incluso de diferentes nodos) pueden retornar error (*EDVSMIGRATE*) en el caso de que alguno de los procesos involucrados (solicitante, origen o destino) se encuentre en proceso de migración.

El programador debe considerar los valores configurados en el *DVS* (*d_max_copypen*) y en el *DC* (*dc_max_copypen*) que especifican el tamaño máximo admisible de bloque de datos que se permite copiar.

4.1.8. APIs para Migración y de Replicación de Procesos

El *DVK* ofrece las siguientes APIs para la Migración y Replicación de Procesos:

rcode = mnx_migr_start(dcid, endpoint);	Esta función le informa al módulo local del <i>DVK</i> que el proceso con el <i>endpoint</i> especificado (tipo Local, Respaldo o Réplica) perteneciente al <i>DC=dcid</i> pretende migrarse de nodo. Si la función no retorna ningún error, el <i>DVK</i> local detiene todas las comunicaciones desde y hacia el <i>endpoint</i> mencionado.
rcode = mnx_migr_rollback(dcid, endpoint);	Esta función le informa al módulo local del <i>DVK</i> que el proceso con el <i>endpoint</i> especificado perteneciente al <i>DC=dcid</i> ha fallado en su intento de ser migrado hacia otro nodo. Por lo tanto, permanecerá ejecutando en el nodo original y se rehabilitan las comunicaciones desde y hacia ese <i>endpoint</i> .
rcode = mnx_migr_commit(pid, dcid, endpoint, new_node);	Esta función le informa al módulo local del <i>DVK</i> que el proceso con el <i>endpoint</i> especificado perteneciente al <i>DC=dcid</i> ha migrado hacia otro nodo con ID <i>new_node</i> . Solo en el caso que el <i>new_node</i> sea el nodo local, entonces debe especificarse el PID del proceso. Luego, las comunicaciones desde y hacia ese <i>endpoint</i> se rehabilitan.

Antes de la migración de un proceso, el *VOS* debe invocar en todos los nodos (abarcados por el *DC* del proceso a migrar) la función *mnx_migr_start()* para que se detengan las comunicaciones desde y hacia ese *endpoint*. Luego, el *VOS* procede a la migración del proceso que puede resultar exitosa o fallida. En caso de ser fallida, el *VOS* debe invocar a *mnx_migr_rollback()* en todos los nodos del *DC* de tal forma que se rehabiliten las comunicaciones desde y hacia ese *endpoint*. En caso de ser exitosa, el *VOS* debe invocar a *mnx_migr_commit()* en todos los nodos del *DC*. Esto hace que se rehabiliten las comunicaciones desde y hacia el *endpoint* del proceso migrado. En el nodo *new_node*, el *VOS* debe indicarle al *DVK* local del nodo el PID Linux del proceso migrado.

Cuando se trabaja con procesos replicados utilizando la técnica de Primario-Respaldo, se pueden utilizar las mismas funciones para indicar un cambio de Primario en la configuración. Cuando uno de los procesos tipo Respaldo detecta un fallo del proceso Primario (por ejemplo, a través de un detector de fallos), lo comunica al resto de los procesos Respaldo (por ejemplo, mediante GCS). Luego se elige el próximo Primario entre ellos. En este caso, se debe invocar en todos los nodos del

DC la función `mnx_migr_start()`, indicando el **endpoint** del Primario fallido (que es el mismo **endpoint** que el de todos los procesos tipo Respaldo). Una vez elegido el nuevo Primario, se debe invocar en todos los nodos del *DC* la función `mnx_migr_commit()` indicando en el parámetro **new_node** el nodo del nuevo Primario, y en dicho nodo se debe indicar el PID Linux del nuevo proceso Primario. Una vez finalizadas estas operaciones, las comunicaciones con el (nuevo) Primario se rehabilitan, siendo esto transparente a los procesos clientes.

Cuando se trabaja con procesos replicados utilizando la técnica de Máquina de Estados se pueden utilizar las mismas funciones para indicar, por ejemplo, un fallo en alguna de los procesos Réplica. Este fallo es detectado por el VOS (por ejemplo, a través de un detector de fallos). Como cada Réplica puede tener un subconjunto de procesos clientes que utilizan sus servicios, estos clientes deben re-direccionarse hacia otra Réplica de lo contrario quedarían sin atención. El VOS debe decidir cuáles de las Réplicas activas brindará servicios al subconjunto de clientes de la Réplica fallida. En primer lugar, en los nodos del *DC* donde no se ejecuta ninguna Réplica y donde se ejecutan clientes de la Réplica fallida, debe invocar `mnx_migr_start()` indicando el **endpoint** de la Réplica fallida y luego `mnx_migr_commit()` indicando en el parámetro **new_node** el nodo de la Réplica elegida. Una vez finalizadas estas operaciones, se rehabilitan las comunicaciones entre los clientes de la Réplica fallida con su nueva Réplica asignada, siendo esto transparente a los procesos clientes.

4.1.9. Directorio `/proc/dvs`

El módulo del *DVK* hace uso de la facilidad del sistema de archivos `/proc` de Linux para presentar información de configuración, de estado y estadísticas de los diversos componentes del *DVS*.

El primer directorio que se crea al iniciar el *DVS* en un nodo es `/proc/dvs`. A continuación, se presentan varios ejemplos con el contenido de ese directorio, sus subdirectorios y archivos creados por el *DVK*. Conjuntamente, con el directorio *dvs* se crea un subdirectorio **proxies** y dos archivos de texto **info** y **nodes**. Luego, por cada uno de los *DC* iniciados en el nodo local, se creará un subdirectorio con el nombre del *DC* correspondiente, en el ejemplo siguiente es el **DC0**.

```
root@node0:/proc/dvs# ls -l
total 0
-r--r--r-- 1 root root 0 Apr 11 06:49 info
-r--r--r-- 1 root root 0 Apr 11 06:49 nodes
dr-xr-xr-x 2 root root 0 Apr 11 06:49 proxies
dr-xr-xr-x 2 root root 0 Apr 11 06:49 DC0
```

En el archivo *info*, se presenta información del *DVS*: el ID del nodo y el contenido de la estructura *dvs_usr*. En este caso, el *DVS* puede constituirse con 32 nodos y contener hasta 32 *DCs*.

```
root@node0:/proc/dvs# cat info
nodeid=0
nr_dcs=32
nr_nodes=32
max_nr_procs=221
max_nr_tasks=35
max_sys_procs=64
max_copy_buf=65536
max_copy_len=1048576
dbglvl=FFFFFFF
version=2.1
```

En el archivo *nodes* se presenta información de estado y estadística de los nodos. Además, se incluye información que representa el mapeo de *DCs* a nodos en el *DVS* (se indica con una **X** en la columna correspondiente). En el siguiente ejemplo, el *DVS* está constituido por los nodos: *node0*, *node1*, *node2*, y el *DC* con ID=0 puede expandirse a los tres nodos.

```
root@node0:/proc/dvs# cat nodes
ID Flags Proxies -pxsent- -pxrcvd- 10987654321098765432109876543210 Name
0      6      -1        0          0 -----X node0
1      2        1         0          0 -----X node1
2      2        2         0          0 -----X node2
```

En el directorio *proxies* se encuentra la información de estado y de configuración de los *proxies*. En el archivo *info* presenta información sobre los *proxies*.

```
root@node0:/proc/dvs/proxies# cat info
Proxies Flags Sender Receiver --Proxies_Name- 10987654321098765432109876543210
1      1      2120      2121      px_node1 -----X-
2      1      2123      2124      px_node2 -----X--
```

En el ejemplo hay dos *proxies*:

- **px_node1**: *proxies* con ID=1, que comunican al *node0* (el nodo local) con el *node1*, conformado por dos procesos, el proxy Emisor con PID=2120 y el proxy Receptor con PID=2121.
- **px_node2**: *proxies* con ID=2, que comunican al *node0* con el *node2*, conformado por dos procesos, el proxy Emisor con PID=2123 y el proxy Receptor con PID=2124.

También se representa un mapeo entre *proxies* y nodos, es decir que indica cuáles *proxies* se utilizan para comunicar con un nodo dado, señalándolo con una **X** en la columna correspondiente.

En el archivo *procs* se presenta información de estado y estadística de los *proxies*, indicando el tipo (si es Emisor o Receptor) y el nombre del programa que realiza esa función.

```
root@node0:/proc/dvs/proxies# cat procs
ID Type -lpid- -flag- -misc- -pxsent- -pxrcvd- -getf- -sendt -wmig- name
1 send 2120 0 1 0 0 27342 27342 27342 tcp_proxy
```

1	recv	2121	0	1	0	0	27342	27342	27342	tcp_proxy
2	send	2123	0	1	0	0	27342	27342	27342	tcp_proxy
2	recv	2124	0	1	0	0	27342	27342	27342	tcp_proxy

Cuando se inicializa un DC, se crea debajo del directorio `/proc/dvs` un subdirectorio con el nombre del DC, en el siguiente ejemplo el nombre es **DC0**. En ese subdirectorio se crean 3 archivos: **info**, **procs** y **stats**.

```
root@node0:/proc/dvs/DC0# ls -l
total 0
-r--r--r-- 1 root root 0 Apr 11 06:57 info
-r--r--r-- 1 root root 0 Apr 11 06:57 procs
-r--r--r-- 1 root root 0 Apr 11 06:57 stats
```

En el archivo **info**, se encuentra la configuración del DC, básicamente la que se encuentra en la estructura de datos del módulo del DVK `dc_usr`. Además, se representa el mapeo entre nodos y el DC, es decir indica cuáles son los nodos a los que puede expandirse el DC (señalándolo con una **X** en la columna correspondiente).

```
root@node0:/proc/dvs/DC0# cat info
dcid=0
flags=0
nr_procs=221
nr_tasks=35
nr_sysprocs=64
nr_nodes=32
dc_nodes=3
warn2proc=0
warnmsg=1
dc_name=DC0
nodes 33222222222211111111111000000000
      10987654321098765432109876543210
      -----XXX
cpumask=ff
```

En el archivo **procs** se encuentra toda la información de los procesos del DC registrados en el módulo local del DVK. En la lista se incluyen tanto procesos Locales como Remotos, de tipo Respaldo o de tipo Réplica. En el caso de los procesos tipo Remoto, se identifica en la columna **node** el nodo donde está ejecutando. En caso de procesos tipo Respaldo la columna **node** indica el nodo donde ejecuta el proceso Primario. En el ejemplo siguiente, como el comando se ejecutó en `node0`, y todos los procesos tienen `node=0`, esto significa que todos los procesos son Locales. El tipo de proceso se encuentra codificado en el valor de la columna **misc**.

```
root@node0:/proc/dvs/DC0# cat procs
DC p_nr -endp- -lpid- node flag misc -getf- -sndt- -wmig- -prxy- name
0 -34 -34 2129 0 0 80 27342 27342 27342 27342 systask
0 -2 -2 2128 0 8 20 31438 27342 27342 27342 systask
0 0 0 2131 0 8 A0 31438 27342 27342 27342 pm
0 1 1 2202 0 8 20 31438 27342 27342 27342 fs
0 2 2 2134 0 0 20 27342 27342 27342 27342 rs
0 6 6 2168 0 8 20 31438 27342 27342 27342 eth
0 9 9 2195 0 8 20 31438 27342 27342 27342 inet
```

En el archivo *stats* se presenta información estadística referente a las transferencias de mensajes y copias de datos de cada uno de los procesos del *DC* registrados en el módulo local del *DVK*.

```

root@node0:/proc/dvs/DC0# cat stats
DC p_nr -endp- -lpid- node --lsnt-- --rsnt-- -lcopy-- -rcopy--
0 -34 -34 2129 0 0 0 0 0
0 -2 -2 2128 0 54 0 35 0
0 0 0 2131 0 43 0 0 0
0 1 1 2202 0 7 0 0 0
0 2 2 2134 0 8 0 0 0
0 6 6 2168 0 14 0 1 0
0 9 9 2195 0 15 0 0 0

```

Se distinguen entre transferencias locales y remotas de mensajes, así como copias de bloques de datos entre procesos locales o entre el proceso local y procesos remotos, porque estos datos resultan de utilidad para evaluar la relocalización de procesos por su afinidad.

Toda la información presentada acerca del *DVS* y sus componentes en el sistema de archivos */proc* solo considera la información contenida en el módulo local del *DVK*.

4.1.10. Sistema de Comunicaciones Grupales (GCS)

Para la implementación de un VOS distribuido y un Gestor de Disco Virtual (VDD: Virtual Disk Driver) se requería un mecanismo de difusión tolerante a fallos que disponga de diferentes alternativas de ordenamiento en la entrega de mensajes. Además, debía soportar fallos de procesos, de nodos, particiones de red total y notificaciones por cambios en la membresía de los grupos. En su momento se analizaron desde diferentes aspectos varias alternativas (ISIS, Horus, ISIS2, Corosync, Paxos, QuickSilver, Spread Toolkit) resultando finalmente la elección de Spread Toolkit [110] por su madurez, adecuado rendimiento [111] y abundante documentación, frente a otros GCS tales como Ring Paxos [127] en el que aún permanecen algunas características sin implementar, o QuickSilver [128], que solo está disponible para plataforma Windows o Corosync [109] que no estaba en ese momento completamente documentado.

El Spread Toolkit es además una herramienta adecuada que puede utilizarse en múltiples aplicaciones distribuidas que requieren alta disponibilidad, alto desempeño, escalabilidad y comunicaciones confiables entre los miembros de un grupo [129]. Spread Toolkit soporta el modelo de membresía de Sincronía Virtual Extendida (EVS) [130]. EVS puede manejar particiones de red y su posterior recomposición, como así también desconexiones (controladas o por fallos) de los

miembros e incorporación de nuevos miembros al grupo. Las APIs del Spread Toolkit que fueron más utilizadas en el prototipo son:

SP_connect	Crea una conexión a la red Spread
SP_disconnect	Desconecta una conexión de la red Spread
SP_join	Hace que un proceso se una a un grupo
SP_leave	Hace que un proceso abandone un grupo
SP_multicast	Envía un mensaje de difusión. Se puede elegir el orden de entrega del mensaje.
SP_receive	Recibe un mensaje. El mensaje puede ser uno enviado desde otro proceso o puede ser un mensaje del GCS indicando un fallo de un proceso, de un nodo, un cambio de membresía en un grupo o una partición o re-uniión de red.
SP_get_memb_info	Permite extraer la información de un mensaje de cambio de membresía
SP_get_vs_set_members	Permite extraer información de los miembros de un grupo de un mensaje de membresía.

Una de las características más interesantes del Spread Toolkit es que soporta miles de grupos diferentes con conjuntos de miembros diferentes. Esta característica se aprovechó para crear grupos cuya identificación está constituida por la dupla: $\{DC, aplicación\}$ para asociar un *DC* a cada grupo. Por ejemplo, si en el DC4 se utilizará Spread Toolkit para el algoritmo de replicación del VDD y también para un algoritmo de asignación de recursos denominado SLOTS, el nombre del grupo de esos grupos sería “DC4-VDD” y “DC4-SLOTS” respectivamente. De esa forma, cada proceso podrá enviar mensajes o recibir solo de/a miembros de su grupo.

Claramente, esta separación de grupos no constituye en sí misma un mecanismo de confinamiento, dado que cualquier proceso podría incorporarse al grupo registrándose en el GCS. Para que puedan aislarse los grupos en el *DVS*, el acceso a las APIs del GCS debería hacerse a través del módulo local del *DVK*, quien previamente a incluir a un proceso en un grupo dado, debe verificar su pertenencia a un *DC*, y el nombre del grupo lo establecería el *DVK*. Se deja esta implementación para trabajos futuros.

4.2. Módulo *Webmin* para Gestión del *DVS*

Webmin [131] es una herramienta con interfaz basada en web para administración de sistemas para Unix (Fig. 32). Utilizando un navegador web, le permite al administrador configurar distintos aspectos del sistema: gestionar cuentas de accesos de usuarios, levantar servicios como *Apache*,

establecer la configuración de red, compartir archivos y obtener otros datos relevantes a la hora de realizar tareas sobre el sistema Linux.

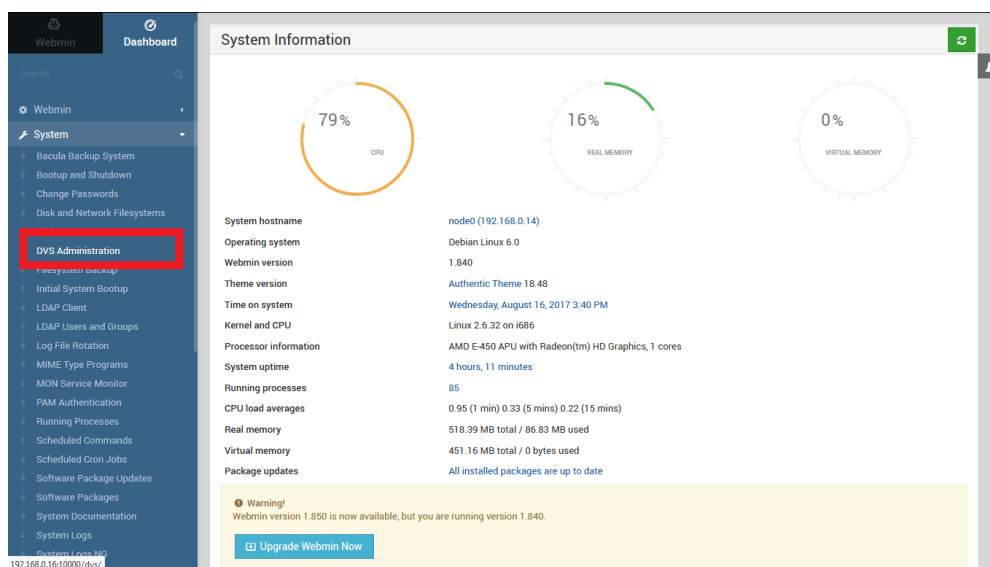


Figura 32. Menú Principal de Webmin.

Webmin elimina la necesidad de editar manualmente archivos de configuración de Unix y permite administrar servidores Linux de forma totalmente gráfica y desde el navegador web. Una de sus características más interesantes es que es un sistema modular que permite fácilmente añadir nuevas funcionalidades. Además, ofrece un conjunto de APIs para desarrolladores que facilita la creación de módulos a medida mediante funciones predefinidas.

Se decidió utilizar *Webmin* para la gestión gráfica del *DVS* ^{Nota2} ya que es una solución conocida, desarrollada y probada. Para ello, se desarrolló un módulo propio [132] de modo que se ajuste a los componentes propios de un *DVS*. Para poder realizar la interfaz de administración, se comenzó por crear un módulo de monitoreo en *Webmin*, que agrupa la información del *DVS* en distintos apartados (Fig. 33).

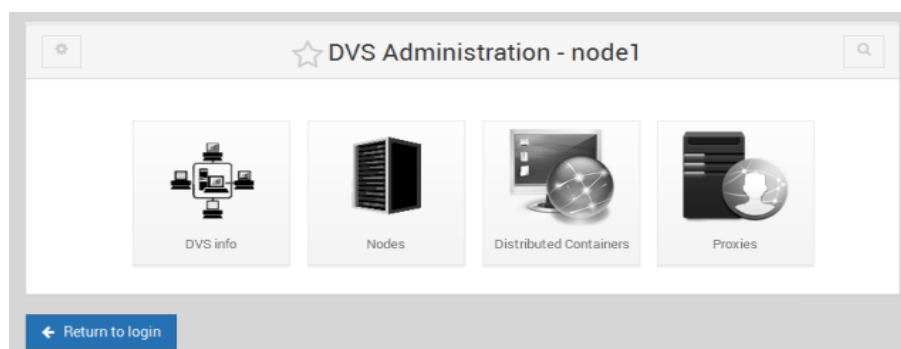


Figura 33. Menú principal de la interfaz de administración de un *DVS*.

En la parte superior se visualiza el nombre del nodo del *DVS* sobre el cual se ha establecido la conexión, se pueden visualizar los nodos que se encuentran activos en el *DVS*, el estado de los *proxies* en el nodo y la finalmente los diferentes *DCs* definidos en el nodo.

Existen dos formas básicas en la que se puede acceder a la información de estado del *DVS* y sus componentes: Utilizando las APIs tipo *mnx_getxxxxx()* para obtener la configuración, estado y estadísticas de los componentes, o utilizar el */proc/dvs* donde se encuentra la misma información. Como *Webmin* utiliza lenguaje Perl para el desarrollo de módulos y éste tiene un muy buen manejo de cadenas de caracteres y procesamiento de texto en general, se decidió utilizar la opción de */proc/dvs*.

Para obtener los datos de los diferentes nodos remotos se utilizó el protocolo SSH. Se recogen datos de los archivos del directorio */proc/dvs* y de sus subdirectorios, para luego procesarlos con Perl y presentar los resultados en la interfaz de administración web. Para poder acceder a cada uno de los nodos remotos mediante SSH, se utilizó el comando *sshpass* junto con un script *bash*. La dirección IP de cada nodo se obtiene del archivo de configuración */etc/hosts* del servidor en que se encuentra ejecutando *Webmin*.

Todos los comandos utilizados para leer los archivos de estado y de configuración propios del *DVS* son ejecutados vía SSH. Esto requiere que sean almacenadas las credenciales de acceso en el servidor para mantener la conexión con el nodo que se está administrando. La utilización de la biblioteca Perl denominada *CGI::Session*, permite el almacenamiento de la sesión del usuario del lado del servidor de forma elegante; y gracias al manejo de *cookies* únicas en el navegador web, los usuarios pueden mantener más de una conexión a los nodos del *DVS* para administrarlos.

La información de configuración y estado del *DVS* se obtiene del archivo */proc/dvs/info*. El módulo *Webmin* accede a este y demás archivos del *DVS*, ejecutando el comando *cat* mediante SSH. La información es procesada y presentada en forma tabular en una página en HTML utilizando las bibliotecas que ofrece *Webmin* para desarrolladores (Fig. 34).

Desde el menú principal también se ofrece acceso a la información de los *DCs* que se encuentran ejecutando en los nodos, como así también a la información específica de cada uno de estos.

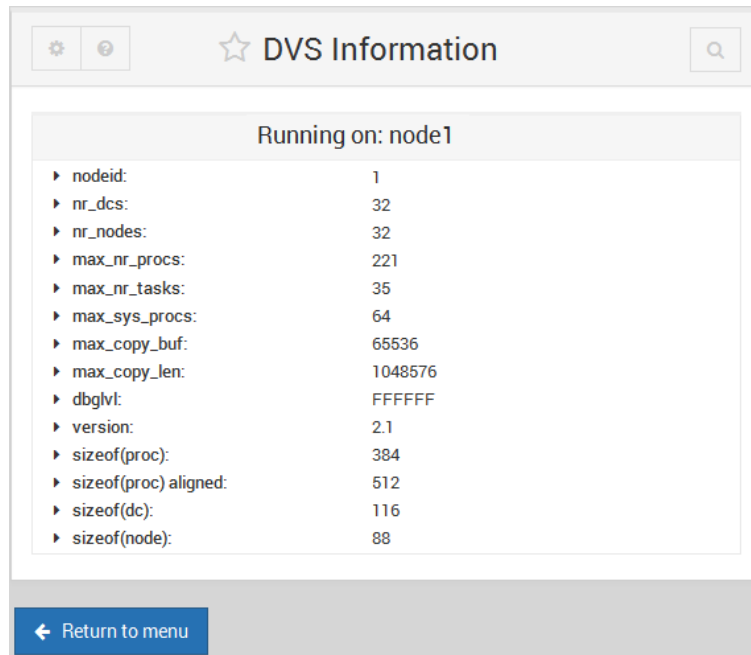


Figura 34. Presentación la información de estado del *DVS* en el nodo.

Para poder visualizar el estado de los *proxies* que se encuentran activos en el *DVS* (Fig. 35), se necesita acceder a 2 archivos: */proc/dvs/proxies/info* de donde se obtienen los nombres de los *proxies* ejecutando en el nodo, con sus datos, estado de la conexión y nodos que representan, y del archivo */proc/dvs/proxies/procs* de donde se obtiene el estado de los procesos *proxies*.

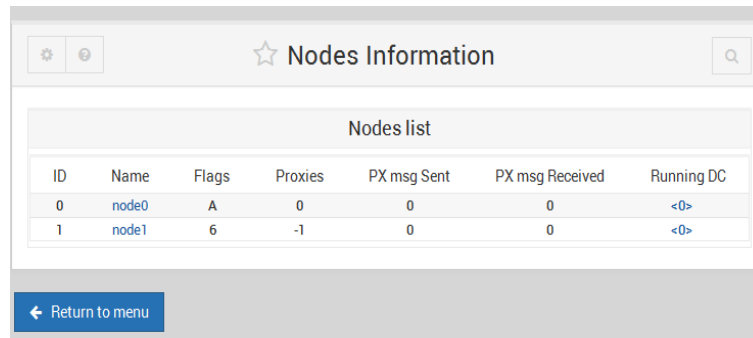
The screenshot shows a window titled 'Proxies Information' with a search icon and a settings icon. Below the title bar, it says 'Proxies list'. A table is displayed with the following columns: Proxy, Proxies_Name, Flags, Sender, Receiver, Running on nodes. Below the table, there is a section titled 'Process' with a table of process details.

Proxy	Proxies_Name	Flags	Sender	Receiver	Running on nodes					
Process										
Id	name	Type	lpid	Flag	Misc	Pxsent	Pxrcvd	Getf	Sendt	Wmig
Type		-lpid-	-flag-	-misc-	-pxsent-	-pxrcvd-	-getf-	-sendt	-wmig-	name
0	tcp_proxy	send	2777	8	3	0	0	31438	27342	27342
0	tcp_proxy	recv	2778	0	1	0	0	27342	27342	27342

At the bottom, there is a blue button labeled 'Return to menu'.

Figura 35. Presentación de información de *proxies* ejecutándose en un nodo.

De forma similar, se presentan al usuario los nodos que se encuentran activos en el *DVS* (Fig. 36). Esta información es obtenida del archivo */proc/dvs/nodes*.



ID	Name	Flags	Proxies	PX msg Sent	PX msg Received	Running DC
0	node0	A	0	0	0	<0>
1	node1	6	-1	0	0	<0>

Figura 36. Nodos activos en el DVS.

El tercer componente de una interfaz presenta información acerca de los *DCs* que se están ejecutando en el nodo del *DVS* (Fig. 37).



DC0

- ▶ id: 0
- ▶ flags: 0
- ▶ nr_procs: 221
- ▶ nr_tasks: 35
- ▶ nr_sysprocs: 64
- ▶ nr_nodes: 32
- ▶ dc_nodes: 3
- ▶ warn2proc: 0
- ▶ warnmsg: 1
- ▶ dc_name: DC0
- ▶ Running in nodes: <1> <0>
- ▶ cpumask: ff

Process

DC	p_nr	endp	lpid	node	flag	misc	getf	sndt	wmig	prxy	name
0	-34	-34	5064	1	0	80	27342	27342	27342	27342	systask
0	-3	-3	5062	1	8	20	31438	27342	27342	27342	systask

Stats

DC	p_nr	endp	lpid	node	lcnt	rsnt/b>	lcopy	rcopy
0	-34	-34	5064	1	0	0	0	0
0	-3	-3	5062	1	0	0	0	0

Figura 37. Presentación de información de *DC* y sus procesos

Dentro de los entornos de los *DCs* se ejecutan procesos, y como los *DCs* son intrínsecamente distribuidos, sus procesos pueden estar dispersos en varios nodos. Cada *DC* tiene sus archivos dentro de un directorio propio (con su nombre) bajo el directorio */proc/dvs/*, y dentro de éste contiene su información de configuración, de estado y la lista de los procesos que lo componen. La interfaz desarrollada permite ver todos los procesos en ejecución dentro de un *DC* y los nodos en los que se distribuye el *DC*.

Se plantea como objetivos para una etapa futura del proyecto, la utilización de archivos de configuración como respaldo de las configuraciones realizadas por el administrador del *DVS* y la ejecución de comandos coordinados en los distintos nodos. La generación y modificación de archivos de configuración utilizados por el *DVS* permitirá gestionar de forma gráfica a través de un navegador

web sus componentes, tal como iniciar un nuevo *DC* en forma automática solicitando al administrador los parámetros necesarios mediante la interfaz web.

Actualmente, muchas plataformas de virtualización utilizan para sus sistemas de gestión propios. Aun así, la tendencia muestra que la mayoría de estas plataformas se integran a *Openstack* [133]. Como trabajo de más largo plazo se propone el desarrollo de las APIs que permitan integrar el DVS a ese sistema de gestión.

4.3. Contenedor Distribuido (DC)

Los *DCs* son dominios aislados para ejecutar VOS y aplicaciones que pueden extenderse a más de un nodo de un cluster DVS. Un *DC* está constituido por un conjunto de mecanismos dependientes del *DVK*. En el caso del prototipo de DVS se pueden distinguir entre tres mecanismos, estos son:

- 1) *DCs de M3-IPC*: Como M3-IPC (embebido en el *DVK*) permite comunicar procesos localizados en el mismo nodo o en diferentes nodos en forma transparente, el *DVK* limita las comunicaciones solo a procesos que pertenezcan al mismo *DC*.
- 2) *Contenedores de Linux*: Son a nivel de OS-host. Un conjunto de contenedores Linux individuales conforman un *DC*. El sistema de Contenedores de Linux permite configurar los espacios de nombres (*Namespaces*) y los límites en la utilización de recursos (*Cgroups*) de las aplicaciones que se ejecutan en el dominio de un Contenedor.
- 3) *Grupo de Difusión*: El Spread Toolkit permite conformar múltiples grupos de difusión. Los procesos pertenecientes a un grupo de difusión dentro de un *DC* no deben poder comunicarse con procesos de otro *DC*.

En el prototipo actual, estos tres componentes no se encuentran integrados en el *DVK* sino que se gestionan por separado. Por ejemplo, para la creación de un *DC* se deben realizar operaciones específicas para cada uno de estos componentes, las que actualmente se efectúan mediante *scripts* de Linux.

Se prevé como trabajo futuro la incorporación al kernel de Linux de un nuevo subsistema “*dvs*” en *Cgroups* que implemente un controlador de la utilización de recursos de cada *DC*. Por

ejemplo, que limite la tasa de mensajes por unidad de tiempo (Locales y Remotos por separado) de cada Contenedor que conforma un *DC*.

4.4. Sistemas Operativos Virtuales (VOS)

Para probar la factibilidad en la implementación de un VOS para el *DVS*, se desarrollaron dos VOS: Un VOS multiservidor basado en Minix 3 [126] que puede ejecutarse en forma distribuida y un VOS unikernel basado en *LwIP* [134]. A continuación, se describen detalles de su diseño e implementación.

4.4.1. VOS Multiservidor

El VOS multiservidor para ejecutar en el *DVS* es una versión de MoL [55] que se adaptó a las APIs de IPC del *DVK* y a trabajar como un VOS distribuido con las siguientes características:

- *PIDs globales*: cada uno de los procesos que se ejecuta en el VOS tendrá su propio PID independientemente del nodo en donde se localiza o si el mismo fue migrado de un nodo origen a uno destino.
- *Ejecución Remota de Procesos*: Permite ejecutar *rexec()* en cualquiera de los nodos para ejecutar un proceso en otro nodo.
- *Soporte de Migración*: Un proceso puede migrar de un nodo al otro basado en el nivel de carga del cluster o por un comando del administrador.
- *ps* global: Un comando *ps* permite visualizar todos los procesos del VOS.
- *init* único: Al igual que un sistema centralizado, el proceso *init* se inicia en un nodo, pero tiene acceso realizar operaciones en todos los nodos. La versión actual no contempla replicación Pasiva, pero fácilmente se puede implementar para tolerar fallos de *init* o su nodo.

Como descendiente de Minix, MoL descompone al OS en múltiples servidores. Minix es un OS completo, de propósito general, tiempo compartido, multitarea y basado en microkernel desarrollado desde cero por Andrew S. Tanenbaum [126]. Es muy utilizado en cursos de grado de OS

y aunque está protegido legalmente por derechos, es ampliamente en universidades, en investigación, incluso dentro de las CPUs Intel [135]. Sus principales características son:

- *Basado en Microkernel*: El microkernel provee gestión de procesos y planificación a bajo nivel, gestión básica de memoria, mecanismos de IPC, gestión de interrupciones y soporte para Entrada/Salida de bajo nivel.
- *Sistema Multicapa*: Permite una implementación más modular y limpia para el agregado de nuevas características.
- *Modelo Cliente/Servidor*: Los servicios del sistema y los gestores de dispositivos se implementan como procesos servidores aislados en sus propios entornos de ejecución.
- *IPC por transferencias de mensajes*: Se utiliza para la sincronización de procesos y para las transferencias de datos entre procesos.
- *Ocultamiento de Interrupciones*: Las interrupciones se convierten en transferencias de mensajes.

Minix 3 fue diseñado para ser un OS confiable, flexible y seguro [136]. El microkernel de Minix y la denominada Tarea del Sistema (SYSTASK) ofrecen servicios a los servidores dado que son los únicos componentes con privilegios para acceder al hardware y a ejecutar instrucciones privilegiadas (Fig. 38).

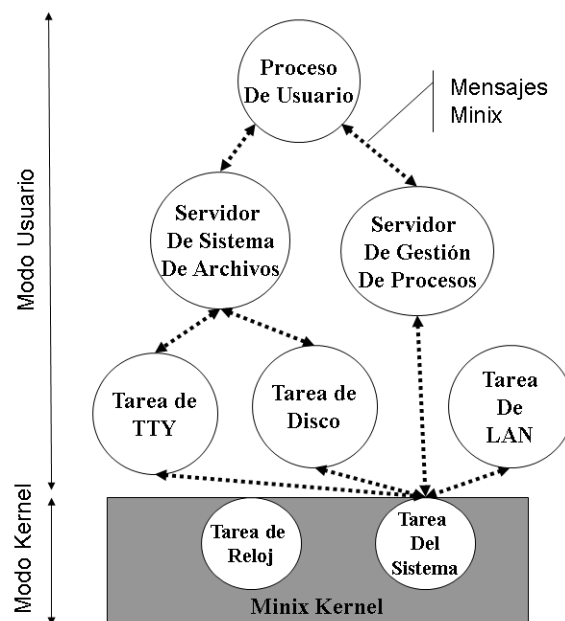


Figura 38. Arquitectura de Minix 3 [126].

En Minix 3 se distinguen 4 capas de procesos. La capa inferior la constituye el microkernel que comparte su espacio de direccionamiento y privilegios con la SYSTASK y la tarea del reloj denominado CLOCK. Los procesos de las 3 capas siguientes ejecutan en modo usuario. Las Tareas son procesos Gestores de Dispositivos al que le sigue la capa de Servidores, de las cuales algunos de ellos ofrecen servicios a la capa de Aplicaciones como Llamadas al Sistema.

MoL, con arquitectura similar a Minix 3, se puede ejecutar con todos sus componentes en un único nodo del *DC* (Fig. 39) o sus componentes pueden expandirse a múltiples nodos (Fig. 40).

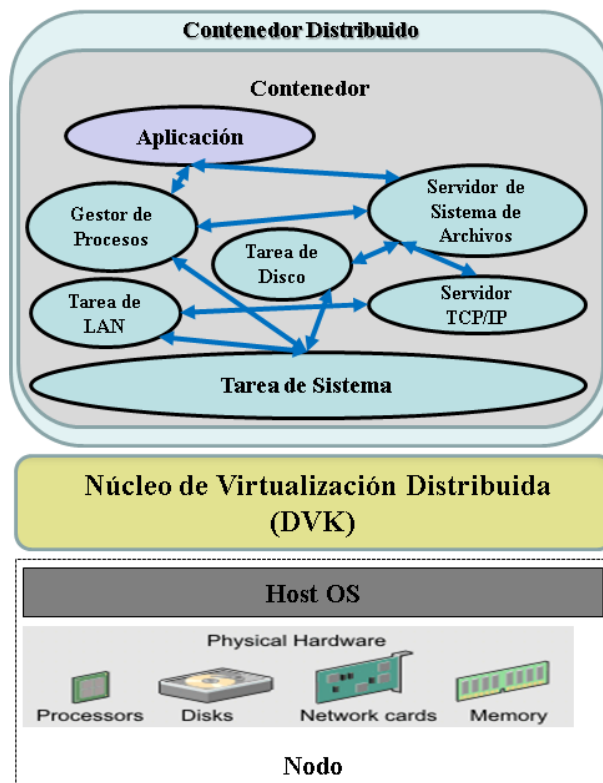


Figura 39. Arquitectura de MoL en un único nodo del DVS.

Muchas de las funciones del microkernel de Minix son las provistas por el *DVK*. Por otro lado, se le otorgaron privilegios a las Tareas de tal modo que éstas puedan tener accesos a los servicios del OS-host (Linux). Por ejemplo, la Tarea de Disco (VDD) requiere acceder a los archivos de Linux que contienen las imágenes de disco. La Tarea de LAN necesita poder acceder al dispositivo TAP de Linux. Este otorgamiento de privilegios a las Tareas reduce la seguridad, pero simplifica la implementación del sistema.

En el caso de requerir mayor seguridad, las funciones de acceso a los recursos del OS-host deben incorporarse al *DVK* quien puede verificar los privilegios entre las tareas y los recursos que estas requieren acceder.

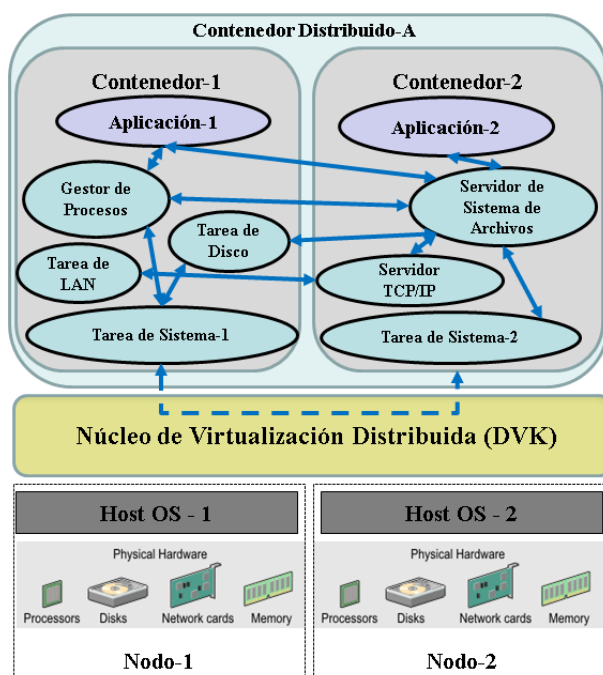


Figura 40. Arquitectura de MoL distribuido en el *DVS*.

❑ Tarea de Sistema (SYSTASK)

Para cuando MoL trabaja distribuido, se instala una SYSTASK en cada uno de los nodos del *DC* y se ejecutan 2 hilos. Uno de los hilos, al que se lo denomina SLOTS, se registra con el mismo *endpoint* en todos los nodos como tipo Réplica. En cada uno de los nodos se registra el otro hilo (denominado SYSTASK) con su propio *endpoint* que depende del ID del nodo. Por ejemplo, en el nodo con ID=0 se registra la SYSTASK con *endpoint*=(-2), en el nodo con ID=1 se registra el hilo SYSTASK con *endpoint*=(-3), y así sucesivamente. La razón por la que cada una de las SYSTASK tiene su propio *endpoint* es porque desde un nodo A se pueden solicitar servicios a la SYSTASK del nodo B. Si ambas SYSTASK tuviesen el mismo *endpoint*, el pedido que parte del nodo A lo haría sobre la SYSTASK del nodo A. Esta forma de trabajar se podría cambiar de tal forma que todo pedido de un proceso a la SYSTASK sea sobre el hilo de su propio nodo y que éste, mediante el GCS, se lo comunique a la SYSTASK del nodo correspondiente. Esta posibilidad fue evaluada, pero sometería al sistema a una mayor sobrecarga por la utilización innecesaria del GCS. En la solución elegida se pierde elegancia, pero se gana en rendimiento y eficiencia.

El hilo SLOTS tiene varias funciones, la principal es un algoritmo de asignación de recursos distribuidos que se utiliza para asignar a cada uno de los nodos la propiedad de los descriptores de procesos. En la [Sección 4.4.3](#) se describe en detalle este algoritmo. Otra función de SLOTS es

difundir entre los diferentes nodos componentes la registraci3n o de-registraci3n de procesos privilegiados (tareas y servidores), de tal forma que cada vez que se registra una nueva tarea o servidor en un nodo, esto se da a conocer en el resto de los nodos y todos ellos registran al *endpoint* de la nueva tarea o servidor como *endpoint* tipo Remoto en forma autom1tica.

Como el GCS dispone de detectores de fallos, cuando se produce un fallo de ca3da de un nodo, todos sus procesos servidores y tareas deben de-registrarse.

❑ Gestor de Procesos (MoL-PM)

El Gestor de Procesos de MoL es el servidor responsable de las Llamadas al Sistema que permiten crear (*fork*) y finalizar (*exit*) procesos, ejecutar programas (*exec*), gestionar se1ales (*kill*), etc. En su versi3n actual no se encuentra replicado.

❑ Sistema de Archivos (MoL-FS)

El Servidor de Sistema de Archivo de MoL [137] es responsable de las Llamadas al Sistema que permiten montar y desmontar sistemas de archivos (*mount/umount*), crear y suprimir directorios (*mkdir/rmdir*), y operar sobre archivos (*open, read, write, close*). No hay diferencias sustantivas entre el c3digo del FS de Minix y MoL-FS, pero existen diferencias en cuanto a sus posibilidades. MoL-FS puede operar en diferentes escenarios (Fig. 41). Puede compartir el mismo nodo que sus procesos Clientes (escenarios A1, A2, B) o pueden estar en diferentes nodos (escenarios C1, C2, D), puede utilizar una imagen de disco del OS-host Linux (escenarios A1, C1), o puede utilizar servicios de almacenamiento de un VDD (escenarios A2, B, C2, D), el cual puede localizarse en el mismo nodo (escenarios A2, C2) o en otro nodo (escenarios B, D).

Al solo efecto de comprobar su rendimiento y como c3digo de ejemplo para realizar interfaces con MoL, se desarroll3 un m3dulo FUSE [138] que le permite a cualquier proceso ordinario Linux (por ejemplo: *mount, cat, more, grep*, etc.) utilizar los archivos de MoL-FS.

Como para comprobar la versatilidad del dise1o modular del MoL, se desarroll3 un nuevo Servidor de Sistema de Archivos tipo FAT, denominado MoL-FAT, tomando como base MoL-FS, utilizando la biblioteca FatFs [139] y desarrollando tambi3n para ella el m3dulo FUSE correspondiente. En este caso, el acceso al dispositivo por parte de la biblioteca FatFs se realiz3 utilizando el equivalente a un RPC con M3-IPC a un MoL-VDD. Tanto MoL-FS como MoL-FAT pueden convivir dentro del mismo VOS.

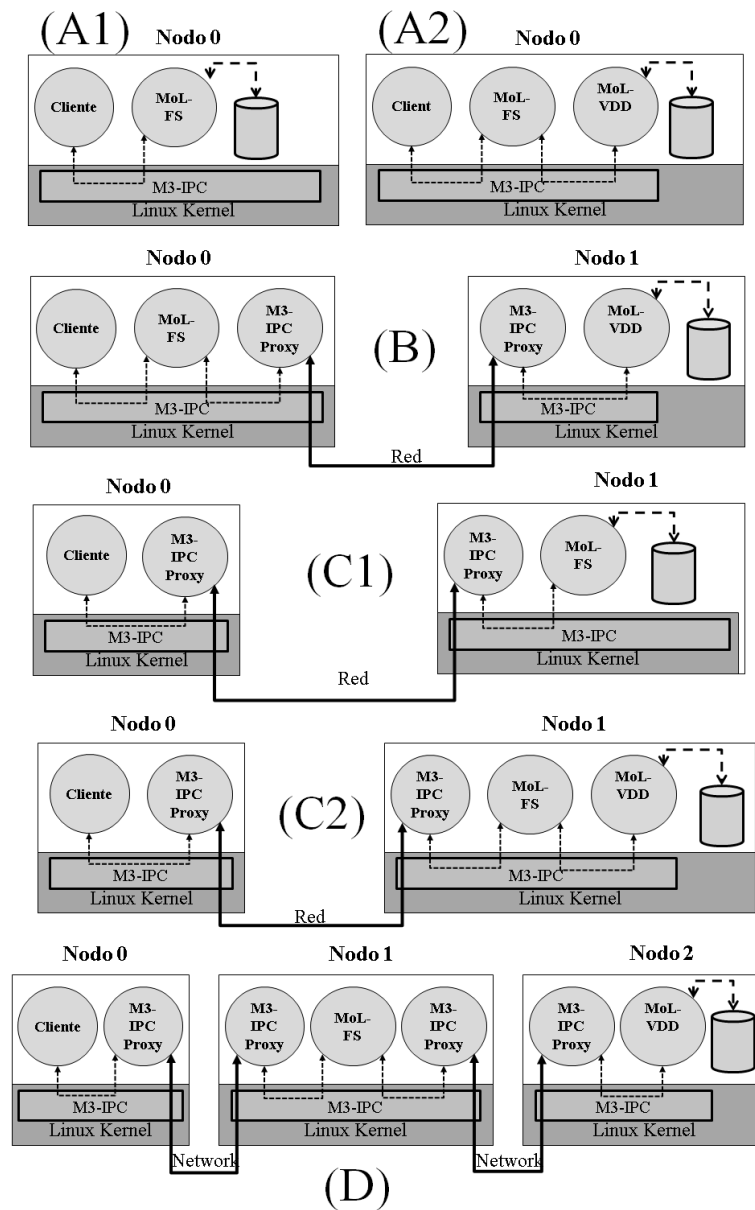


Figura 41. Escenarios de Operación de MoL-FS.

❑ Servidor de Datos (MoL-DS)

El Servidor de Datos de MoL permite publicar datos en formato clave/valor, para luego poder recuperar el valor los datos suministrando la clave. No hay diferencias sustantivas entre el DS de Minix y MoL-DS.

❑ Servidor de Información (MoL-IS)

El Servidor de Información de MoL permite presentar en la pantalla los datos del estado de los procesos y de otros componentes del sistema. Difiere del IS de Minix en cuanto a la información presentada dado que MoL es un VOS mientras que Minix es un OS completo.

❑ Servidor de Internet (MoL-INET)

El Servidor de Internet de MoL implementa toda la pila de protocolo TCP/IP. No difiere de la implementación del servidor INET de Minix porque MoL-INET resulta de portar su código fuente a la nueva infraestructura (*DVS/DVK*) de éste.

❑ Tarea de LAN Ethernet (MoL-ETH)

La Tarea de LAN Ethernet de MoL es responsable de gestionar la interface del dispositivo Virtual de Red (vNIC). MoL-ETH no trabaja sobre un dispositivo real como la tarea ETH de Minix, sino que lo hace sobre un dispositivo virtual tipo TAP del que dispone el OS-host Linux.

❑ Tarea de Disco Virtual Replicado (MoL-VDD)

La Tarea de Disco Virtual Replicado de MoL es responsable de gestionar los discos virtuales. Inicialmente, MoL-VDD surge del código fuente del Gestor de Dispositivo de disco RAM de Minix. Luego ese código evoluciona implementando replicación Primario/Respaldo y permitiendo gestionar diferentes tipos de dispositivos (discos RAM, imágenes en archivos regulares de Linux, archivos de dispositivos, etc.). Generalmente, MoL-VDD trabajará sobre imágenes de disco almacenadas en archivos regulares del OS-host Linux. MoL-VDD soporta la replicación de dispositivos ([Sección 4.4.4](#)) y la sincronización de datos de nuevos servidores de replicación ([Sección 4.4.5](#)).

❑ Servidor Web

Se implementó un servidor Web basado en el código de *nweb* [140] como aplicación típica de un servidor de Internet. Este servidor web fue modificado de tal forma que pueda trabajar sobre páginas estáticas embebidas en su código, sobre archivos regulares del OS-host Linux o sobre archivos suministrados por MoL-FS. Por las características de MoL, como VOS distribuido, el Web Server puede compartir o no el mismo nodo que el MoL-FS.

❑ Tareas y Servidores Pendientes

Queda pendiente de desarrollo el gestor de terminales (MoL-TTY) que utilice las terminales virtuales (*vtY*) disponibles en el OS-host Linux. El código del INET de Minix 3 original (adaptado

luego a MoL), fue reemplazado en la actualidad por la pila de protocolos TCP/IP de *LwIP*. A futuro se pretende portar el código fuente de *LwIP* a MoL.

❑ Confinamiento de Procesos

Uno de los principales desafíos en la Virtualización de OS es lograr el confinamiento de los procesos. En general (y en Linux en particular) los procesos interactúan con el OS a través de los siguientes mecanismos (Fig. 42):

- *Llamadas al Sistema*: El proceso puede operar sobre archivos, semáforos, colas, etc. solicitando servicios al OS a través de Llamadas al Sistema o a través de bibliotecas que las utilizan.
- *Señales*: Son los mecanismos que el OS utiliza para notificar eventos a los procesos.
- *Memoria Compartida*: Es un artificio del OS para que dos o más procesos puedan compartir una misma región de memoria. Como la memoria compartida es parte del espacio de direcciones de los procesos, no hay intervención por parte del kernel.

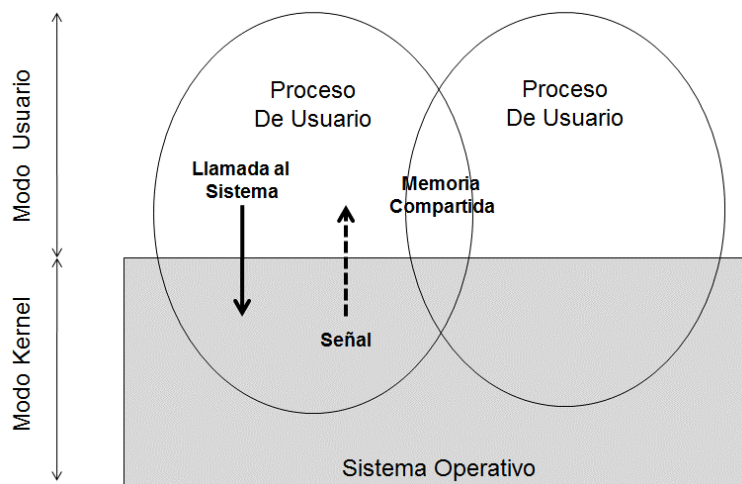


Figura 42. Interacciones entre Procesos y Sistema Operativo

Como Minix no soporta memoria compartida en su distribución ordinaria (aunque existe una implementación experimental de esta facilidad) MoL no soporta memoria compartida. Por ello, para emular la ejecución de un entorno Minix solo se requiere interceptar las Llamadas al Sistema y darles tratamiento correcto a las señales.

Para que un proceso solo pueda interactuar con MoL, se instala una coraza (Fig. 43) a modo de blindaje que atrapa las Llamadas al Sistema dirigidas a Linux y las redirecciona a MoL conformando el equivalente a un vPROC [95]. Las Llamadas al Sistema a MoL, no utilizan trampas o compuertas, sino que al igual que Minix, utilizan transferencias de mensajes.

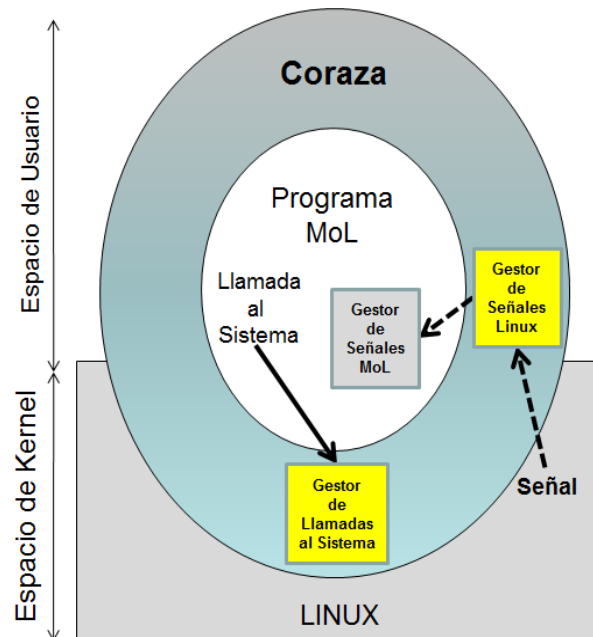


Figura 43. Coraza de Aislamiento de un Proceso.

La coraza atrapa las señales que Linux le envía al proceso confinado y, dependiendo el tipo de señal y del tipo de máscara de señales que el proceso haya configurado, se invocará al gestor de señales del proceso, se ignorará o se informará al MoL-PM utilizando transferencia de mensajes.

Actualmente, las Llamadas al Sistema del proceso confinado son interceptadas a nivel de biblioteca utilizando la variable de entorno `LD_PRELOAD` de Linux, para ser reemplazadas por sus contrapartes de MoL.

La variable de entorno `LD_PRELOAD` permite reemplazar las Llamadas al Sistema Linux que el proceso confinado invoca, por las equivalentes del VOS. No es posible atrapar con este método aquellas Llamadas al Sistema que no lo hacen a través de una biblioteca de enlace dinámico. Pero para ello, hay dos alternativas para realizar la interceptación: 1) utilizar la Llamada al Sistema `ptrace` tal como lo hace la primera versión de UML [51], 2) instalar un módulo de kernel que intercepte las Llamadas al Sistema Linux de procesos registrados en el `DVK` y las transforme en transferencia de mensajes. Se propone investigar ambas alternativas para trabajos futuros.

4.4.2. VOS Unikernel (ukVOS)

Para completar el prototipo básico del DVS se implementó un VOS de tipo Unikernel al que se lo denominó ukVOS. En ukVOS se delegan ciertas funciones de un kernel tradicional al `DVK` y al OS-host, tales como la planificación de hilos, los mecanismos de IPC, la gestión de memoria virtual o

la utilización de la Entrada/Salida estándar. Por otro lado, la vNIC utiliza TAP que es una abstracción de Linux. El servidor web no requiere utilizar Llamadas al Sistema de Linux porque consta de su propio sistema de archivos y pila de protocolo TCP/IP.

Se tomó como base para el desarrollo el software LwIP [134], que contiene toda la pila de protocolo TCP/IP, un gestor de interface de red y un servidor web de páginas estáticas en memoria. La versión modificada es aquella que utiliza TAP de Linux como vNIC.

El servidor web tiene tres versiones (Tabla XI) definidas en función de la fuente de las páginas web. En la versión 1, el servidor web obtiene sus archivos de un MoL-FS (el cual puede ser local o remoto) que se ejecuta dentro de un *DC*. A su vez, el MoL-FS puede operar sobre un archivo local de imagen de disco o utilizar los servicios de un MoL-VDD (local o remoto respecto a MoL-FS) dentro del mismo *DC*. En la versión 2, el servidor web obtiene sus archivos mediante el uso de la biblioteca FatFs que se obtiene los bloques de disco de un MoL-VDD, el cual puede ser local o remoto respecto al servidor web.

TABLA XI. VERSIONES DE UNIKERNELS EN EL PROTOTIPO.

Versión	Servidor Web	TCP/IP	NIC	Sistema de Archivos	Almacenamiento
1	nweb	LwIP	TAP	MoL-FS	Archivo de Imagen
2	nweb	LwIP	TAP	FatFs	MoL-VDD
3	nweb	LwIP	ETH	FatFs	Archivo de Imagen

A futuro se planea implementar un Unikernel utilizando *rumpkernel*[141] con soporte POSIX.

4.4.3. Algoritmo de Asignación de Ranuras (*SLOTS*)

Uno de los problemas que debe enfrentar un DOS, es la asignación de recursos en forma distribuida. En particular, para el caso de MoL distribuido, los recursos más importantes a asignar en forma excluyente son los descriptores de procesos. Cuando se crea un proceso (*fork()*), se le debe asignar un descriptor para que almacene información de entorno, de estado y estadística. Cada descriptor o ranura de proceso (ranura) se distingue por el *p_nr* (número de proceso).

Una posibilidad para la asignación del *p_nr*, es que una única SYSTASK (Tarea del Sistema) sea la responsable de la asignación de este recurso. Esta solución, si bien es sencilla (centralizada), resultaría en un sistema que no toleraría el fallo de esa SYSTASK. Además, por cada proceso que se

cree en cualquier nodo, debería solicitarle a esa SYSTASK la asignación del p_nr provocando un aumento de tráfico en la red, haciéndolo poco escalable.

Una solución distribuida podría asignar a cada SYSTASK (una en cada nodo) una cantidad fija de descriptores y que cada una de ellas los administre. También ésta es una solución sencilla, pero al ser estática no contemplaría las diferencias que puede haber de nodos con muchos procesos y nodos con pocos procesos.

Otra solución distribuida, pero dinámica, sería utilizar un mecanismo de exclusión mutua distribuida. La tabla con todos los descriptores de recursos sería protegida por este mecanismo. Todas las SYSTASK de todos los nodos tienen su propia réplica de la tabla de descriptores. Cuando una SYSTASK necesita crear un proceso, se solicita acceso excluyente a la tabla al resto de las SYSTASKs mediante un mensaje de difusión. Una vez obtenido el acceso se busca un descriptor libre, se lo marca como ocupado y se libera la tabla informándolo a todas las SYSTASKs mediante un mensaje de difusión. De igual forma sucede cuando se necesita liberar un descriptor por finalización de un proceso (*exit*). Como el acceso a los descriptores es excluyente, no hay problemas de condiciones de competencia. El problema de esta solución es que cada creación (*fork*) y cada finalización (*exit*) de procesos implica una difusión de mensajes con ordenamiento de tipo Orden Total o Atómico. Si la tasa de creación/finalización del VOS (considerando todos sus nodos) es alta, el mecanismo de difusión de mensajes requerido acota su escalabilidad y eleva la latencia de esas operaciones.

Para el diseño del algoritmo de asignación de descriptores/ranuras de procesos se establecieron los siguientes requerimientos, características y propiedades:

- *Seguro (safety [142, 143])*: Ningún recurso debe asignarse a más de un miembro a la vez (Exclusión Mutua - libre de colisiones) bajo todas las condiciones posibles.
- *Consistente*: La cantidad de recursos en el sistema se mantiene constante. Cada recurso debe ser asignado a un miembro o estar disponible.
- *Simétrico*: Todos los miembros siguen los mismos procedimientos para solicitar recursos.
- *Activo (Liveness [142, 143])*: No debe existir condición lógica por la que el algoritmo detenga su funcionamiento. No deben existir estados donde ocurran bloqueos activos (live-locks) o pasivos (dead-locks).

-
- *Sin Inanición*: Si existen recursos libres en el sistema, un miembro demandante obtendrá los recursos en un tiempo dado.
 - *Concurrente*: Múltiples miembros podrán solicitar recursos concurrentemente.
 - *Alto Rendimiento*: Ante una mayor demanda debe presentar una alta utilización de recursos y un bajo tiempo de respuesta promedio.
 - *Realizar el Mejor Esfuerzo*: No habrá garantías de que un miembro que requiera determinada cantidad de recursos, obtenga lo que requiere. Puede obtener, más de lo requerido, exactamente lo requerido, menos de lo requerido, incluso ningún recurso.
 - *Eficiente*: En períodos de alta carga debe presentar una alta tasa de utilización de recursos respecto al tráfico en la red generado por el propio protocolo.
 - *Previsor*: No todos los recursos libres asignados a un miembro participan del protocolo. Algunos recursos pueden reservarse para su pronta utilización por parte del propio miembro.
 - *Equitativo*: Se asignarán mayor cantidad de recursos a aquellos miembros con mayores demandas.
 - *Estable*: Después de ser sometido a condiciones de cargas variables transitorias, el algoritmo debe converger a un estado estable.
 - *Elástico*: Si hay recursos suficientes, se debe satisfacer la demanda temporaria.
 - *Adaptativo*: Debe soportar la adición y remoción de miembros al cluster.
 - *Alta Disponibilidad*: Debe soportar fallos benignos tales como las desconexiones de los procesos miembros, los nodos donde estos residen y sus re-arranques. Debe permanecer operativo y mantener la consistencia aun ante particiones de la red y posteriores recomposiciones de la misma.

No se incluye la propiedad de escalabilidad, porque está se relaciona con las características de eficiencia y rendimiento del propio algoritmo, y porque en la práctica estará afectada por las características de escalabilidad del GCS utilizado.

En [144] se propone un algoritmo distribuido de asignación de recursos con tolerancia a fallos. Básicamente, consiste en nombrar a la primera SYSTASK en arrancar como Primaria, asignándole la propiedad de todas las ranuras. Cuando arrancan sucesivamente otras SYSTASKs, se le asigna a cada una de ellas un mínimo de ranuras de procesos. Cada SYSTASK propietaria de un conjunto de ranuras (ortogonales entre sí) puede hacer uso de sus ranuras sin notificar al resto de las SYSTASKs. Cuando una SYSTASK no dispone o tiene escasez de ranuras propias, solicita una donación de ranuras al resto del grupo mediante difusión tolerante a fallos. Cada una de las SYSTASK miembro del grupo responde con una donación para la solicitante, desprendiéndose de la propiedad de ranuras y transfiriéndola a la SYSTASK solicitante.

En caso de fallo de una SYSTASK, la propiedad de todas sus ranuras pasa a la SYSTASK Primaria. Si la fallida es la Primaria, se elige una nueva Primaria y se le transfiere la propiedad de sus ranuras a la nueva Primaria. Si se produce una partición de red (Fig. 44), se selecciona un Primario por cada Partición y el algoritmo continúa en cada Partición con el subconjunto de ranuras que son propiedad de todas las SYSTASKs componentes de esa Partición.

Cuando la red se vuelve a unir, se elige un nuevo Primario entre todos los miembros, y se fusionan las tablas de ranuras de todas las particiones que, como son ortogonales, no presentan problemas de consistencia. Esa nueva tabla de ranuras es entonces replicada entre todos los miembros y permite al algoritmo continuar normalmente.

Cada SYSTASK está constituida por dos hilos, uno de ellos es responsable de atender a las peticiones de otras tareas y servidores utilizando M3-IPC, el otro hilo (denominado *SLOTS*) es responsable de las comunicaciones grupales entre SYSTASK de diferentes nodos. Para esta última se utilizó el Spread Toolkit [110]. Cuando los hilos deben acceder a estructuras de datos compartidas tales como las ranuras de procesos, utilizan mutexes para realizar exclusión mutua. Cada uno de los hilos tiene su propio *endpoint*. El hilo *SLOTS* tiene el mismo *endpoint* en todos los nodos (tipo Réplica), dado que solo requiere de hacer operaciones con el *DVK* de su propio nodo.

Cada SYSTASK está informada respecto a los procesos privilegiados que se crean o finalizan en otros nodos, pero no respecto a lo que ocurre con el resto de los procesos de otros nodos.

Si se incorpora una nueva SYSTASK al grupo, previamente la SYSTASK primaria le transfiere el estado del grupo antes de que pueda incorporarse como miembro. Luego le asigna la propiedad de un número mínimo de ranuras de procesos para comenzar a trabajar.

Cluster: {1,2,3,4}		Partición: {2,3}		Partición: {1,4}	
<i>p_nr</i>	Propietario	<i>p_nr</i>	Propietario	<i>p_nr</i>	Propietario
1	Miembro 3	1	Miembro 3		
2	Miembro 3	2	Miembro 3		
3	Miembro 2	3	Miembro 2		
4	Miembro 3	4	Miembro 3		
5	Miembro 2	5	Miembro 2		
6	Miembro 3	6	Miembro 3		
7	Miembro 4			7	Miembro 4
8	Miembro 4			8	Miembro 4
9	Miembro 3	9	Miembro 3		
10	Miembro 1			10	Miembro 1
11	Miembro 1			11	Miembro 1
12	Miembro 1			12	Miembro 1
13	Miembro 3	13	Miembro 3		
14	Miembro 3	14	Miembro 3		
15	Miembro 4			15	Miembro 4
16	Miembro 3	16	Miembro 3		

Figura 44. Partición de Tabla de Descriptores/Ranuras.

Si una SYSTASK finaliza explícitamente o implícitamente (fallo), la propiedad de sus ranuras de procesos pasa a las SYSTASK Primaria y en todos los nodos se de-registran los *endpoints* de las tareas y servidores que estaban registrados en ese nodo como de tipo Local.

4.4.4. Gestor de Discos Virtuales Replicados

Como base para el desarrollo del Gestor de Discos Virtuales de MoL (MoL-VDD) se tomó el código de Minix para gestión de discos RAM, pero se le incorporó la posibilidad de trabajar con varios tipos de dispositivos y con varias réplicas en modalidad Primario-Respaldo [145]. De esta forma era posible comprobar la conmutación transparente de procesos ante un fallo (Respaldo fallido, nuevo Primario) provista por el *DVK*.

Existen diversas formas de realizar la replicación Primario/Respaldo, en Fig. 45 se presentan algunas de ellas. En la primera de ellas (Fig. 45-A), cuando el Primario recibe la petición de escritura (Write) de un cliente, primero replica la petición mediante una difusión FIFO a las réplicas de Respaldo. Luego, actualiza (W) los datos en el dispositivo, y en forma concurrente también lo hacen los Respaldos. El Primario no le retorna respuesta al Cliente hasta tanto no obtenga las respuestas de todos los Respaldos. Esta opción provoca un retraso en la respuesta al cliente dado que el tiempo de respuesta se ve influenciado por los tiempos de comunicación entre el Primario y los Respaldos, pero asegura que todos los Respaldos han actualizado sus réplicas.

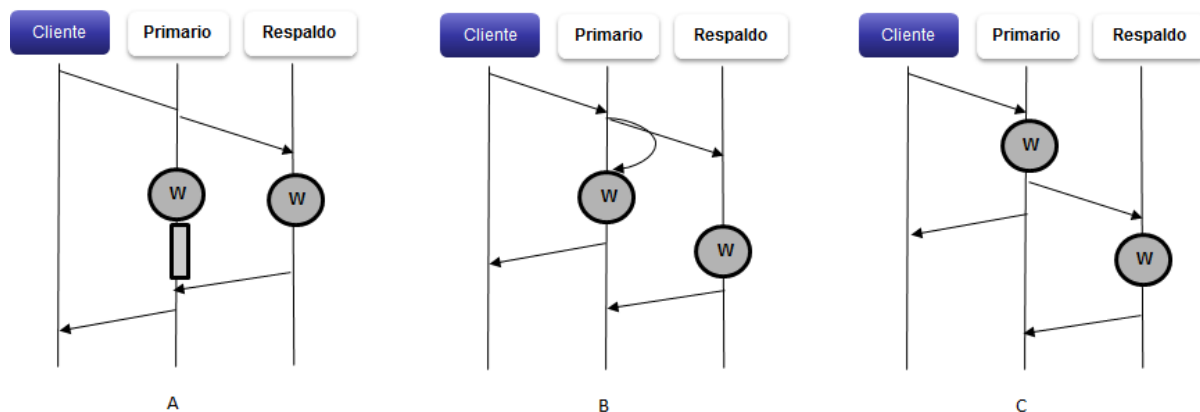


Figura 45. Distintas técnicas de Replicación Primario-Respaldo.

En la segunda opción (Fig.45-B), el Primario replica la petición mediante una difusión FIFO a las réplicas de Respaldo. Recién cuando el Primario recibe su propio mensaje realiza la actualización (W) del dispositivo. Luego, el Primario le responde al Cliente sin esperar por las respuestas de los Respaldos. El Primario no acepta nuevas peticiones hasta tanto no lleguen las respuestas de todos los Respaldos. Esta opción mejora el tiempo de respuesta, pero mantiene idéntica tasa de transacciones de actualizaciones por unidad de tiempo que la primera opción.

Finalmente, la tercera opción (Fig.45-C) es tal que una vez que el Primario recibe la petición del Cliente, lleva a cabo la actualización de su dispositivo (W) y le responde al Cliente (de igual forma que si ejecutase sin Respaldos). Luego, replica la petición mediante una difusión FIFO a las réplicas de Respaldo y no acepta nuevas peticiones hasta finalizar de recoger los resultados de todas los Respaldos. Esta opción tiene aún mejor tiempo de respuesta que la segunda e igual tasa de escrituras por unidad de tiempo, pero puede suceder que si inmediatamente después de que el Primario realiza su actualización éste falla antes de realizar la difusión, no se llevan a cabo las actualizaciones en los Respaldo.

Para MoL-VDD se eligió la primera opción de Replicación por su simplicidad. Cuando no utiliza replicación, solo se crea un hilo que atenderá las peticiones de los clientes (generalmente el Servidor de Sistemas de Archivos – MoL-FS). Los servicios que básicamente brinda el gestor son: abrir o conectar el dispositivo, leer uno o varios bloques de datos, escribir uno o varios bloques de datos, y cerrar o desconectar el dispositivo. Todas estas peticiones se hacen utilizando M3-IPC, y en particular *mnx_vcopy()* para realizar la copia de los bloques de datos entre el cliente y MoL-VDD.

Cuando MoL-VDD se utiliza replicado (una réplica por nodo del DC), se crea el hilo Principal con las funciones ya mencionadas, y el hilo Réplica que utiliza como GCS a Spread Toolkit (Fig. 46).

En el Primario, ambos hilos están activos, en cambio en los Respaldo el hilo Principal se encuentra bloqueado hasta tanto esa Réplica se transforme en Primario por un fallo del original.

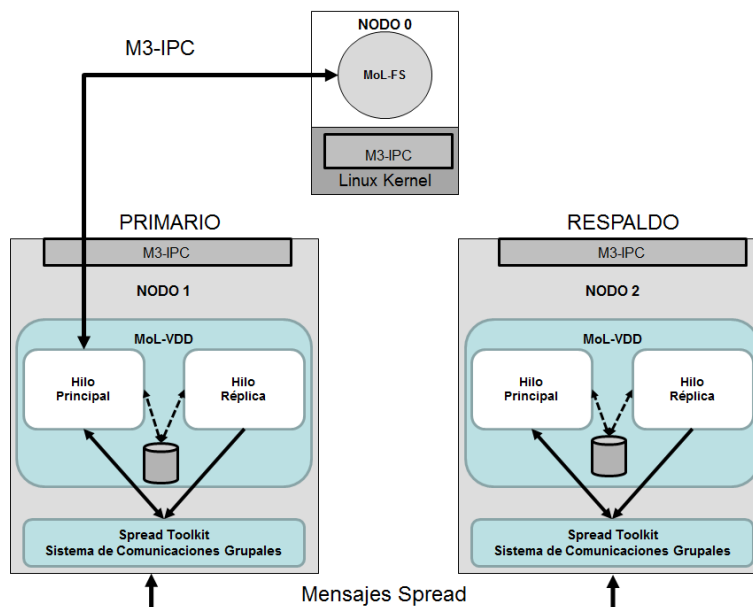


Figura 46. MoL-VDD Replicado.

Cuando se realizan actualizaciones sobre la imagen de disco (operaciones WRITE), el MoL-VDD Primario realizará una difusión FIFO a los procesos MoL-VDD (Respaldo) de los otros nodos del DC utilizando Spread Toolkit. Todas las réplicas se actualizarán y finalmente el Primario retornará el código del resultado de la operación al proceso solicitante (en este ejemplo MoL-FS).

En caso de producirse un fallo de caída (crash) de MoL-VDD Primario, Spread Toolkit lo detecta e informa a los otros nodos de tal evento mediante un mensaje (Fig. 47). Al recibir este mensaje, todas las réplicas detienen las comunicaciones con el *endpoint* del Primario utilizando *mxn_migr_start()*.

Entre los MoL-VDD Respaldo activos y sincronizados, se ejecuta un algoritmo de selección de líder basándose en la lista de procesos. El líder se transforma en el nuevo MoL-VDD Primario y su *endpoint* se transforma de tipo Respaldo a tipo Local mediante la ejecución de *mxn_migr_commit()* en todos los nodos del DC. A partir de este momento, se reinician las comunicaciones M3-IPC entre el cliente y el hilo Principal del nuevo Primario.

El algoritmo contempla los siguientes eventos que se pueden dar en el grupo: 1) La incorporación de nuevos miembros al grupo (*Join*), la finalización explícita o por fallo de miembros (*Disconnect*), la detección de partición de red (*Partition*) y la re-uniión de esas particiones (*Merge*).

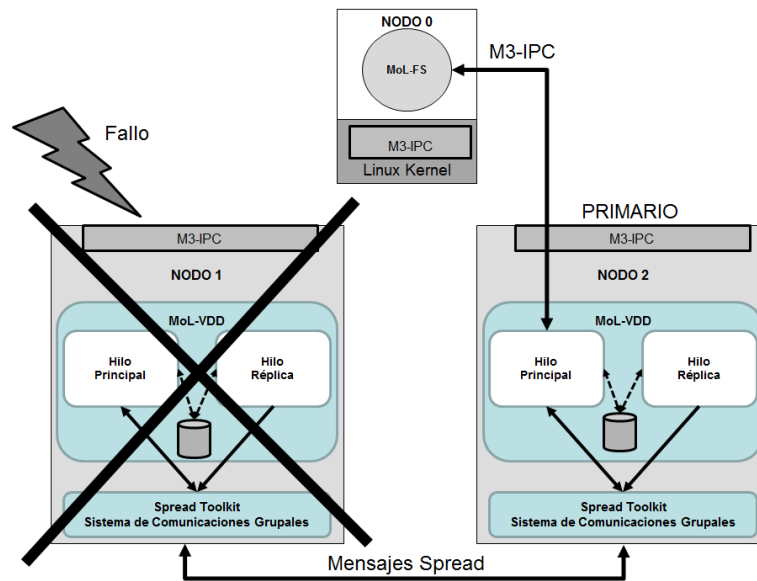


Figura 47. MoL-VDD Replicado después el fallo de Primario Original (en NODO1).

Los hilos Principal y Réplica comparten variables de estado, por lo que para evitar condiciones de competencia se utilizan mutexes de Linux para el acceso serializado a las mismas.

El hilo Principal es responsable dar atención a las peticiones de E/S provenientes de MoL-FS utilizando el protocolo M3-IPC y sólo en las operaciones de escritura, difunde mensajes de actualización a todo el grupo de réplicas. En el hilo Réplica se implementa el protocolo Primario-Respaldo de los servicios de MoL-VDD. Se impone como condición que la primera réplica a iniciar (a la que se le asignará la función de Primario) disponga de la versión más actual de la imagen de disco.

En la primera versión de MoL-VDD se asumía que las imágenes de disco del Primario y de los Respaldo son idénticas en el momento de comenzar su ejecución. En esa versión tampoco se admite la incorporación de nuevos procesos MoL-VDD al grupo una vez que comienza a operar dado que podría implicar actualizaciones de la imagen de disco. La versión actual soporta el proceso de Sincronización Dinámica de Réplicas que se detallará en la [Sección 4.4.5](#).

Para incrementar la eficiencia de la red MoL-VDD dispone de la opción de comprimir las comunicaciones entre las réplicas (GCS), para ello utilizó el algoritmo LZ4 [146]. De igual forma se podrían cifrar las comunicaciones para asegurar su privacidad e integridad. Las comunicaciones entre MoL-VDD y sus clientes no se comprime (aunque se podría hacer) porque se pueden utilizar *proxies* del DVS con compresión para todas las comunicaciones entre nodos.

4.4.5. Sincronización Dinámica de Réplicas

Cuando se replica almacenamiento, como es el caso de los discos rígidos o de los discos de estado sólido (SSD), todas las réplicas deben ser idénticas al iniciar los servicios y las mismas deben ser la última versión del contenido del dispositivo.

Cómo la replicación se utiliza como técnica para tolerar fallos, es frecuente que un servidor tipo réplica o un dispositivo de una de las réplicas fallen. Si falló el servidor, durante el tiempo en que éste no estuvo disponible, pudieron efectuarse actualizaciones en el resto de las réplicas, por lo que la versión de los datos contenidos en el dispositivo se encuentra fuera de “*sincronismo*”.

De igual forma ocurre cuando una unidad de disco falló y se reemplazó por una nueva unidad sin datos o quizás con los datos del último respaldo. Cualquiera sea el caso, los datos del dispositivo están fuera de “*sincronismo*” con respecto al resto de las réplicas.

Una alternativa para volver a sincronizar los dispositivos es detener a todas las réplicas (o al menos las actualizaciones) y transferirle todo el contenido a la nueva réplica o a la réplica repuesta, para luego volver a integrarla al grupo de réplicas. Claramente, esta solución no es adecuada por el tiempo de indisponibilidad del servicio.

En el prototipo se desarrolló el Gestor de Disco Virtual (MoL-VDD) con replicación y sincronización dinámica que permite continuar brindando el servicio mientras se sincronizan las réplicas.

El MoL-VDD utiliza M3-IPC para comunicarse con sus clientes (generalmente MoL-FS), y utiliza Spread Toolkit como GCS para implementar el protocolo de replicación Primario-Respaldo. Cuando se incorpora un nuevo proceso Respaldo al grupo, éste debe sincronizar el contenido del disco virtual con el resto de las réplicas activas. Para ello, se establece una sesión TCP/IP entre la nueva réplica y el Primario. El Primario crea un hilo específico por cada una de estas operaciones de sincronización y mantiene la sesión durante la sincronización. El proceso de sincronización consiste en las siguientes operaciones:

- La nueva réplica aplica un algoritmo de resumen (hash MD5) a un conjunto de bloques de discos en secuencia ascendente de números de bloques. Le transfiere al Primario los números de bloques considerados y el resumen obtenido. Luego queda a la espera de la respuesta del Primario.

-
- El Primario aplica el mismo algoritmo sobre el mismo conjunto de bloques de disco, y si obtiene el mismo resumen implica que los bloques son idénticos respondiéndole a la nueva Réplica para que continúe con el siguiente conjunto de bloques.
 - Si el resumen obtenido por el Primario es diferente, éste le transfiere todos los bloques del conjunto a la nueva réplica y luego le responde a la nueva Réplica que continúe con el siguiente conjunto de bloques.

De esta forma se lograría que la nueva réplica obtuviese del Primario los datos modificados mientras ésta se encontraba fuera del grupo (si es que era una réplica fallida). Pero, durante el proceso de sincronización mencionado, el Primario siguió recibiendo peticiones de sus clientes (en otro hilo), algunas de las cuales pudieron haber sido actualizaciones de bloques que ya fueron transferidos desde el Primario a la nueva réplica. Como consecuencia de esto, la nueva réplica perdería sincronización de los datos nuevamente. Para evitar esta situación, la nueva réplica crea otro hilo que se une al grupo de réplicas como “oyente” (no se la considera como Respaldo) recibiendo así todos los mensajes de difusión del Primario cuando hay actualizaciones. De esta forma, cuando la nueva réplica recibe un mensaje de una actualización por parte del Primario, analiza si ese bloque de datos ya le fue transferido por el Primario previamente. De ser así, actualiza el bloque nuevamente conforme al mensaje difundido. Si no le fue transferido aún, lo ignora dado que el Primario transferirá ese bloque cuando corresponda en la secuencia. Cuando la nueva réplica finalizó su sincronización, se convierte de *oyente* a Respaldo activo. De esta manera, se puede realizar la sincronización de los datos del dispositivo en forma dinámica sin detener los servicios.

MoL-VDD soporta sincronización Total (FULL) en donde todos los bloques del dispositivo a sincronizar son actualizados, o Parcial (PARTIAL) en donde se utiliza la verificación de igualdad de bloques mediante MD5.

Para mejorar el rendimiento, las transferencias de bloques entre el Primario y la nueva réplica se pueden realizar con compresión utilizando el algoritmo LZ4 (opcional de configuración). De igual forma, se puede implementar cifrado de datos para evitar la pérdida de confidencialidad, aunque esto último no está disponible en la versión actual del prototipo.

4.5. Misceláneos

Para probar el desempeño y versatilidad de M3-IPC se desarrollaron un conjunto de aplicaciones simples.

4.5.1. Transferencia de Archivos entre MoL (Guest) y Linux (Host)

m3copy es una herramienta que se ejecuta sobre MoL. Permite copiar un archivo Linux a un archivo MoL-FS y viceversa.

El formato de uso es:

```
m3copy {-p|-g} <mol_fname> <linux_fname>
```

Donde *p* (PUT) se utiliza para realizar una copia desde un archivo Linux a un archivo MoL, y *g* (GET) se utiliza para realizar una copia desde un archivo MoL a un archivo Linux.

4.5.2. Servidor FTP y Cliente FTP utilizando M3-IPC

Generalmente el FTP (File Transfer Protocol) está asociado al protocolo TCP. En este caso a modo de prueba de versatilidad del protocolo M3-IPC se desarrollaron un servidor y un cliente FTP basados en M3-IPC. Ambos deben ejecutarse dentro del mismo DC.

El servidor *ftpd*, no requiere argumentos, y el cliente *ftp*, tiene el siguiente formato de uso:

```
m3ftp {-p|-g} <srv_ep> <rmt_fname> <lcl_fname>
```

Donde *srv_ep* es el *endpoint* del servidor *ftpd*. El selector *p* (PUT) se utiliza para realizar una transferencia desde un archivo Linux con nombre *lcl_fname* hacia el servidor *ftpd* con nombre *rmt_fname*. El selector *g* (GET) se utiliza para realizar una transferencia desde un archivo Linux desde el servidor *ftpd* con nombre *rmt_fname* hacia un archivo con nombre *lcl_fname*.

4.5.3. Servidor Web y Cliente Web utilizando M3-IPC

Generalmente se asocia el protocolo HTTP al protocolo TCP. Se desarrolló (modificando código de las herramientas *nweb* y *urlget*) un servidor Web y un cliente Web básicos que utilizan HTTP sobre M3-IPC.

El servidor *websrv* no tiene argumentos, pero si tiene un archivo de configuración asociado en el mismo directorio. Como pueden ejecutar varios servidores concurrentemente, en este archivo de configuración se establecen las relaciones entre el *endpoint* del servidor y el par {*puerto, dirección IP*}. El formato del archivo de configuración es el siguiente:


```

websrv SERVER1 {
  port 80;
  endpoint 20;
  rootdir "/lib/init/rw/nweb";
  ipaddr "192.168.1.100";
};

```

Este puerto y dirección IP son falsos, pero permiten entregarle al cliente esa información en forma equivalente a lo que entregaría un servidor HTTP sobre TCP/IP.

4.5.4. Proxy HTTP – M3-IPC

Cuando se ejecuta un servidor web sobre MoL o sobre ukVOS, éste podría tener asignado una dirección IP dada por su VOS. Otra opción es que el VOS no utilice una pila de protocolo TCP, y que el servidor web ofrezca sus servicios en protocolo M3-IPC. Esto resultaría de poca utilidad dado que los clientes web en general no dispondrían del soporte de este protocolo y los obligaría a formar parte de un DC para poder acceder al servidor web.

Otra opción, es utilizar un proxy HTTP-M3IPC que convierta las peticiones de los clientes HTTP ordinarios sobre TCP a peticiones HTTP sobre M3-IPC. En la Fig. 48 se muestra un ejemplo de uso en donde un server Web sobre M3-IPC ejecuta sobre un MoL.

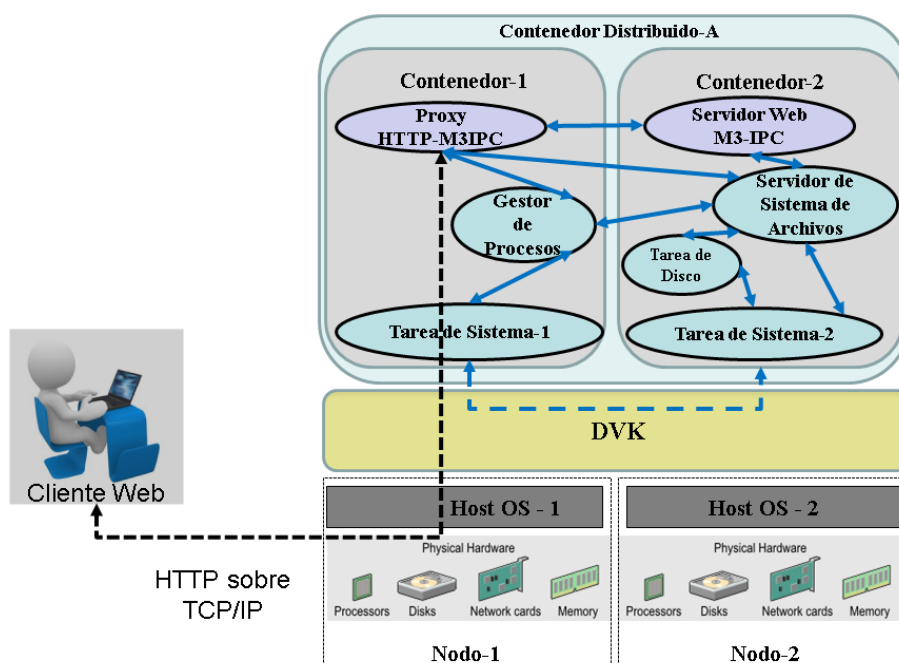


Figura 48. Ejemplo de uso del Proxy HTTP-M3IPC

En este ejemplo, MoL no tiene soporte de red, pero tiene un proxy HTTP-M3IPC con acceso a la red del Linux Host (debe tener los privilegios adecuados). De esta forma, cuando llega desde el

cliente web una petición HTTP sobre TCP, el proxy HTTP-M3IPC lo convierte en HTTP sobre M3-IPC y se la envía al servidor web M3-IPC. La conversión inversa se hace sobre la respuesta del servidor.

5. EVALUACION DEL PROTOTIPO

Un DVS tiene un sin número de aspectos a evaluar: rendimiento, utilización de recursos, tolerancia a fallos, disponibilidad de servicios, escalabilidad, capacidad de gestión, etc. En esta sección se presentarán las evaluaciones parciales de los componentes más significativos, focalizando las mismas en el rendimiento.

5.1. M3-IPC

Esta sección está dedicada a la evaluación del rendimiento de M3-IPC frente a otros mecanismos de IPC. El cumplimiento de los otros objetivos de diseño expuestos en la [Sección 4.1.7](#) fue verificado durante y después de la etapa de desarrollo utilizando varios escenarios de prueba.

Como se mencionó previamente, el autor desarrolló MoL [55] utilizando los mecanismos de comunicaciones disponibles en Linux (Unix/TCP/UDP Sockets). Si bien el prototipo de MoL funcionaba correctamente, el rendimiento de IPC era su punto más débil. Es por esta razón es que se desarrolló M3-IPC y por cual el rendimiento es considerado una métrica crítica.

Las pruebas de rendimiento se distinguen entre transferencias de mensajes y copia de bloques de datos. A su vez, se diferencian aquellas realizadas entre procesos ejecutando en el mismo nodo y otras entre procesos en diferentes nodos. Es importante comentar que durante las pruebas de rendimiento se midió el consumo de CPU, resultando este entre el 98% y 100% de utilización (por cada núcleo utilizado) en las transferencias Locales y menor al 8% en las transferencias remotas, dependiendo del protocolo de transporte utilizado.

Para los micro-benchmarks se utilizó el siguiente patrón de comunicación: un servidor espera para recibir un mensaje de un cliente e inmediatamente responde. Una vez que el cliente envía la

solicitud al servidor, espera la respuesta. La respuesta puede ser un mensaje (36 bytes) o un bloque de datos (36 bytes a 64 Kbytes).

Las pruebas se realizaron en un clúster de 8 (ocho) CPU Intel(R) i5 que consta de 2 núcleos con 2 hilos por núcleo a [650@3.20GHz](#) con un ancho de banda de memoria de 20,841 [Gbytes/s] para bloques de 64 Kbytes (informados por el utilitario *bm_mem*) unidos por un switch de LAN dedicado de 1 Gbps. De acuerdo a como Linux considera esta CPU (*/proc/cpuinfo*), en los gráficos se referencia como de 4 núcleos procesadores.

Algunas otras pruebas de transferencias locales se hicieron utilizando una CPU Intel(R) Core(TM) i7 CPU que consta de 4 núcleos con 2 hilos por núcleo a [860@2.80GHz](#) para poder comprobar el desempeño con una mayor cantidad de núcleos. De acuerdo a como Linux considera esta CPU (*/proc/cpuinfo*), en los gráficos se referencia como de 8 núcleos procesadores.

La habilitación/deshabilitación de núcleos procesadores en Linux se realiza mediante una simple escritura en un archivo de pseudo-filesystem */proc*. Por ejemplo, para deshabilitar el núcleo procesador 1, se debe ejecutar el siguiente comando:

```
echo 0 > /sys/devices/system/cpu/cpu1/online
```

En tanto que para volverlo a habitar:

```
echo 1 > /sys/devices/system/cpu/cpu1/online
```

Las pruebas entre procesos coubicados permiten comparar el rendimiento de M3-IPC con otros mecanismos de IPC disponibles en Linux. Uno de los objetivos de diseño establece que el rendimiento esperado debería ser tan bueno como los mecanismos de IPC más rápidos disponibles en ese OS. Los mecanismos de IPC Message Queues, RPC, TIPC, FIFOs, pipes, Unix Sockets, Sockets TCP, SRR, SIMPL se probaron utilizando micro-benchmarks disponibles en [147, 148] y otros fueron adaptados para lograr equivalencia en las operaciones.

Para las pruebas entre procesos ubicados en diferentes nodos se utilizaron tres protocolos diferentes para los *proxies* de M3-IPC (TCP, TIPC y RAW Ethernet) y dos protocolos de contraste (TIPC, RPC). Se debe considerar que M3-IPC no realiza control de flujo, control de errores o control de congestión. La resolución de estos problemas se delega a los *proxies*.

Las mediciones para las pruebas de transferencias locales se realizaron ejecutando 10 pruebas de 3.000.000 de ciclos cada una. En tanto que para las transferencias remotas se ejecutaron 10 pruebas de 5000 ciclos cada una.

En septiembre del 2015 comenzó el desarrollo de *gRPC* [149], un mecanismo genérico de RPC, el cuál no fue evaluado en la oportunidad de realizar los micro-benchmarks de M3-IPC, dado que éstos fueron realizados a principios de ese mismo año. Este protocolo es utilizado por servicios y empresas de renombre (Google, Netflix, Juniper, Cisco, etc), por lo que merece un análisis detallado de sus características y la posibilidad de integrarlo al *DVS*.

❑ Rendimiento de transferencia de mensajes (Local)

Los resultados presentados en Fig. 49 resumen el rendimiento alcanzado por los mecanismos de IPC que ejecutan un solo par de procesos Cliente/Servidor. Los mecanismos de IPC de Linux con el mayor rendimiento con 4 núcleos habilitados fueron las tuberías sin nombre (PIPE) y los sockets UNIX, seguidos por M3-IPC. En tanto que cuando se realizaron las pruebas habilitando un solo núcleo, las tuberías con nombre (o FIFO) obtuvieron el mayor rendimiento por sobre M3-IPC (925.314 [msj/s]).

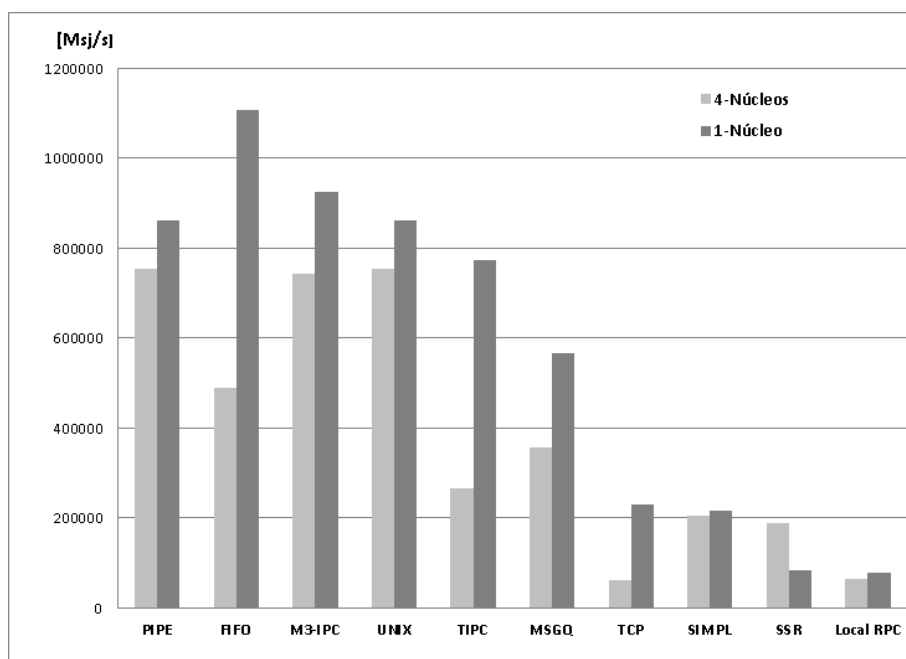


Figura 49. Rendimiento comparativo de transferencias Locales de mensajes.

M3-IPC ha sido desarrollado siguiendo las recomendaciones de programación para trabajar eficientemente en sistemas multi-núcleo y SMP. Sin embargo, dado que utiliza las facilidades disponibles en el kernel de un Linux ordinario (mutexes, spinlocks, contadores de referencia, RCU, etc.), su escalabilidad está limitada por el rendimiento de esas facilidades, como se reporta en [150]. Los micro-benchmarks revelan un rendimiento decreciente con un mayor número de núcleos activos. Varios de los mecanismos de IPC que brinda el kernel de Linux utilizan RCU como mecanismo de exclusión mutua. RCU tiene un muy buen rendimiento cuando se utilizan accesos de Lectura, pero

mayor sobrecarga en las escrituras [150]. Esto se ve reflejado en las diferencias que se presentan en el rendimiento al utilizar 4 núcleos activos o 1 núcleo en los micro-benchmarks en la Fig. 49.

En los micro-benchmarks para la transferencia de mensajes de M3-IPC se realizan las siguientes operaciones: El servidor se bloquea a la espera de recibir un mensaje desde cualquier otro *endpoint*. El Cliente realiza un ciclo en el cual envía un mensaje al Servidor y luego se bloquea a la espera de la respuesta. Cuando el Servidor recibe el mensaje del Cliente, inmediatamente le envía un mensaje de respuesta. El pseudo-código de los micro-benchmarks para la transferencia de mensajes M3-IPC es el siguiente:

SERVIDOR	CLIENTE
<pre>for(i = 0; i < loops; i++) { ret = mnx_receive(ANY, (long) m_ptr); ret = mnx_send(src_ep, (long) m_ptr); }</pre>	<pre>for(i = 0; i < loops; i++){ ret = mnx_sendrec(dst_ep, (long) m_ptr); }</pre>

En los micro-benchmarks para la transferencia de bloques de datos se realiza lo siguiente: El servidor se bloquea a la espera de recibir un mensaje desde cualquier otro *endpoint*. El Cliente envía un mensaje al Servidor en donde le indica la dirección del buffer donde requiere recibir los datos transferidos, luego se bloquea a la espera de la respuesta. El Servidor recibe el mensaje, del cual obtiene la dirección del buffer de cliente. Luego, inicia un ciclo de transferencias de un mismo bloque de datos desde el espacio de direcciones del Servidor al espacio de direcciones del Cliente. Finalmente, el Servidor le envía el mensaje de respuesta al Cliente, dando por terminadas las transferencias. El pseudo-código es el siguiente.

SERVIDOR	CLIENTE
<pre>message m, *m_ptr; m_ptr = &m; ret = mnx_receive(ANY, (long) m_ptr); clt_buf = m_ptr->m1_p1; for(i = 0; i < loops; i++) { ret = mnx_vcopy(svr_ep, svr_buf, clt_ep, clt_buf, maxbuf); } ret = mnx_send(src_ep, (long) m_ptr);</pre>	<pre>message m, *m_ptr; m_ptr = &m; m_ptr->m1_p1 = clt_buf; ret = mnx_sendrec(dst_ep, (long) m_ptr);</pre>

Para comprobar el rendimiento de M3-IPC en una CPU con varios núcleos habilitados se ejecutó otro micro-benchmark de transferencias de mensajes entre múltiples pares de procesos Cliente/Servidor en forma concurrente (Fig. 50).

El rendimiento más alto fue de 2.109.006 [msj/s], que se alcanzó con 6 pares de procesos Cliente/Servidor sobre una CPU de 8 núcleos. Se evaluó también el rendimiento de transferencia de

mensajes de un OS Minix [41] arrojando 1.875.000 [msj/s] para la versión 3.12, y 1.449.309 [msj/s] para la versión 3.20. Esas versiones de Minix no soportan SMP, por lo cual utiliza solo un núcleo.

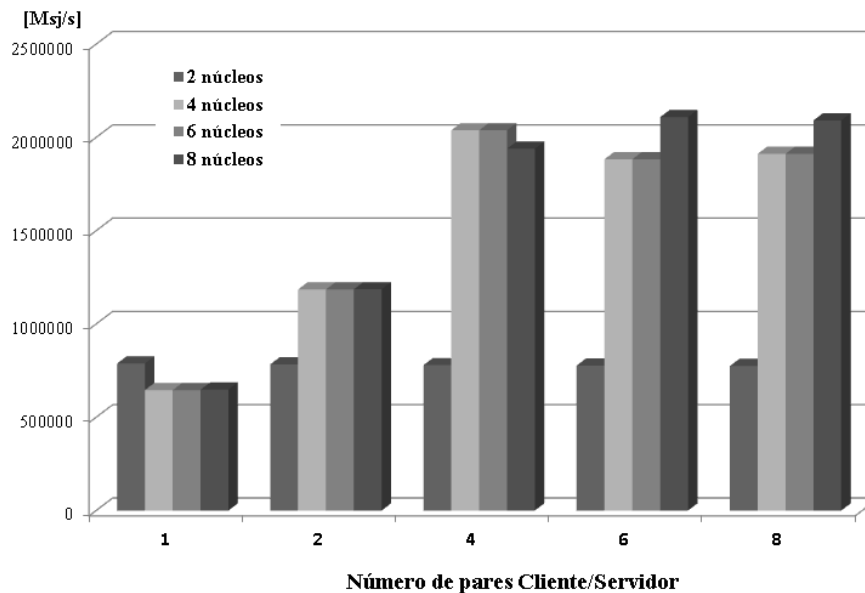


Figura 50. Rendimiento en múltiples transferencias Locales simultáneas de mensajes.

□ Rendimiento de copia de datos (Local)

En la Fig. 51 se presenta el rendimiento comparativo entre varios mecanismos de IPC en Linux.

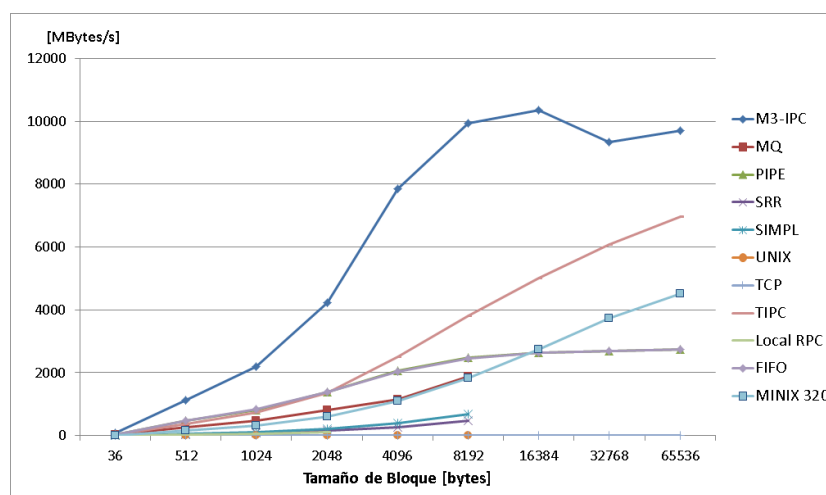


Figura 51. Rendimiento comparativo de copias Locales de bloques de datos.

Las razones de que M3-IPC supere ampliamente a todos los otros mecanismos de IPC son: 1) M3-IPC realiza una única copia de datos entre espacios de direcciones, mientras que los demás realizan al menos dos copias (origen a kernel, kernel a destino), lo que implica un cambio de contexto;

2) M3-IPC no requiere cambio de contexto ya que la copia se hace en el contexto del proceso solicitante; 3) M3-IPC utiliza la función `page_copy()` proporcionada por el kernel de Linux que utiliza instrucciones MMX de alto rendimiento.

Para comprobar el rendimiento de M3-IPC en una CPU con varios núcleos habilitados se ejecutó otro micro-benchmark de copia de bloques de datos entre múltiples pares de procesos Cliente/Servidor en forma concurrente (Fig. 52). Aquí resulta nuevamente una diferencia (entre 10% y 18%) de pérdida de rendimiento al aumentar el número de núcleos activos.

Se evaluó también el rendimiento de copia de bloques de datos de un OS Minix [41] arrojando 3,211 [Gbytes/s] para la versión 3.12, y 4,883 [Gbytes/s] para la versión 3.20. Esas versiones de Minix no soportan SMP, por lo cual utiliza solo un núcleo.

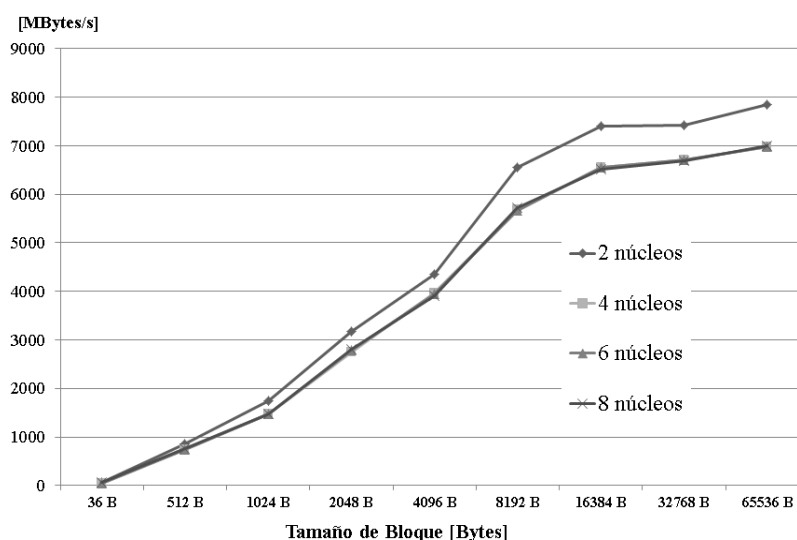


Figura 52. Rendimiento en múltiples copias Locales simultáneas de datos.

❑ Rendimiento de transferencia de mensajes (Remoto)

Cuando M3-IPC utiliza *proxies* en modo usuario y en modo kernel con protocolo TCP tiene un rendimiento similar al de RPC (Fig. 53). El rendimiento es bajo (al igual que con RPC) como consecuencia de que el TCP es un protocolo “conversador”, es decir, por cada mensaje M3-IPC que se envía de un nodo a otro, se requiere un mensaje del proxy del nodo emisor y un mensaje de reconocimiento del proxy del nodo receptor.

pero por cada uno de esos mensajes, TCP requiere un reconocimiento, resultando entonces que por cada mensaje transferido de la aplicación se requieren 2 mensajes de proxies que a su vez totalizan 4 mensajes de TCP. Es importante destacar que en los micro-benchmarks se deshabilitó el algoritmo de Nagle, de tal forma de reducir la latencia en el envío de los segmentos TCP lo que podría reducir el número de mensajes, pero aumenta la latencia de los mensajes de reconocimiento retrasando así, indirectamente a la aplicación.

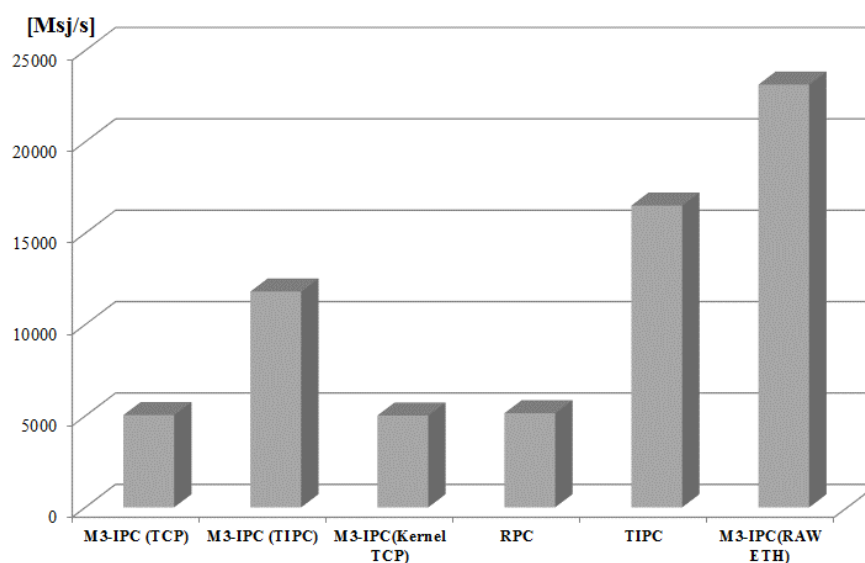


Figura 53. Rendimiento en transferencias Remotas de mensajes.

El rendimiento notable de TIPC sugirió que podría ser una buena opción para ser utilizado por los *proxies* M3-IPC como protocolo de transporte. La versatilidad y flexibilidad de M3-IPC en la programación de *proxies* permitió modificar el código fuente de *proxies* en pocos minutos para usar TIPC en lugar de TCP lográndose un incremento en el rendimiento del 133%. Posteriormente se desarrolló un protocolo sencillo basado en RAW Ethernet (sin capacidad de ruteo) y se implementó un proxy M3-IPC, obteniendo un incremento respecto a TCP de 358%.

La Fig. 53 evidencia el impacto que tiene el protocolo de transporte en el rendimiento de M3-IPC en las transferencias de mensajes.

❑ Rendimiento de copia de datos (Remoto)

Como se muestra en la Fig. 54, TIPC presenta el mayor rendimiento (54 [MBytes/s]) con bloques de datos hasta 8 KBytes, lo que confirma los resultados presentados en [120]. Luego es superado por M3-IPC con proxy RAW Ethernet obteniéndose una tasa de transferencia de 87 [MBytes/s] para bloques de 32 KBytes.

Los resultados de las transferencias de mensajes y de copias de bloques de datos entre procesos en diferentes nodos, hace dudar al administrador respecto del protocolo a utilizar en sus *proxies*. Los *proxies* con protocolo RAW Ethernet tienen mejor rendimiento para las transferencias de mensajes, mientras que para las copias de bloques de datos de tamaño pequeño TIPC tiene un mejor desempeño. Un aspecto que puede definir la decisión es que TIPC utiliza IP, por lo que es un protocolo de capa 3 con capacidad de ruteo, mientras que el protocolo en RAW Ethernet solo admite comunicaciones dentro de la misma LAN.

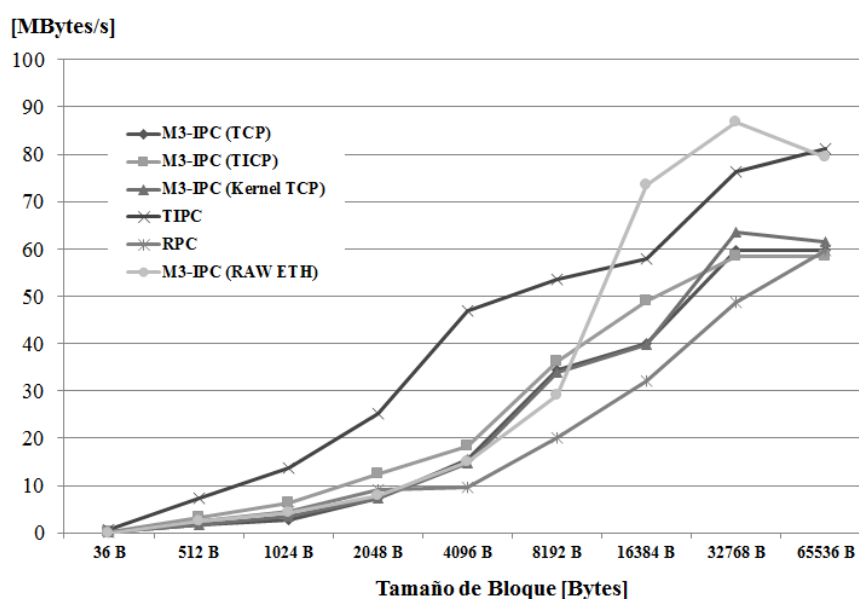


Figura 54. Rendimiento en transferencias Remotas de datos.

Entre los protocolos utilizados para la construcción de los *proxies* también utilizó UDT conforme al rendimiento presentado en [151]. Los micro-benchmarks realizados con los *proxies* sobre este protocolo no se correspondieron con ese rendimiento, probablemente porque las transferencias realizadas eran de hasta 64 Kbytes. Como el rendimiento mostrado por el proxy UDT resultó tan bajo, éste no fue incluido en las gráficas, esperando realizar un análisis más exhaustivo de su comportamiento con los resultados presentados en publicaciones al respecto en el futuro.

Dado el buen desempeño en general de TIPC y sus *proxies* en modo usuario para M3-IPC, se propone desarrollar los *proxies* con TIPC en modo kernel ya que este protocolo se encuentra embebido en el kernel de Linux.

5.2. Algoritmo de Asignación de Ranuras

Para la evaluación de rendimiento de SLOTS se utilizó como metodología la simulación del algoritmo, siguiendo un enfoque similar a [152] donde se modelan y evalúan algoritmos para asignación de recursos de una red Wireless. La simulación permitió utilizar modelos estadísticos y así realizar pruebas generando diversos tipos de cargas de trabajo.

Se realizó un modelo^{Nota3} de SLOTS para el simulador DAJ [153], al igual que el modelo del algoritmo de exclusión mutua utilizado para contraste de rendimiento. El código de simulación de ambos algoritmos se encuentra disponible en <https://github.com/oajara/slots-simulation>.

En esta sección los recursos estarán representados por ranuras que almacenan descriptores de procesos. Para la creación de un nuevo proceso, otro proceso realiza una llamada al sistema *fork()*. Para que el *fork()* sea exitoso se debe disponer de un ranura libre para el nuevo descriptor de proceso.

Para la simulación se implementó la versión simple del algoritmo incluidas las optimizaciones (no es tolerante a fallos ni a cambios de membresía). El modelo simulado permite efectuar modificaciones de parámetros de operación tales como:

- NR_NODES*: Cantidad de nodos/miembros.
- NR_SLOTS*: Cantidad de ranuras a compartir.
- FREE_LOW*: Límite mínimo de ranuras libres por nodo.
- λ : Tasa de arribo de solicitudes de creación de procesos.
- μ : Tiempo de vida (lifetime) de los procesos.

En [154] se reporta que el tiempo entre arribos de sesiones en un web server sigue una distribución de Poisson. Como se puede asumir una equivalencia entre el inicio de una nueva sesión y la creación de un proceso para atenderla, se adoptó esta misma distribución λ para la simulación. En tanto que para μ se adoptó una distribución normal inversa [155].

El GCS se modeló como un nodo servidor de difusión (llamado SPREAD) encargado de reenviar los mensajes con Orden Total hacia los nodos componentes del cluster. Cuando un nodo miembro requiere realizar una difusión de un mensaje, éste envía el mensaje utilizando *send()* hacia el nodo SPREAD. Luego, el nodo SPREAD reenvía ese mensaje (también utilizando *send()*) hacia cada uno de los nodos que componen el cluster, incluido el emisor original. Dado que la entrega de

Nota3: El desarrollo de programa en el simulador DAJ estuvo a cargo del ingeniero Oscar Jara de Universidad Tecnológica Nacional, Facultad Regional Santa Fe bajo la dirección de Pablo Pessolani.

mensajes sigue el orden FIFO tanto en el nodo SPREAD como en los nodos miembros, se asegura entrega con Orden Total.

Utilizando la misma topología se implementó un algoritmo basado en exclusión mutua para realizar un contraste del rendimiento. En este algoritmo, cuando un miembro no dispone de ranuras libres para utilizar realiza una difusión (a través del nodo SPREAD) de un mensaje tipo *MSG_FORK* indicando la cantidad requerida de ranuras (*FREE_LOW*). Cuando un miembro (incluido el emisor) recibe este mensaje, busca las primeras *FREE_LOW* ranuras libres de la tabla de estado de los procesos (PST: Process Status Table) y se los asigna al emisor del mensaje. Como la difusión respeta el Orden Total de entrega, todos los miembros reciben el mensaje en el mismo orden, y todos asignan las mismas ranuras al emisor. Cuando un miembro dispone de una cantidad de ranuras mayor o igual a un valor establecido *FREE_HIGH*, libera ranuras hasta quedarse solo con *LOW_SLOTS*. Para ello difunde un mensaje tipo *MSG_EXIT*, especificando los identificadores de las ranuras liberadas.

Los indicadores de rendimiento evaluados son:

- A. *Utilización*: Es la suma de ranuras utilizadas por todos los miembros respecto al total (*NR_SLOTS*).
- B. *Tasa de Peticiones Fallidas/Exitosas*: Es la tasa entre las peticiones para utilizar una ranura (*fork*) cuando no hay ranuras disponibles en el miembro respecto a las peticiones que resultaron exitosas.
- C. *Eficiencia de Mensajes*: Para el caso de SLOTS es la tasa entre las peticiones exitosas respecto al total de mensajes de todos los ciclos de donación. Cada ciclo de donación implica *NR_NODES* difusiones de mensajes por lo que es un indicador de eficiencia en lo que a utilización de la red refiere.
- D. *Tiempo de Respuesta*: Como el tiempo de simulación no es comparable con el tiempo real, la unidad utilizada es el Tiempo de Difusión (Δ_{mcast}). En una implementación realista, el Δ_{mcast} dependerá del GCS utilizado y de la infraestructura del cluster. El Tiempo de Respuesta considera el número de difusión de mensajes que se requieren desde que un miembro emite una solicitud de donación de ranuras hasta que recibe un mensaje de donación de al menos una ranura.

Por las características de DAJ, los modelos utilizados en las simulaciones realizan rondas (rounds) en las cuales cada uno de los nodos lleva a cabo el procesamiento, es decir se hace un

procesamiento secuencial rotando el nodo activo. En una ronda todos los nodos han ejecutado “*concurrentemente*”. Por esta característica es que; tanto para la tasa de arribos de requerimiento de recursos, como para el tiempo de uso del recurso, se utilizó como unidad de medición del tiempo a la ronda de simulador.

En la Fig. 55 se presenta la utilización de ranuras para la carga de trabajo impuesta al cluster de una de las simulaciones con los siguientes parámetros en cada nodo i : $NR_SLOTS=768$, $NR_NODES=32$; $FREE_SLOTS=4$; $0,5 \leq \lambda_i \leq 0,8$; $1 \leq \mu_i \leq 100$

En la simulación se introdujeron variaciones (cada 10000 rondas de simulador) del λ_i asignado a cada miembro. Aun así, la gráfica revela la estabilidad del algoritmo. Una vez alcanzado un estado de régimen, donde las ranuras fueron equitativamente distribuidas entre los diferentes miembros, la utilización media es del 73%.

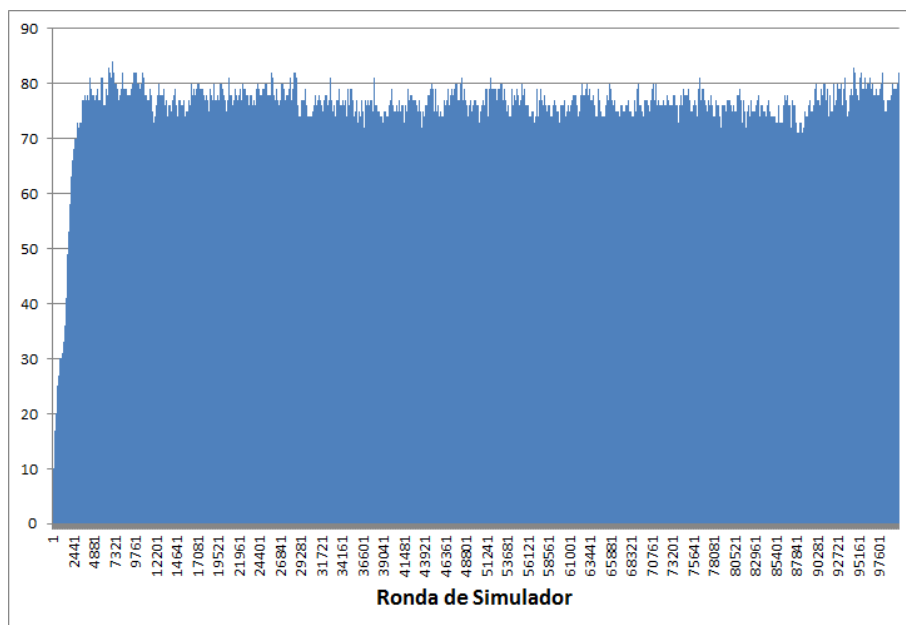


Figura 55.Utilización de Ranuras en [%].

El algoritmo de exclusión mutua tuvo una utilización media de recursos del 91%, pero logrado a consecuencia de una pérdida de eficacia con una mayor tasa de *forks()* fallidos y de la pérdida de eficiencia al requerir un mayor número de difusiones (Tabla XII).

TABLA XII.DESEMPEÑO DE SLOTS VS. EXCLUSIÓN MUTUA DISTRIBUIDA.

ALGORITMO	fork() Exitosos/ Difusión	fork() Fallidos/ fork() Exitosos	Utilización [%]
SLOTS	11,45	0,17	73
Exclusión Mutua	6,28	1,06	91

Como indicador de eficiencia se tomó la cantidad de *fork()* exitosos que se realizaron por cada difusión requerida por el protocolo. En la misma simulación este indicador arrojó una media de 11,45 frente a 6,28 del algoritmo de exclusión mutua.

En la Fig. 56 se presenta el gráfico comparativo de las peticiones de creación de procesos exitosas frente a aquellas que fueron fallidas como consecuencia de que el miembro no disponía de ranuras libres para asignar. En esa figura también evidencia la estabilidad del algoritmo.

En forma resumida, la Tabla XII indica que el 17% de las peticiones de *fork()* son fallidas frente a las de peticiones *fork()* exitosas. Para el algoritmo basado en exclusión mutua este indicador es del 106%.

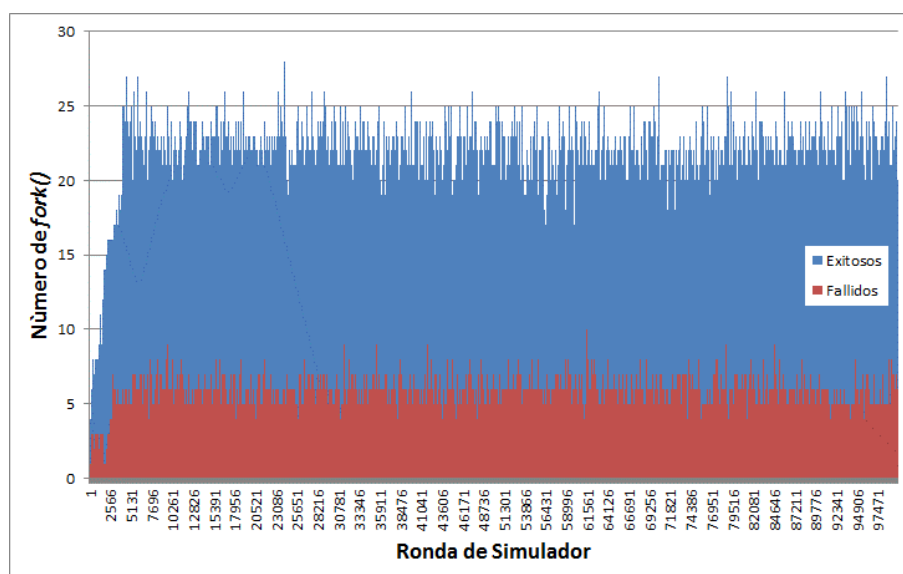


Figura 56. Cantidad de *fork()* existosos vs. *fork()* fallidos.

Para el análisis de estos valores se debe considerar que tanto la simulación de SLOTS, el algoritmo basado en exclusión mutua y el módulo que actualmente se encuentra ejecutando en el cluster de laboratorio no son bloqueantes, es decir que cuando se solicita crear un proceso y el miembro no dispone de ranuras, no se bloquea al miembro solicitante sino que se rechaza la petición. Las sucesivas solicitudes de *fork()* serán rechazadas hasta tanto el miembro reciba una donación de al menos un ranura libre o, finalice en el miembro un proceso liberando un ranura. Si se optara por una implementación bloqueante, entonces no habría peticiones de *fork()* fallidas sino peticiones demoradas que bloquearían al solicitante.

En la Tabla XII, se tomó como parámetro para ambos algoritmos la cantidad mínima de ranuras libres (*FREE_LOW*) que debe tener un miembro. Este parámetro se utiliza para decidir

cuándo un miembro debe solicitar la donación de ranuras, y para decidir si un miembro está en condiciones de donar ranuras. Actualmente, *FREE_LOW* es valor constante establecido para todos los miembros, pero este no es un condicionante del algoritmo. Se podrían establecer diferentes valores para cada miembro o establecer el valor dinámicamente a partir de la evaluación de algunas métricas.

El Tiempo de Respuesta es un indicador de la latencia del algoritmo para satisfacer la necesidad de recursos de un miembro. En la Fig. 57 se puede observar que los valores son mayoritariamente dos (79%) o tres (16%) tiempos de difusión (Δ_{mcast}). Esto significa que un miembro solicitante probablemente recibirá una donación de al menos una ranura en el primer o segundo mensaje de donación.

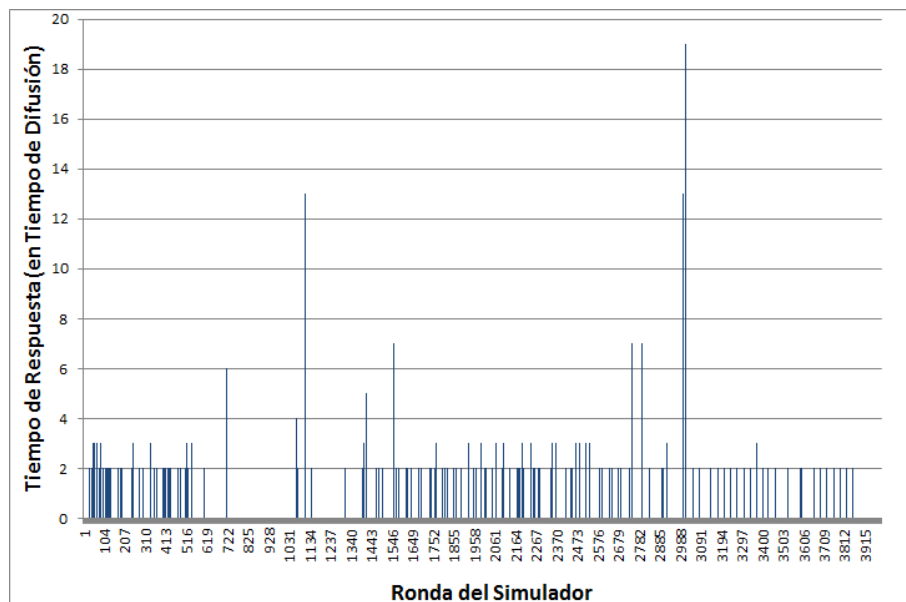


Figura 57. Tiempo de Respuesta a una Solicitud de Donación.

Los resultados de la simulación demuestran que SLOTS cumple con los requerimientos de rendimiento, eficiencia, estabilidad y elasticidad establecidos como objetivos.

SLOTS presenta una alta tasa de utilización de recursos combinada con un reducido tiempo de respuesta que satisfacen los requerimientos impuestos. El algoritmo presentado es simple y práctico, superador de otros puramente teóricos que dejan sin resolver aspectos esenciales para su implementación tales como los cambios de conformación del cluster y/o los fallos de procesos y de red. Actualmente, SLOTS se encuentra operando como componente del VOS multiservidor (SYSTASK de MoL) en el prototipo del DVS. El código de SLOTS en lenguaje C para Linux se encuentra disponible en <https://github.com/oajara/slots-donation>.

5.3. MOL-VDD

Para utilizar los servicios de MoL-VDD se debe hacer uso del protocolo M3-IPC desarrollado para el *DVS*. Para extender su utilización a procesos Linux, se desarrolló un módulo BUSE (<https://github.com/acozzette/BUSE>) que permite que Linux use el dispositivo (lo pueda montar como parte de su sistema de archivos) sin requerir la utilización de M3-IPC.

Como servidor de almacenamiento equivalente para contrastar el rendimiento de MoL-VDD se utilizó NBD [156]. NBD dispone de un componente cliente, que es un módulo gestor de disco que se instala en Linux para acceder a un dispositivo remoto. El componente servidor, en modo usuario, gestionará el dispositivo propiamente dicho (Fig. 58). A diferencia de MoL-VDD, NBD no soporta redundancia.

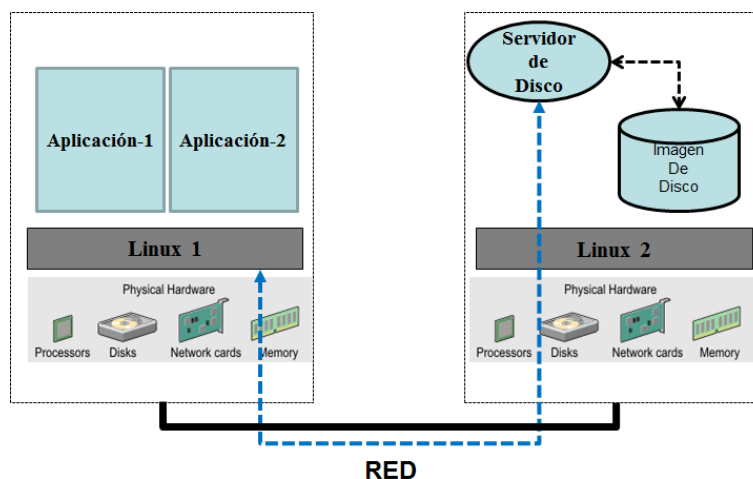


Figura 58. Esquema de funcionamiento de NBD

Para evaluar el rendimiento de MoL-VDD (con módulo BUSE) y de NBD se desarrollaron dos tipos de micro-benchmarks:

- *Tests Locales*: El cliente y el servidor se ejecutan en el mismo nodo.
- *Tests Remotos o en Cluster*: El cliente y el servidor se ejecutan en diferentes nodos.

Los micro-benchmarks utilizaron los comandos *time* y *dd* para realizar las transferencias de datos y tomar las mediciones. Se utilizaron archivos de 300 MBytes localizados en un disco RAM de Linux a fin de evitar la latencia propia de los discos rígidos. Los equipos utilizados fueron PCs Intel(R) Core(TM) i5 CPU 650 @ 3.20GHz, cache de 4096 KB y 4 GBytes de RAM y una red con un LAN switch dedicado de 1 Gbps por puerto.

Cuando en las Fig. 59 y Fig. 60 se referencia a **MoL-VDD** significa que la transferencia fue realizada desde un programa cliente que utiliza M3-IPC para transferir hacia y desde el MoL-VDD. Cuando se hace referencia a **BUSE** significa que la transferencia fue realizada por el cliente a través del módulo BUSE. Si se menciona **simple** indica que el servidor no trabaja replicado. Si se menciona **replicado**, indica que hay un segundo MoL-VDD de tipo Respaldo en otro nodo realizando las réplicas de las escrituras.

En la Fig. 59 se presentan los resultados arrojados por los micro-benchmarks en los que el cliente y servidor se ejecutan en el mismo equipo.

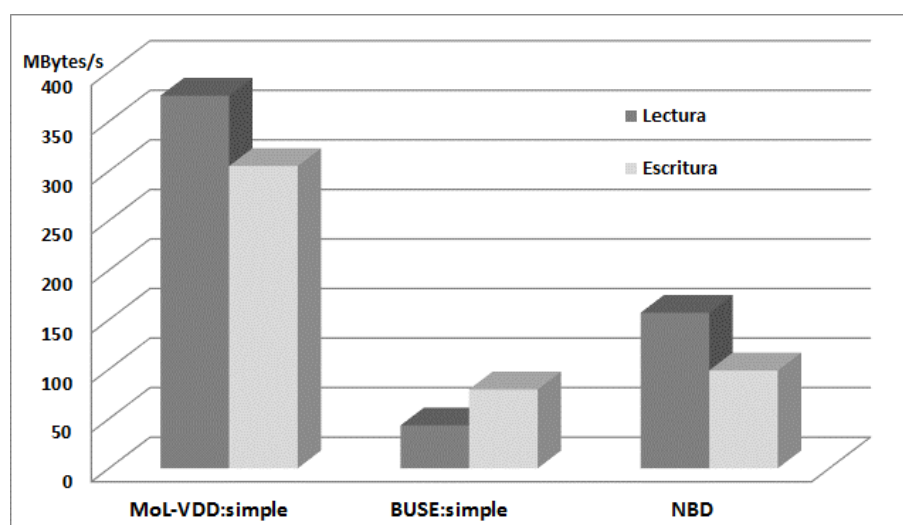


Figura 59. Rendimiento de MoL-VDD en pruebas Locales.

Los resultados muestran una diferencia importante de MoL-VDD dado que utiliza M3-IPC que se encuentra optimizado para su ejecución local. También se revela la importante sobrecarga que le impone utilizar el módulo BUSE. Esto se debe a que el módulo BUSE utiliza hilos y las facilidades de exclusión mutua y sincronización de hilos que ofrece Linux. Estas, al menos en la versión de kernel utilizada tienen un pobre desempeño que impacta directamente en las aplicaciones.

En la Fig. 60 se presentan los resultados de los micro-benchmarks en los que el cliente y servidor se ejecutan en diferentes nodos de un cluster.

Los resultados muestran una diferencia importante a favor de NBD. Esto se debe a que M3-IPC utiliza procesos *proxies* (en este caso con protocolo TCP) en modo usuario para las comunicaciones con nodos remotos, lo que implica 4 cambios de contexto adicionales entre cliente y servidor y 4 copias de los bloques de datos por cada transferencia de bloque de datos.

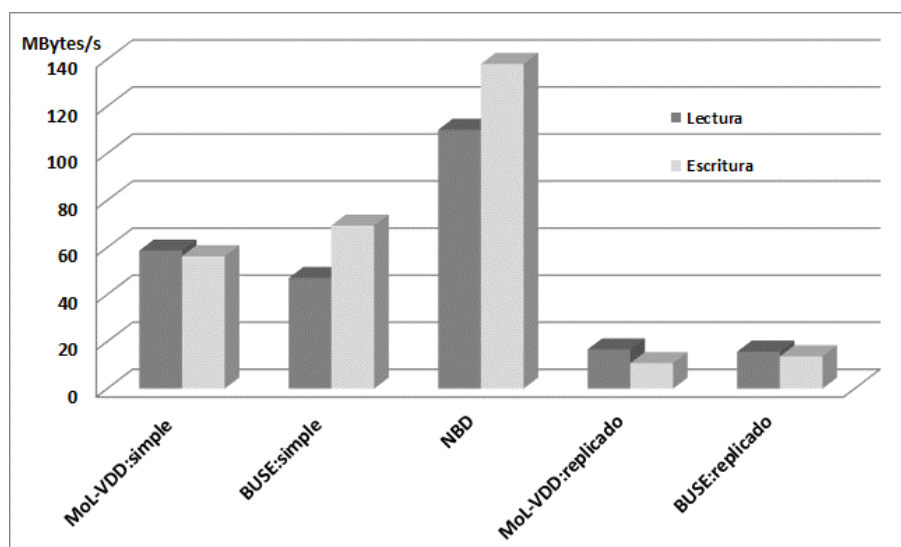


Figura 60. Rendimiento de MoL-VDD en pruebas Remotas.

Por otro lado, se puede apreciar el impacto sobre el rendimiento al habilitar la facilidad de replicación en MoL-VDD. Es razonable encontrar diferencias entre las lecturas y las escrituras dado que estas últimas deben replicarse. La diferencia en el rendimiento entre la ejecución *simple* y *replicado* se debe a que cuando MoL-VDD trabaja en modo *simple*, solo se crea un hilo en el proceso. Cuando trabaja en modo *replicado*, se crea un hilo adicional que es el encargado de realizar las actualizaciones entre Primario y Respaldo utilizando el GCS. Ambos hilos sincronizan el acceso a los recursos compartidos utilizando las facilidades que ofrece Linux, lo que impacta notablemente en el rendimiento.

Para el caso de MoL-VDD, solo se realizaron benchmarks que lo evaluaban considerando un único cliente. MoL-VDD soporta peticiones concurrentes de múltiples clientes, pero como en el prototipo de MoL sólo se utilizaría MoL-FS como tal, no se realizaron pruebas más complejas, posponiendo estas para trabajos futuros a medida que MoL evolucione.

5.4. MOL-FS

En la Sección 4.1.1 (Sistemas de Archivos del VOS multiservidor) se detallan varios escenarios operacionales donde se podría usar MoL-FS. En esta sección se presenta la evaluación de rendimiento de algunos de ellos.

Como MoL-FS se implementa en espacio de usuario, su rendimiento se comparó con otro sistema de archivos también de espacio de usuario. Se eligió un servidor NFS (UNFS [157]) y un

cliente NFS (HSFS [158]), ambos implementados en el espacio de usuario. Se deben considerar las sugerencias de [159] sobre las implicaciones de rendimiento de la ejecución de sistemas de archivos anidados.

La evaluación del rendimiento se realizó utilizando PCs AMD A6-3670 APU, 4 núcleos, 2.70GHz, 4 GBytes de RAM cada uno, y un switch dedicado de 1 Gbps por puerto.

Se usó el comando *cp* de Linux para realizar copias hacia y desde el servidor en el que los archivos de origen y destino se encontraban en un sistema de archivos de disco RAM a fin de evitar la latencia propia de los discos rígidos. Para proporcionar una variedad de muestreo, se usaron tres tamaños de archivo en los micro-benchmarks: 1, 10 y 100 Mbytes. Las pruebas de las operaciones de escritura no pudieron realizarse porque el par UNFS/HSFS sólo era compatible con el sistema de archivos de solo lectura (a la fecha de realización de las pruebas).

El primer conjunto de pruebas evalúa el rendimiento de las transferencias locales donde el cliente, el servidor y los datos comparten el mismo nodo (escenarios A1 y A2, Fig. 41). Los resultados se muestran de los tiempos de transferencias se presentan en la Tabla XIII, donde se evidencia que MoL-FS supera al NFS en modo de usuario. Por otro lado, queda claro que el rendimiento de las escrituras es significativamente menor que para las lecturas. Aunque existe una brecha importante en el rendimiento de MoL-FS al usar un archivo de imagen de disco en lugar de usar un VDD para las lecturas ([escenario A1 – Fig. 41](#)), no hay una diferencia notable para las escrituras ([escenario A2– Fig. 41](#)).

TABLA XIII. TIEMPO DE TRASFERENCIAS LOCALES [s].

ESCENARIO	1 Mbytes	10 Mbytes	100 Mbytes
NFS: Lectura Local	0,644	6,406	64,003
MoL-FS: Lectura A1	0,031	0,079	2,058
MoL-FS: Escritura A1	0,050	0,553	5,307
MoL-FS: Lectura A2	0,006	0,041	0,523
MoL-FS: Escritura A2	0,052	0,520	5,115

El segundo conjunto de pruebas evalúa el rendimiento de las transferencias de datos a través de la red (escenarios B, C1, y C2, Fig. 41). Se probaron varios escenarios operacionales, pero antes de comenzar los micro-benchmarks, se midió el rendimiento de la red con otras herramientas que arrojaron los siguientes resultados:

-*SSH scp*: informa 43 [MBytes/s].

-*TIPC*: una aplicación personalizada que utiliza informes TIPC de 81 [MBytes/s].

-*M3-IPC*: una aplicación personalizada que utiliza M3-IPC utilizando TIPC en *proxies* informa 58 [MBytes/s].

La Tabla XIV presenta los resultados de los tiempos de las transferencias remotas donde el archivo de origen y los archivos de destino de las copias están ubicados en diferentes nodos.

El rendimiento del NFS en espacio de usuario supera a todos los escenarios de MoL-FS para archivos de gran tamaño. Aunque los resultados no despiertan interés para adoptar Mol-FS como un sistema de archivos remoto, se debe considerar que se las pruebas se realizaron utilizando la pasarela FUSE que provoca una degradación importante del rendimiento [160]. FUSE utiliza hilos de Linux, y tal como se vio en otras pruebas de rendimiento, el uso de los mecanismos de exclusión mutua (mutexes) y de sincronización (variables de condición) hace caer significativamente el rendimiento del conjunto de hilos.

TABLA XIV. TIEMPO DE TRASFERENCIAS REMOTAS [s].

<i>ESCENARIO</i>	1 Mbytes	10 Mbytes	100 Mbytes
NFS: Lectura Remota	0,240	0,176	1,671
MoL-FS: Lectura C1	0,125	1,050	9,851
MoL-FS: Escritura C1	0,506	4,344	49,076
MoL-FS: Lectura C2	0,142	0,949	8,451
MoL-FS: Escritura C2	0,428	4,009	46,921
MoL-FS: Lectura B	0,159	1,118	11,309
MoL-FS: Escritura B	1,200	12,318	126,378

Otras pruebas con M3-IPC nativo que se realizaron con MoL-FS sin utilizar la pasarela FUSE, mostraron un rendimiento de transferencia único de 47 [Mbytes/s] resultando en 2,12 [s] para transferir un archivo de 100 [MBytes].

Hay dos características interesantes de MoL-FS y la utilización de la pasarela FUSE: en primer lugar, se puede acceder al mismo tiempo al Sistema de Archivos desde procesos Linux ordinarios y desde procesos de un VOS contenido en un *DC* (utilizando M3-IPC). En segundo lugar, varios servidores pueden ejecutarse en el mismo VOS asignándole a cada uno un *endpoint* diferente.

Otro uso que puede darse a MoL-FS es para entornos de laboratorios y desarrollo en donde se requieren espacios de almacenamiento de archivos estancos para cada grupo de trabajo. Es posible entonces disponer de múltiples servidores contenidos cada uno en su propio *DC*, y cada uno de los clientes con pasarela FUSE en el mismo *DC* de su servidor, pero en distintos nodos. De esta forma, cada aplicación cliente accede a su propio sistema de archivos, incluso con privilegios de usuario *root*.

Para el caso de MoL-FS, solo se realizaron benchmarks que lo evaluaban considerando un único cliente. MoL-FS soporta peticiones concurrentes de múltiples clientes, pero como en el

prototipo de MoL sólo se utilizaría MoL-NWEB u otro utilitario, no se realizaron pruebas mas complejas, posponiendo estas para trabajos futuros a medida que MoL evolucione.

5.5. MOL-NWEB

Los benchmarks se realizaron sobre un servidor web (*nweb*) ejecutando sobre dos tipos de VOS: 1) un VOS unkernel (ukVOS) y 2) un VOS multiservidor (MoL). Además, se consideraron diferentes escenarios para su ejecución: 1) centralizada en un único Nodo y 2) distribuida en dos Nodos. El cliente web (*wget*) estaba ubicado en el mismo DC y en el mismo Nodo que el servidor web. Los archivos de imagen de disco utilizados en los benchmarks se ubicaron en discos RAM para evitar las latencias propias del disco rígido.

Las pruebas de rendimiento solo se limitaron al tiempo de transferencia de archivos, por lo que se requiere realizar evaluaciones más completas, las cuales consideren el número de peticiones simultáneas que el sistema soporta hasta la saturación de alguno de sus componentes, la utilización de CPU, latencia, etc. Para los benchmarks, en los que todos los procesos comparten el mismo nodo (Fig. 61), se plantearon tres escenarios.

En Config-A (Fig. 61, A), el servidor web obtiene sus archivos del sistema de archivos de Linux localizado en un disco RAM, y representa una medición de referencia. En Config-B (Fig. 61, B), el servidor web obtiene los archivos de MoL-FS utilizando un archivo de imagen de disco. En Config-C (Fig. 61, C), el servidor web obtiene los archivos de MoL-FS, el cual utiliza un MoL-VDD como servidor de almacenamiento para obtener datos de un archivo de imagen.

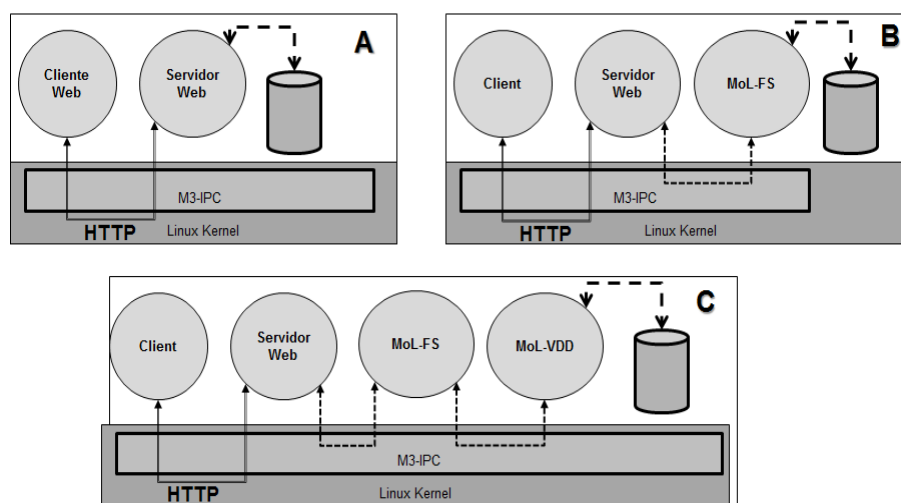


Figura 61. Escenarios de evaluación de rendimiento web en el mismo Nodo.

Los tres procesos se ejecutan en el mismo *DC* y en el mismo nodo, usando M3-IPC entre ellos. La Tabla XV muestra los resultados de rendimiento de los benchmarks donde todos los procesos comparten el mismo nodo.

Existe una diferencia de rendimiento evidente entre ejecutar el servidor web en ukVOS y ejecutarlo en MoL. Esto sugiere que la implementación de pila de protocolo TCP/IP en modo de usuario de *LwIP* con hilos de Linux no es eficiente, siendo esto un factor común en todas las pruebas realizadas. El rendimiento de MoL para aquellas configuraciones en las que los componentes del proceso se encuentran en el mismo nodo (Config-B y Config-C) es algo menor que si se ejecuta el servidor web directamente en Linux (Config-A).

TABLA XV. RENDIMIENTO EN TRANSFERENCIA DE ARCHIVOS DEL SERVIDOR WEB (MISMO NODO).

Tamaño de Archivo [Mbytes]	<i>VOS Unikernel (ukVOS)</i>			<i>VOS Multi-server (MoL)</i>		
	Config-A [Mbytes/s]	Config-B [Mbytes/s]	Config-C [Mbytes/s]	Config-A [Mbytes/s]	Config-B [Mbytes/s]	Config-C [Mbytes/s]
10	7,44	7,03	7,04	74,62	70,00	67,54
50	6,86	6,76	6,85	81,43	79,10	79,52
100	7,96	6,82	6,72	97,11	92,13	92,31

Para realizar los benchmarks en los que los procesos se encuentran distribuidos en dos nodos se utilizaron 3 escenarios (Fig. 62).

En Config-D (Fig. 62, D), el servidor web (Nodo0) obtiene los archivos de MoL-FS (Nodo0), el cual obtiene datos de MoL-VDD (Node1) utilizando un archivo de imagen de disco. En Config-E (Fig. 62, E), el servidor web (Nodo0) obtiene los archivos de MoL-FS (Nodo1) obteniendo los datos de un archivo de imagen de disco. En Config-F (Fig. 62, F), el servidor web (Nodo0) obtiene los archivos de MoL-FS (Nodo1) el cual utiliza MoL-VDD (Nodo1) como fuente de sus bloques de disco. La Tabla XVI muestra los resultados de rendimiento de los benchmarks realizados en un servidor web y otros procesos relacionados en el mismo *DC*, pero en diferentes nodos.

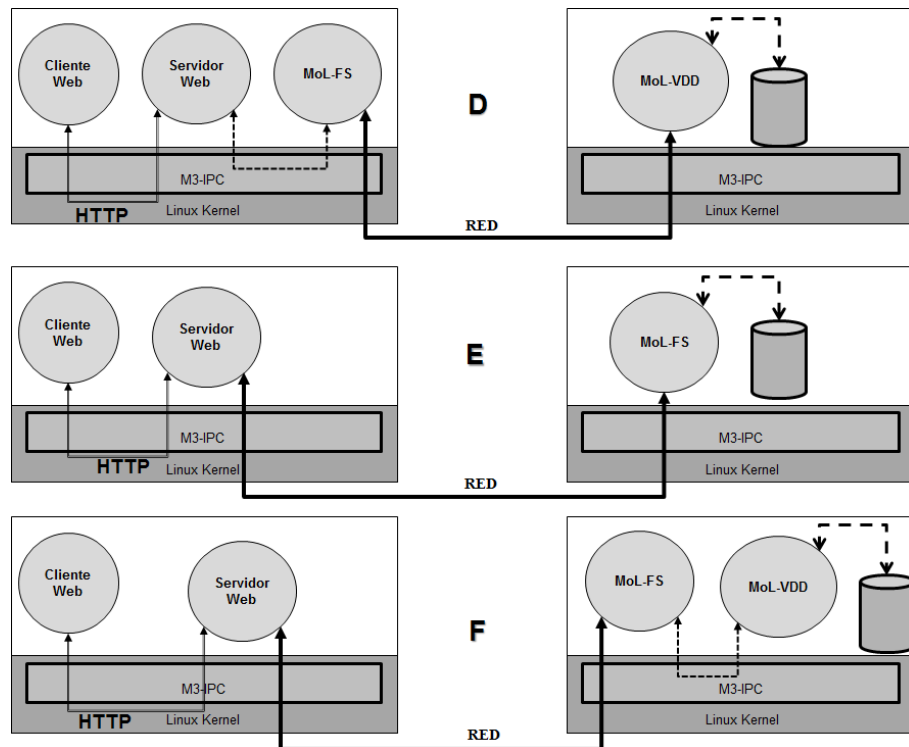


Figura 62. Escenarios de evaluación de rendimiento web en dos Nodos.

Los *proxies* de espacio de usuario utilizados en los benchmarks utilizaban TCP en modo usuario. Se esperan mejores resultados al utilizar *proxies* TIPC o Ethernet RAW de acuerdo con la evaluación de desempeño M3-IPC, como así también por la compresión de datos y el loteo de mensajes.

TABLA XVI. RENDIMIENTO EN TRANSFERENCIA DE ARCHIVOS DEL SERVIDOR WEB (DOS NODOS).

Tamaño de Archivo [Mbytes]	Unikernel (ukVOS)			Multi-server (MoL)		
	Config-D [Mbytes/s]	Config-E [Mbytes/s]	Config-F [Mbytes/s]	Config-D [Mbytes/s]	Config-E [Mbytes/s]	Config-F [Mbytes/s]
10	2,62	2,90	2,75	4,08	3,81	3,51
50	2,72	2,89	2,88	4,32	3,75	3,57
100	2,70	2,94	2,74	4,32	3,79	3,55

5.6. Resumen de la Evaluación del Prototipo

El DVS es un modelo de arquitectura teórico para el cual se desarrolló un prototipo como prueba de concepto para demostrar su factibilidad. El mismo fue desarrollado principalmente por el autor de esta tesis con la colaboración de otros investigadores y alumnos. Durante el desarrollo se focalizaron los objetivos en la factibilidad, correctitud y funcionalidades del prototipo. Como objetivo secundario se estableció que el rendimiento debía ser comparable con productos similares existentes,

teniendo en consideración las diferencias que pudiesen existir entre ellos. Se realizaron esfuerzos en aspectos relacionados a la optimización del rendimiento solo en aquellos casos en que éste fuera comparativamente considerado inaceptable o injustificable.

Se emplearon diversas metodologías para evaluar el rendimiento de los diferentes componentes del prototipo del DVS. Para M3-IPC, en transferencias locales de mensajes y bloques de datos, se evaluó la tasa máxima de transferencia entre un par Cliente/Servidor, (parámetro fundamental para el desempeño de los VOS). También se evaluó el desempeño llevando todos los núcleos de la CPU a la saturación utilizando múltiples pares Cliente/Servidor a fin de obtener la tasa máxima de transferencia total. Para el caso de las transferencias remotas de mensajes y bloques de datos se evaluó la tasa máxima de transferencia para un solo par de procesos Cliente/Servidor. Actualmente se dispone de los micro-benchmarks que permiten ejecutar múltiples pares Cliente/Servidor que permitirían llevar el sistema hasta la saturación de la red. Respecto a los componentes de MoL, se evaluó el algoritmo de donación de ranuras en aquellas métricas consideradas importantes para la aplicación. Para ello, se utilizó un simulador a los fines de poder realizar múltiples pruebas alterando diferentes parámetros de ejecución. Tanto MoL-VDD, MoL-FS, MoL-NWEB como el Unikernel UK-VOS fueron evaluados solo considerando la tasa máxima de transferencia de archivos.

En general, no se utilizaron micro-benchmarks reconocidos para evaluar el rendimiento. Muchas de estas herramientas no eran compatibles con el sistema a evaluar y requerían de modificaciones cuyo impacto no se podía prever. En cambio, se optó por utilizar como herramientas de evaluación programas básicos de Linux, tales como el *cp*, *tar*, *dd*, *nweb*, etc. que expresan de mejor forma el rendimiento que percibe el usuario final.

Los resultados de las evaluaciones presentadas en este capítulo dependen del componente analizado. En algunos casos los resultados superaron las expectativas y en otros el rendimiento fue apenas discreto, pero considerado como aceptable sabiendo que permanecen muchos factores que pueden optimizarse. Por ejemplo, no se presentan aquí las evaluaciones de rendimiento de transferencias de mensajes y bloques de datos de M3-IPC utilizando algoritmos de compresión en los proxies que mejoran el rendimiento en las transferencias entre nodos y el loteado (batching) de mensajes en las transferencias entre proxies. La compresión no solo mejora los tiempos de transferencia sino también la utilización de la red, a cambio de un mayor uso de CPU en los nodos. El loteado de mensajes reduce tanto la utilización del ancho de banda de la red, la utilización de CPU por mensaje transferido y la latencia de los mensajes cuando se opera con alta demanda.

De igual forma, tampoco se presentaron aquí los resultados de la compresión de los mensajes de replicación difundidos por MoL-VDD a través de Spread Toolkit. En definitiva, quedan varios aspectos relacionados al rendimiento que merecen un análisis mas detallado y dedicado.

Como durante el diseño del modelo de arquitectura de *DVS* se fue desarrollando el prototipo, el cual incluye una gran cantidad de componentes, a medida que éstos se iban implementando forzaban a realizar modificaciones en otros componentes previamente evaluados. Este hecho produce un desfase entre la evaluación de un componente aislado y la evaluación de otros componentes que utilizan al primero, como por ejemplo MoL-FS utiliza los servicios de MoL-VDD, y ambos utilizan M3-IPC. En paralelo se realizaron cambios, correcciones y optimizaciones en algunos componentes que no fueron evaluados rigurosamente, pero ésto forma parte de la dinámica que propone el modelo y su prototipo.

Actualmente, el prototipo del *DVS* (*DVK*, bibliotecas, proxies, VOS, etc.) está siendo migrado a una versión más moderna de kernel de Linux (4.9.88) que impacta directamente en los recursos que utiliza respecto a la versión aquí presentada. Este hecho obligará a realizar nuevas evaluaciones del rendimiento de los diferentes componentes.

6. TRABAJOS FUTUROS

El modelo de arquitectura de *DVS* admite la aplicación de diferentes enfoques y estrategias para su implementación. Es posible que a medida que se realizan cambios, ampliaciones e incorporación de funcionalidades al prototipo, éstas revelen la necesidad de realizar modificaciones al modelo. Lo que si se evidencia es la amplitud de posibilidades que brinda esta nueva tecnología de virtualización. A continuación, se describen algunos trabajos planeados a futuro para mejorar el modelo de arquitectura de un *DVS* y del prototipo.

En primer lugar, aún permanecen abiertas cuestiones tales como la gestión completa del prototipo de *DVS* desde *Webmin*. Para ello, sería conveniente disponer de una aplicación distribuida que permita realizar la gestión desde cualquier nodo sin necesidad de utilizar scripts (como se hace actualmente) y utilizando archivos de configuración. Esto también permitirá integrar de manera más elegante los servicios del *DVS* a sistemas de gestión de Servicios en la Nube ampliamente utilizados, tal como *Openstack* o *Kubernetes*. Para ello habrá que focalizar la investigación y el desarrollo en aspectos relativos a la seguridad para que el *DVS* esté en concordancia con los requerimientos de estos sistemas de gestión con calidad de proveedor para brindar servicios en la Nube.

El módulo del *DVK* del prototipo presentado en esta tesis fue desarrollado para la versión de kernel de Linux 2.6.32. Para tener los beneficios de las muchas características de los nuevos kernels, se está procediendo a migrar el *DVK* a una de estas versiones (4.9.88), lo que permite disponer de mejores características en lo que a Contenedores se refiere, disponer de software actualizado y de nuevos servicios que pueden incorporarse al *DVK*. Otra cuestión en curso relacionado al *DVK* es la implementación de sus APIs utilizando la Llamada al Sistema *ioctl()*, aun cuando esto implique un impacto negativo en el rendimiento. Su objetivo es evitar la necesidad de modificar el kernel para implementar el conjunto de llamadas al *DVK*, facilitando así su mantenimiento futuro.

Como se mencionó en su oportunidad, el GCS del prototipo no se encuentra integrado al módulo del *DVK* por lo que no es posible controlar los accesos y por lo tanto brindar un mejor aislamiento entre los procesos de diferentes *DCs*. Se propone entonces realizar la integración de las APIs del *Spread Toolkit* a las APIs del *DVK*.

Actualmente los *DCs* se construyen en base a las APIs del *DVK* y en forma separada utilizando herramientas disponibles en Linux para gestionar Contenedores individuales. Es más conveniente integrar todo este conjunto de operaciones en una única operación que interactúe con el sistema de gestión distribuido del *DVS*.

En cuanto al aislamiento de los procesos que ejecutan dentro de un *DC*, queda por explorar la posibilidad de interceptar las Llamadas al Sistema del proceso utilizando *ptrace* en la coraza de aislamiento, tal como lo hace en la primera versión de *UML*. Otra posibilidad es instalar un módulo de kernel el cual intercepte las Llamadas al Sistema Linux de procesos registrados en el *DVK* y las transforme en transferencia de mensajes. Para reforzar el aislamiento de los procesos dentro de los *DCs*, se pueden utilizar las facilidades de *seccomp* [161] en los kernels actuales de tal modo de reducir la superficie de ataque.

Para los próximos dos años, se planean desarrollar los siguientes proyectos que pueden producir un gran impacto en la adopción del modelo de *DVS*: 1) Ejecutar *UML* como *VOS* dentro de un *DC*. 2) Ejecutar un *VOS Unikernel* generado con *rumpkernel* con soporte *POSIX* dentro de un *DC*. Ambos podrían utilizar servicios distribuidos en otros nodos del mismo *DC*. Estos proyectos demostrarían que un Linux en modo usuario (*UML*) o un *Unikernel* basado *NetBSD* pueden adaptarse como *VOS* de un *DVS*, aumentando así la confianza en el modelo propuesto.

7. CONCLUSIONES

Los proveedores de IaaS siempre requieren de altos niveles de rendimiento, escalabilidad y disponibilidad para brindar sus servicios de virtualización. Estos requerimientos se pueden satisfacer con una tecnología de virtualización distribuida. El modelo de arquitectura de *DVS* propuesto combina e integra las tecnologías de virtualización y de los Sistemas Distribuidos para proporcionar los beneficios de ambos mundos, lo que lo hace adecuado para ofrecer Servicios en la Nube con nivel de proveedor. La arquitectura se basa en poner a disposición de los VOS y aplicaciones mecanismos de IPC, GCS y servicios autónomos, distribuidos y débilmente acoplados otorgándoles flexibilidad para los desarrollos de nuevos servicios y aplicaciones, pero manteniendo las características de aislamiento requeridas.

El modelo de arquitectura de *DVS* propone utilizar Contenedores en los nodos que componen el cluster de virtualización, pero no en la forma tradicional en la que las aplicaciones tienen acceso directo a las facilidades que ofrece el OS-host. En un *DVS*, un *DC* está constituido por un conjunto de Contenedores dispersos en diferentes nodos. Un *DC* provee a las aplicaciones y a los VOS de los mecanismos de comunicación transparentes que no consideraran la localización de los diferentes procesos. Los *DCs* permiten integrar procesos de diferentes nodos a través de IPC y otras facilidades, mientras que los Contenedores que lo conforman aíslan, en cada uno de los nodos, a los procesos que ejecutan dentro del Contenedor del resto de los procesos que ejecutan en el host. Además, los Contenedores que componen un *DC*, son utilizados para asignarle recursos físicos al conjunto de sus procesos en cada uno de los nodos (memoria, tasa máxima de operaciones de E/S, tasa máxima de utilización de la red, buffers, caches, etc.). Es decir, el *DC* brinda facilidades a los procesos que ejecutan en él y cada Contenedor que lo compone provee el aislamiento de seguridad y de rendimiento al asignarle y limitarle los recursos a esos procesos en cada uno de los nodos.

La propuesta de utilización de VOS en el modelo de arquitectura del *DVS* no solo le aporta mayor aislamiento al impedir que las aplicaciones accedan directamente a las facilidades del OS-host, sino que también le otorga la flexibilidad de poder ejecutar diferentes VOS en el *DVS*. Esto permite ejecutar aplicaciones desarrolladas para diferentes OS cuyo VOS equivalente esté disponible en el *DVS*.

Otra característica que hace atractivo a un *DVS* es que facilita la migración/portación de aplicaciones heredadas desde servidores locales (on-premise) a la Nube. Con un VOS que suministre

las API POSIX y RPC permitiría la reutilización de gran parte del código de esas aplicaciones, mejorando los costos y tiempos de implementación al reducir el esfuerzo de codificación. Por otro lado, también se adapta perfectamente a las aplicaciones basadas en el MSA o SOA, éstas pueden ejecutar un conjunto de microservicios en varios nodos del clúster mediante el uso de IPC/RPC propio del *DVS* para las comunicaciones.

Para verificar el diseño, detectar errores y proponer mejoras se desarrolló un prototipo de *DVS* que permitió comprobar la viabilidad del modelo propuesto. Uno de los productos más importantes que arrojó el prototipo es M3-IPC, como mecanismo de comunicaciones entre procesos. M3-IPC resuelve algunos problemas relacionados con el soporte de subprocesos (threads), la transparencia de la ubicación, la redirección de mensajes en la migración de procesos, el protocolo de transporte de red independiente y el confinamiento de IPC para la virtualización. Los resultados muestran que M3-IPC logra todos sus objetivos de diseño, con un alto rendimiento tanto para los mensajes y transferencias de datos intra-nodo e inter-nodo.

Para comprobar la factibilidad de la propuesta, se desarrollaron dos VOS: MoL, que es multiservidor y que puede ejecutarse en forma centralizada o distribuida gracias a las facilidades que le aporta M3-IPC, y ukVOS que es de tipo unikernel. Para transformar a MoL en un DOS se debió desarrollar un algoritmo distribuido de asignación de ranuras el cual utiliza los servicios de un GCS para replicación, detección de fallos, elección de líder, etc., enfatizando la utilidad de la arquitectura.

Con un *DVS* se da respuesta a las dos preguntas planteadas en la Introducción:

1.¿Cómo se pueden expandir el poder de cómputo y la utilización de recursos de una VM o Contenedor a varios computadores?

Los límites de un entorno aislado para la ejecución de aplicaciones (*DC*) pueden expandirse a todos los nodos del clúster (agregación) mejorando sus características de rendimiento, escalabilidad y disponibilidad. Además, los nodos de un cluster pueden compartirse entre varios *DCs* (partición), mejorando así la utilización de la infraestructura de hardware.

2.¿Cómo lograr mayores niveles de rendimiento y escalabilidad de las aplicaciones que se ejecutan en la Nube?

La metodología actual para lograr que las aplicaciones obtengan mayores niveles de rendimiento y escalabilidad consiste en utilizar un gestor de contenedores para desplegar un “enjambre” de ellos en los cuales se ejecutan los componentes de las aplicaciones y sus

réplicas. Con esta metodología, los programadores y administradores que deben implementar, desplegar, operar, gestionar y controlar cada aplicación carecen de una visión integrada, lo que incrementa los tiempos y costos. A diferencia de los Gestores de Contenedores, un VOS distribuido ejecutando dentro de un DC aporta una visión integrada de las aplicaciones equivalente a un sistema centralizado evitando que el programador deba considerar la localización de cada componente en el cluster, su monitoreo, su direccionamiento (direcciones IP, puertos TCP, etc), su seguridad (reglas de firewall), etc. Esta visión integrada facilita la gestión reduciendo los tiempos y los costos de implementación, operación y mantenimiento de aplicaciones en la Nube.

El modelo de arquitectura propuesto abre la posibilidad de investigar y desarrollar varios aspectos relacionados tanto al *DVS* propiamente dicho, como de algunos de los componentes tales como los VOSs. Por ejemplo, el *DVK* adopta un esquema basado en servicios autónomos, se podría pensar entonces que un VOS también podría adoptar tal como lo hace un OS de microkernel o multikernel, pero con servicios autónomos independientes del VOS.

Las futuras etapas de investigación y desarrollo se centrarán en realizar mejoras orientadas a proporcionar servicios de virtualización escalables, elásticos, de alto rendimiento y de alta disponibilidad y fácilmente gestionables.

Glosario

API (Application Programming Interface)	Interface de programación de aplicaciones. Es el conjunto de subrutinas, procedimientos y funciones que ofrece un software para poder ser utilizado por otro software.
AWS (Amazon Web Services)	Servicios Web de Amazon. Servicios de plataforma de computación en la Nube que ofrece AMAZON a través de internet
BaaS (Backend as a Service)	Servicios que se ofrecen en la Nube para complementar otras aplicaciones.
Backup	Copia o Equipo de Respaldo.
Bitmap	Mapa de Bits. Estructura de datos que generalmente se utiliza para representar el estado (0,1) de gran número de elementos.
Cgroups (Control groups)	Funcionalidad que ofrecen los Linux actuales que permiten asignar recursos discrecionalmente y controlar su utilización a un conjunto de procesos.
CLI Command Line Interface	Interface de línea de comandos.
Cluster	Conjunto de computadores unidos por una red de alta velocidad.
Cookies	Estructura de datos de pequeño tamaño que se envía desde un servidor web al navegador para conservar información de actividad anterior.
CoW (Copy on Write)	Copia en Escritura- Cuando un proceso solicita la copia de un bloque de datos, este no se copia realmente sino que se referencia. En el momento que uno de los procesos que tiene acceso a la estructura realiza una operación de escritura, se remueve un enlace y se realiza la copia efectivamente.
Datacenters	Espacio físico especialmente acondicionado en donde se localizan recursos informáticos y de comunicaciones.
DB	Base de Datos. Es una metodología de almacenar los datos de tal forma de facilitar su recuperación y acceso.
DBMS - DataBase Management Server	Servidor Gestor de Base de Datos
DC (Distributed Container)	Es un entorno de ejecución aislado y distribuido que ofrece recursos computacionales, virtuales y abstractos a un conjunto de procesos.
Device Drivers	Gestores o Controlador de Dispositivos Es un módulo de software que interactúa sobre un dispositivo de hardware y actúa como interface para un software en una capa superior (generalmente un Sistema Operativo)
DOS, Distributed Operating System	Sistema Operativo Distribuido. Es un sistema Operativo que abarca los nodos de un cluster permitiendo que las aplicaciones se ejecuten en diferentes nodos.
DVK (Distributed Virtualization Kernel)	Componente del DVS que se ejecuta distribuido en cada Nodo. Es el responsable de gestionar todos los recursos del DVS y suministrar

	servicios a los VOS que se ejecutan en el dominio de un DC.
DVS (Distributed Virtualization System)	Sistema de Virtualización Distribuido el cual permite construir dominios de ejecución denominados Contenedores Distribuidos (DC:Distributed Containers) que pueden extenderse abarcando múltiples nodos de un cluster.
FAT (File Allocation Table)	Tabla de Asignación de Archivos- Sistema de Archivos sencillo desarrollado originalmente para MS-DOS.
FIFO (First In- First Out)	Primero que Entra, Primero que Sale - En UNIX es un mecanismo de IPC que permite que un proceso inserte datos en un orden dado y otro proceso pueda extraerlos en el mismo orden.
FTP (File Transfer Protocol)	Protocolo de Transferencia de Archivo - Es un protocolo de transferencia de archivos estándar del TCP/IP que utiliza protocolo TCP.
FUSE (Filesystem in User Space)	Sistema de Archivos en Espacio de Usuario - Facilidad que ofrece Linux para dar tratamientos a sistemas de archivos pero sin incluir su módulo en el kernel. Un proceso en espacio de usuario es invocado por el kernel para acceder al sistema de archivos.
Hipervisor (Hypervisor)	Es el software que permite la implementación de máquinas virtuales, su creación, arranque detención, etc.
Host	Computador anfitrión en donde reside una aplicación que es utilizada por usuarios conectados a él o por otros procesos ejecutando en otros computadores.
HTTP HyperText Transfer Protocol	Protocolo de Transferencia de Hipertexto- Permite la transferencia de información entre un clientes, servidores y proxies.
IaaS - Infrastructure as a Service	Infraestructura como Servicio - Servicio en la Nube que proporciona recursos informáticos tales como computadores, switches, firewalls, sistemas operativos, etc. que pueden ser virtuales o físicos.
IDS/IPS - Intrusion Prevention/Detection System	Sistema de Detección/Prevención contra Intrusos – Existen IDS e IPS de hosts y de red (Network IDS/IPS)
IPC - (Interprocess Communications)	Comunicacion entre procesos - Mecanismos de comunicacion que ofrece un sistema operativo que permite el intercambio de datos y/o la sincronización de procesos.
JVM (Java Virtual Machine)	Máquina Virtual Java - Es una máquina virtual específicamente construida para un proceso generado por el compilador JAVA.
LwIP - (Light weight IP)	Software que implementa el stack de protocolos TCP/IP que puede incluirse como parte de otros programas.
LXC/LXD - Linux Containers	Software que permite gestionar los contenedores de Linux desde la shell.
Many-copres	Microprocesadores con muchos núcleos (mayor a 100)
MESOS - (Apache Mesos)	Software que permite gestionar clusters
Microservicios	Es una técnica de desarrollo de software que propone factorizar las aplicaciones en componentes (microservicios) independientes que se comunican entre si
Minix	Sistema Operativo de microkernel desarrollado por Andrew Tanenbaum
Mutexes	Mecanismo de sincronización y exclusión mutua que se utiliza para evitar condiciones de competencia entre diferentes procesos o hilos.
Namespaces	Espacio de Nombres- Es el conjunto de nombres válidos por los que pueden ser reconocidos los objetos de un sistema.
NAS (Network Attached Storage)	Es un servidor de archivos dedicado conectado a una red y al que pueden conectarse clientes heterogéneos.
NIC (Network Interface Card)	Tarjeta de hardware que se puede integrar a un computador y que permite conectarlo a una red.
NIST (National Institute of Standards	Organismo estadounidense que establece estándares en tecnología.

and Technology)	
PaaS - (Platform as a Service)	Plataforma como Servicio - Servicios en la nube que ofrecen herramientas y entornos para el desarrollo de aplicaciones.
Patch	Parque - Módulo de software que permite reparar o modificar un programa o sistema.
Pipe	Tubería - Similar a FIFO - En Unix las Tuberías con nombre (named pipes) son FIFOs.
POSIX	Portable Operating System Interface – Estandar para sistemas operativos.
Proxy	Representante. Es un proceso que actúa como intermediario entre 2 procesos. Para cada uno de los procesos extremos el proxy representa al otro proceso.
RAID (Redundant Array of Independent Disk)	Es una tecnología del almacenamiento que utiliza múltiples discos físicos y que permite componer discos lógicos con diferentes características de redundancia.
SAN - (Storage Area Network)	Es una red que provee de almacenamiento a nivel de bloques. Incluye tanto a discos como a dispositivos de cinta.
SCC - (Single-chip Cloud Computer)	Es un computador compuesto por múltiples núcleos, con memoria propia y compartida pero que internamente conforman una cluster.
Scripts	Código fuente de lenguaje interpretado.
Señales	Mecanismo por el cual un sistema operativo notifica de eventos a una aplicación.
Semáforos	Mecanismo de IPC utilizado para realizar exclusión mutua y sincronización de procesos.
Sincronía Virtual Extendida (EVS)	Define un mecanismo de comunicaciones grupales que permite recuperar cuando se producen caída de computadores o particiones de red.
SLA - (Service Level Agreement)	Es el acuerdo contractual entre el proveedor de un servicio y su cliente.
SMP - (Simetric Multiprocessing)	Es un multiprocesador en el cual varios procesadores idénticos en jerarquía comparaten bus y memoria
SOA (Service-oriented architecture)	Principios y metodología para el desarrollo de software basado en servicios ofrecidos por componentes.
spinlocks	Mecanismo de exclusión mutua utilizado en arquitecturas SMP.
SSI	Sistema distribuido que se presenta al usuario como un unico computador.
Switch (network)	Dispositivo de red LAN de capa 2 (Enlace) que trabaja en modo bridge (puente) pero que dispone de múltiples puertos.
TAP (Terminal Access Point)	Dispositivo de red virtual que emula una NIC.
TIPC (Transparent Inter Process Communication)	Protocolo de IPC diseñado para utilizar en clusters.
Unikernel	Es un programa que incluye aplicaciones, pila de protocolos, bibliotecas y código de gestores de dispositivos en un unico espacio de direcciones y que puede ejecutar sobre un hipervisor.
VDD - (Virtual Disk Driver)	Programa que se comporta como un gestor de dispositivos pero que opera sobre un dispositivo virtual.
VM - (Virtual Machine, Máquina Virtual)	Abstracción que emula un computador.
VMM - (Virtual Machine Monitor)	idem hipervisor.
vNIC - (Virtual Network Interface Card)	Tarjeta de LAN virtual. Abstracción de un hipervisor o un sistema operativo que emula el comportamiento de una NIC. Generalmente se implementa utilizando algún mecanismo de IPC.

VOS	Sistema Operativo Virtual - Sistema operativo que no incluye la gestión de dispositivos ni hardware real, lo hace con dispositivos virtuales.
Web Services	Mecanismo de comunicación entre programas que utiliza protocolo HTTP.

Referencias

- [1] P. M. Mell, T. Grance, “*The NIST Definition of Cloud Computing*”, Technical Report, NIST, SP 800-145, 2011.
- [2] ScaleSMP, “*vSMP Foundation Architecture*”, WhitePaper, Available online at <http://www.scalemp.com/media-hub/resources/white-papers>, 2013, accedido en Enero 2018.
- [3] J. Gallard, A. Lèbre, G. Vallée, P. Gallard, S. Scott, C. Morin, “*Is Virtualization Killing Single System Image Research?*”, Research Report RR-6389, INRIA, 2007.
- [4] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, “*Microservices in Practice, Part 1: Reality Check and Service Design*”, IEEE Softw. 34, pp 91-98, Jan. 2017, 2017.
- [5] N. Bieberstein et al., “*Service-Oriented Architecture Compass*”, Pearson, ISBN 0-13-187002-5, 2006.
- [6] J. Turnbull, “*The Docker Book*”, 2014, Available online at: <https://www.dockerbook.com/>, accedido en Enero 2018.
- [7] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and Ion Stoica, “*Mesos: a platform for fine-grained resource sharing in the data center*”, Proc. of the 8th USENIX conference on Networked systems design and implementation (NSDI'11), USENIX Association, Berkeley, CA, USA, pp. 295-308, 2011.
- [8] N. Poulton, “*The Kubernetes Book*”, ISBN-13: 978-1521823637, ISBN-10: 1521823634, 2017.
- [9] D. Wentzlaff, C. Gruenwald III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, A. Agarwal, “*An operating system for multicore and clouds: mechanisms and implementation*”, in Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10). ACM, New York, NY, USA, 3-14, 2010.
- [10] C. Wang, F. C. Lau, and W. Zhu, “*JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support*”, 2013 IEEE International Conference on Cluster Computing (CLUSTER), Chicago, Illinois, 2002, pp. 381, doi: 10.1109/CLUSTER.2002.1137770.
- [11] E. Sirer, R. Grimm, B. N. Bershad, A. J. Gregory, S. Mcdirmid. “*Distributed Virtual Machines: A System Architecture for Network Computing*”, In Proceedings of the Eighth ACM SIGOPS European Workshop, 1998.
- [12] Amazon Lambda, <https://aws.amazon.com/es/lambda/>, accedido Enero 2018.
- [13] N. Pappas, “*Network IDS & IPS Deployment Strategies*”, SANS Institute, April 2, 2008.
- [14] M. Chapman, G. Heiser, “*vNUMA: a virtual shared-memory multiprocessor*”, in Proc. of the 2009 conference on USENIX Annual technical conference (USENIX'09), USENIX Association, Berkeley, CA, USA, pp. 2-2, 2009.
- [15] K. Kaneda, Y. Oyama, A. Yonezawa, “*A virtual machine monitor for utilizing non-dedicated clusters*”, in proc. of the twentieth ACM symposium on Operating systems principles (SOSP '05), ACM, New York, NY, USA, pp. 1-11, doi: <https://doi.org/10.1145/1095810.1118618>.
- [16] TildaScale, <https://www.tildascale.com/>, accedido Enero 2018.
- [17] D. Hall, D. Scherrer, J. Svntek, “*A Virtual Operating System*”, Journal Communication of the ACM, 1980.
- [18] S. Galley, “*PDP-10 Virtual Machines*”, in Proc. ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, Cambridge, MA, 1969.
- [19] A. Popek, R. Goldberg, “*Formal Requirements for Virtualizable Third Generation Architectures*”. Communications of the ACM 17 (7): 412–421. 1974.
- [20] G. Banga, P. Druschel, J. C. Mogul, “*Resource Containers: A New Facility for Resource Management in Server Systems*”, in Operating Systems Design and Implementation, 1999.
- [21] D. Price, A. Tucker, “*Solaris Zones: Operating System Support for Consolidating Commercial Workloads*”, in 18th Large Installation System Administration Conference, 2004.
- [22] P. Kamp, R. N. M. Watson, “*Jails: Confining the omnipotent root*”, in Proc. 2nd Intl. SANE Conference, 2000.
- [23] J. S. Robin, C. E. Irvine, “*Analysis of the Intel Pentium's ability to support a secure virtual machine monitor*”, in Proceedings of the 9th conference on USENIX Security Symposium - Volume 9 (SSYM'00), Vol. 9. USENIX Association, Berkeley, CA, USA, 10-10, 2000.
- [24] K. Adams, “*A comparison of software and hardware techniques for x86 virtualization*”, in ASPLOS-XII: Proceedings of the 12th international conference on Architectural2, 2006.
- [25] J. Fisher-Ogden, “*Hardware support for efficient virtualization*”, UC San Diego Report, USA, 2006.
- [26] K. Adams, O. Agesen, “*A Comparison of Software and Hardware Techniques for x86 Virtualization*”, International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA. ACM 1-59593-451-0/06/0010, 2006.
- [27] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, D. Magenheimer, “*Are virtual machine monitors microkernels done right?*”, in Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10 (HOTOS'05), Vol. 10. USENIX Association, Berkeley, CA, USA, 1-1, 2005.

- [28] G. Heiser, V. Uhlig, J. LeVasseur, "Are virtual-machine monitors microkernels done right?", SIGOPS Oper. Syst. Rev. 40, 1 (January 2006), 95-99. DOI=<http://dx.doi.org/10.1145/1113361.1113363>, 2006.
- [29] F. Armand, M. Gien, "A Practical Look at Micro-Kernels and Virtual Machine Monitors", in 6th IEEE CCCN Conference, 2009.
- [30] P. Barnham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugerberger, I. Pratt, A. Warfield, "Xen and the art of virtualization", in SOSPP '03: Proceedings of the nineteenth ACM symposium on Operating system principles, pages 164-177, 2003.
- [31] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, "KVM: the Linux Virtual Machine Monitor", In Proceedings of the 2007 Ottawa Linux Symposium (OLS '07), 2007.
- [32] Microsoft Hyper.V, [https://msdn.microsoft.com/es-es/library/hh831564\(v=ws.11\).aspx](https://msdn.microsoft.com/es-es/library/hh831564(v=ws.11).aspx), accedido Enero 2018.
- [33] Oracle Virtual Box, <https://www.virtualbox.org/>, accedido Enero 2018.
- [34] Z. Amsden, D. Arai, D. Hecht, A. Holler, P. Subrahmanyam, "Vmi: an Interface for Paravirtualization", Ottawa Linux Symposium, 2006.
- [35] E. Bugnion, S. Devine, M. Rosenblum, "Disco: running commodity operating systems on scalable multiprocessors", in Proceedings of the Sixteenth ACM Symposium on Operating System Principles, October 1997.
- [36] A. Whitaker, M. Shaw, S. D. Gribble, "Denali: Lightweight Virtual Machines for Distributed and Networked Applications", In Proceedings of the USENIX Annual Technical Conference, 2002.
- [37] [37] T.[T] Hohmuth M.; Peter M., et al. "Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors"; EW 11; 2004.
- [38] B. Ford, M. Hibler, et al, "Microkernels meet recursive virtual machines", Second Symposium on OSDI, 1996.
- [39] S. Biemüller, U. Dannowski, "L4-Based Real Virtual Machines - An API Proposal", in Proceedings of the MIKES 2007.
- [40] P. Pessolani, "Virtualization Extensions into a Microkernel based Operating System", CONAISI 2014.
- [41] A. Tanenbaum, et al, "Minix 3: Status Report and Current Research", login: The USENIX Magazine, 2010.
- [42] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors", in Proc. 2nd ACM European Conference on Computer Systems, 2007.
- [43] W. Felber, R. Ferreira, R. Rajamony, J. Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers", IBM Research Report, 2014.
- [44] Linux CGroups, <https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt>, accedido Enero 2018.
- [45] Linux Containers, <https://linuxcontainers.org/>, accedido Enero 2018.
- [46] OpenVZ, https://openvz.org/Main_Page, accedido Enero 2018.
- [47] Linux VServer, http://linux-vserver.org/Welcome_to_Linux-VServer.org, accedido Enero 2018.
- [48] Windows Containers, <https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/>, accedido Enero 2018.
- [49] Y. Yu, F. Guo, S. Nanda, L. Lam, T. Chiueh. "A feather-weight virtual machine for windows applications", in Proceedings of the 2nd international conference on Virtual execution environments (VEE '06). ACM, New York, NY, USA, 24-34. 2006.
- [50] P. Kamp, R. N. M. Watson, "Jails: Confining the omnipotent root", in Proc. 2nd Intl. SANE Conference, 2000.
- [51] J. Dike, "A user-mode port of the Linux kernel", USENIX Association. Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta Oct 10-14, 2000.
- [52] D. Aloni, "Cooperative Linux", in Proc. of the Linux Symposium, 2004.
- [53] S. M. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. W. Trickey, P. Winterbottom, "The Inferno™ operating system", in Bell Labs Technical Journal, vol. 2, no. 1, pp. 5-18, Winter 1997.
- [54] Inferno - <http://www.vitanuova.com/inferno/docs.html>, accedido Enero 2018.
- [55] P. Pessolani, O. Jara, "Minix over Linux: A User-Space Multiserver Operating System", in Proc. Brazilian Symposium on Computing System Engineering, Florianopolis, 2011, pp. 158-163, doi: 10.1109/SBESC.2011.17.
- [56] P. Pessolani; T. Cortes; F. Tinetti; S. Gonnet, "Un Mecanismo de IPC de microkernel embebido en el kernel de Linux". XV Workshop de Investigadores en Ciencias de la Computación. Paraná. 2013.
- [57] P. Pessolani, T. Cortes, F. G. Tinetti, and S. Gonnet, "An IPC Software Layer for Building a Distributed Virtualization System", Congreso Argentino de Ciencias de la Computación (CACIC 2017) La Plata, Argentina, October 9-13, 2017.
- [58] K. Lawton et al., "The Bochs IA-32 Emulator Project", <http://bochs.sourceforge.net>, accedido Enero 2018.
- [59] F. Bellard, "Qemu: a fast and portable dynamic translator", in Proceedings of the USENIX Annual Technical Conference, ATEC '05, pages 41-46, Anaheim, California, USA, 2005.
- [60] DOSbox, <http://www.dosbox.com/information.php?page=0>, accedido Enero 2018.
- [61] Wine, <https://www.winehq.org/help>, accedido Enero 2018.
- [62] A. S. Tanenbaum, "Sistemas Operativos Distribuidos", Prentice Hall, ISBN 9789688806272, 1995.
- [63] D. R. Cheriton, "The V Distributed System", in Communications of the ACM 31 (3): 314-333, March 1988.
- [64] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, H. van Staveren, "Amoeba: A Distributed Operating System for the 1990s", Vrije Universiteit, May 1990.
- [65] Sprite, <https://www2.eecs.berkeley.edu/Research/Projects/CS/sprite.html>, Accedido Enero 2018
- [66] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, W. Neuhauser, "Overview of the CHORUS Distributed Operating Systems", Computing Systems, 1991.
- [67] G. Popek, B. Walker, "The LOCUS Distributed System Architecture", MIT Press, 1985.
- [68] A. Barak, "The MOSIX Multicomputer Operating System for High Performance Cluster Computing", Journal of Future Generation Computer Systems, 1998.
- [69] OpenSSI (Single System Image) Clusters for Linux, <http://www.openssi.org/cgi-bin/view?page=openssi.html>, accedido Enero 2018.
- [70] D. Margery, G. Vallee, R. Lottiaux, C. Morin, J. Berthou, "Kerrighed: A SSI Cluster OS Running OpenMP", in Proc. 5th European Workshop on OpenMP (EWOMP '03), 2003.

- [71] Xtremos. <http://www.xtremos.eu/>, accedido Enero 2018.
- [72] D. Golub, R. Dean, A. Forin, R. Rashid, “*Unix as an Application Program*”, Proceedings of the USENIX Summer Conference, 1990.
- [73] D. Wentzlaff, A. Agarwal, “*Factored operating systems (fos): the case for a scalable operating system for multicores*”, SIGOPS Oper. Syst. Rev. 43, 2 (April 2009), 76-85. DOI=<http://dx.doi.org/10.1145/1531793.1531805>
- [74] D. R. Engler, M. F. Kaashoek, J. O’Toole Jr., “*Exokernel: an operating system architecture for application-level resource management*”, in Proc. 15th ACM Symposium on Operating Systems Principles (SOSP), pages 251–266, Copper Mountain, CO, USA, December 3–6 1995.
- [75] A. Madhavapeddy, D. Scott, “*Unikernels: The rise of the virtual library operating systems*”, Communications of the ACM (CACM) 57, 1(Jan. 2014), 61–69.
- [76] N. Linnenbank, “*Implementing Minix on the Single Chip Cloud Computer*”, Master Thesis, Vrije Universiteit, 2011.
- [77] Shang Rong Tsai, Ru Jing Chen, “*Interprocess communication with multicast support in DMINIX operating system*”, Microprocessing and Microprogramming, Volume 32, Issues 1–5, 1991.
- [78] A. Bricker, M. Gien, M. Guillemont, J. Lipkis, D. Orr, M. Rozier, “*Architectural issues in microkernel-based operating systems: the Chorus experience*”, Computer Communications volume 14, number 6, pages 347 - 357, 1991.
- [79] P. Pessolani, “*MINIX4RT: a real-time operating system based on MINIX*”, Master Thesis. 2006.
- [80] G. Heiser, K. Elphinstone, “*L4 Microkernels: The Lessons from 20 Years of Research and Deployment*”, ACM Trans. Comput. Syst., vol. 34, Article 1, April 2016, doi: <http://dx.doi.org/10.1145/2893177>.
- [81] The QNX Neutrino Microkernel, http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/kernel.html , accedido Enero 2018.
- [82] M. Děcký, “*Component-based General-purpose Operating System*”, in Proceedings of WDS’07, Prague, Czech Republic, June 2007.
- [83] K. P. Birman, “*The process group approach to reliable distributed computing*”, Commun. ACM 36, Dec. 1993, pp. 37-53, doi:<http://dx.doi.org/10.1145/163298.163303>.
- [84] K. P. Birman, “*Building Secure and Reliable Network Applications*”, Manning Publications Co., Greenwich, CT, USA, 1997.
- [85] Waseem Ahmed, Yong Wei Wu, “*A survey on reliability in distributed systems*”, Journal of Computer and System Sciences, Volume 79, Issue 8, 2013.
- [86] L. Lamport, “*Paxos made simple*”, ACM SIGACT News 32, 4 (Dec. 2001), 18-25.
- [87] D. Ongaro, J. Ousterhout, “*In search of an understandable consensus algorithm*”, in Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC’14), Garth Gibson and Nikolai Zeldovich (Eds.), USENIX Association, Berkeley, CA, USA, 305-320, 2014.
- [88] V. Hadzilacos, S. Toueg., “*Fault-tolerant broadcasts and related problems. In Distributed systems (2nd Ed.)*”, Sape Mullender (Ed.). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA 97-145, 1993.
- [89] Y. Artsy, R. Finkel, “*Designing a process migration facility: the Charlotte experience*”, in Mobility, Dejan Milojicic & cacute, Frederick Douglas, and Richard Wheeler (Eds.). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA 89-108, 1999.
- [90] Jihun Kim, Dongju Chae, Jangwoo Kim, Jong Kim, “*Guide-copy: fast and silent migration of virtual machine for datacenters*”, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC ’13). ACM, New York, NY, USA, , Article 66 , 12 pages. DOI: <https://doi.org/10.1145/2503210.2503251>, 2013.
- [91] Checkpointing and live migration, https://openvz.org/Checkpointing_and_live_migration, accedido Enero 2018.
- [92] C. Lu., “*Process Migration in Distributed Systems*”, Ph.D. Dissertation. University of Illinois, Urbana, IL, USA. UMI order no: GAX89-16278, 1989.
- [93] V. Medina, “*A survey of migration mechanisms of virtual machines*”, ACM Computing Surveys Fall, pp. 30–33, 2016.
- [94] S. Nadgowda, S. Suneja, N. Bila, C. Isci, “*Voyager: Complete Container State Migration*”, 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 2137-2142, 2017.
- [95] R. Buyya, Toni Cortes, Hai Jin, “*Single System Image*”, Int. J. High Perform. Comput. Appl. 15, 2 (May 2001), 124-135. DOI=<http://dx.doi.org/10.1177/109434200101500205>, 2001.
- [96] C. Clark, K. Fraser, H. Steven, J. Gorm Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield, “*Live Migration of Virtual Machines*”, In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2005.
- [97] Checkpoint/Restore In Userspace, https://criu.org/Live_migration , accedido Enero 2018.
- [98] Container migration tool, <https://github.com/marcosnils/cmt> , accedido Enero 2018.
- [99] N. Budhiraja, K. Marzullo, F. B. Schneider, S.Toueg, “*The primary-backup approach*”, in Distributed systems (2nd Ed.), Sape Mullender (Ed.). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA 199-216, 1993.
- [100] F. B. Schneider, “*Implementing fault-tolerant services using the state machine approach: a tutorial*”, ACM Comput. Surv. 22, 4 (December 1990), 299-319. DOI=<http://dx.doi.org/10.1145/98163.98167>, 1990.
- [101] M. J. Fischer, N. A. Lynch, M. S. Paterson, “*Impossibility of distributed consensus with one faulty process*”, J. ACM, 32(2):374–382, 1985.
- [102] P. Pessolani, T. Cortes, F. Tinetti, S. Gonnet, “*An Architecture Model for a Distributed Virtualization System*”, Cloud Computing 2018, The Ninth International Conference on Cloud Computing, GRIDS, and Virtualization, Barcelona, 2018.
- [103] P. Pessolani, T. Cortes, F. Tinetti, S. Gonnet, “*Un sistema de virtualización distribuida*”, XIX Workshop de Investigadores en Ciencias de la Computación (WICC 2017), Buenos Aires, 2017.
- [104] H. Ong, J. Vetter, R. S. Studham, C. McCurdy, B. Walker, A. Cox., “*Kernel-level single system image for petascale computing*”, SIGOPS Oper. Syst. Rev. 40, 2 (April 2006), 50-54. DOI=<http://dx.doi.org/10.1145/1131322.1131335>, 2006.
- [105] J. Dongarra, “*Report on the Sunway TaihuLight System*”, www.netlib.org , accedido Enero 2018.
- [106] R. F. van der Wijngaart, T. G. Mattson, W. Haas, “*Light-weight communications on Intel’s single-chip cloud computer processor*”, SIGOPS Oper. Syst. Rev. 45, 1 (February 2011), 73-83, DOI=<http://dx.doi.org/10.1145/1945023.1945033>, 2011.

- [107] R. Van Renesse, K. P. Birman, S. Maffei, "Horus: A flexible group communication system", Comm. of the ACM, 1996.
- [108] K. Birman, H. Sohn, "Hosting dynamic data in the cloud with Isis2 and the Ida DHT", in Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS '13). ACM, New York, NY, USA, , Article 10 , 15 pages. 2013.
- [109] S. Dake, C. Caulfield, A. Beekhof, "The Corosync Cluster Engine", in Linux Symposium, 2008.
- [110] The Spread Toolkit. <http://www.spread.org> , accedido Enero 2018.
- [111] Y. Amir, C. Danilov, M. M. Amir, J. Schultz, J. Stanton, "The Spread toolkit: Architecture and performance", Technical Report CNDS-2004-1, Johns Hopkins University, Center for Networking and Distributed Systems, Baltimore, MD, USA, 2004.
- [112] R. F. Rashid, "An inter-process communication facility for UNIX", Computer Science Department. Paper 2417, 1980.
- [113] G. Zellweger, S. Gerber, K. Kourtis, T. Roscoe, "Decoupling Cores, Kernels, and Operating Systems", in Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014), October 2014.
- [114] S. J. Mullender, G. van Rossum, A. S. Tananbaum, R. van Renesse, H. van Staveren, "Amoeba: a distributed operating system for the 1990s", in Computer, vol. 23, no. 5, pp. 44-53, May 1990.
- [115] Barrera, J., "A Fast Mach Network IPC Implementation", in Proceedings of the USENIX Mach Symposium, November 1991.
- [116] J. Collins, R. Findlay, "Programming the SIMPL Way", ISBN 0557012708, 2008.
- [117] SRR- QNX API compatible message passing for Linux. <http://www.opcdatahub.com/Docs/booksr.html>
- [118] M. Sharifi, K. Karimi, "DIPC: A Heterogeneous Distributed Programming System", in Proceedings of the 3rd Annual Computer Conference of the Computer Society of Iran, 1997
- [119] A. D. Birrell, B. Jay Nelson, "Implementing remote procedure calls", ACM Transactions on Computer Systems, 1984
- [120] J. P. Maloy, "TIPC: Providing Communication for Linux Clusters", Proceedings of the Linux Symposium, 2004
- [121] B. N. Bershad, T. E. Anderson, E. D. Lazowska, H. M. Levy, "User-level interprocess communication for shared memory multiprocessors", ACM trans. Comput. Syst. 9, 2 (May 1991), 175-198.
- [122] D-Bus, <https://lwn.net/Articles/484203/> , accedido Enero 2018.
- [123] F. Duignan, "CORBA, DCOP and DBUS. A performance comparison", Dublin Institute of Technology (accedido Enero 2018), <http://eleceng.dit.ie/frank/rpc/CORBAGnomeDBUSPerformanceAnalysis.pdf> , 2002.
- [124] R. Slominski, "Fast User/Kernel Data Transfer", Master Thesis, April 20, 2007.
- [125] L. Veraldi, "Efficient Capability-Based Messaging (ECBM)", <http://web.tiscali.it/lucavera/www/root/ecbm/>, 2003 (Italian).
- [126] A. S. Tanenbaum, A. S. Woodhull, "Operating Systems Design and Implementation (3rd Edition)", ISBN: 0131429388, 2006.
- [127] P. Marandi, M. Primi, N. Schiper, F. Pedone, "Ring Paxos: A high-throughput atomic broadcast protocol", in DSN, 2010.
- [128] K. Ostrowski, K. Birman, D. Dolev, "QuickSilver Scalable Multicast (QSM)", in 7th IEEE International Symposium on Network Computing and Applications (IEEE NCA 2008), Cambridge, MA. July 2008.
- [129] Y. Amir, C. Danilov, J. Stanton, "A low latency, loss tolerant architecture and protocol for wide area group communication", in Proceedings of the International Conference on Dependable Systems and Networks, pp. 327-336, June 2000.
- [130] L. E. Moser , Y. Amir, P. M. Melliar-Smith, D. A. Agarwal, "Extended Virtual Synchrony", in Proceedings of the IEEE 14th International Conference on Distributed Computing Systems (Poznan, Poland, June), IEEE Computer Society Press, 1994.
- [131] Webmin, <http://www.webmin.com/> , accedido Enero 2018.
- [132] Módulo DVS para Webmin, <https://github.com/lucasleichhorn/WebminDRVS> , accedido Enero 2018.
- [133] Openstack, <https://www.openstack.org/> , accedido Enero 2018.
- [134] LwIP, <http://savannah.nongnu.org/projects/lwip/> , accedido Enero 2018.
- [135] <https://www.techpowerup.com/238514/intel-cpu-on-chip-management-engine-runs-on-minix> , accedido Enero 2018.
- [136] J. N. Herder, "Design and Implementation of a Reliable Operating System", Proc. First EuroSys Doctoral Workshop, Brighton, England, Oct. 2005.
- [137] D. Padula, M. Alemandi, P. Pessolani, S. Gonnnet, T. Cortes, F. Tinetti, "A User-space Virtualization-aware Filesystem", 3er Congreso Nacional de Ingeniería Informática/Sistemas de Información (CoNaIISI 2015), Buenos Aires, 2015.
- [138] Filesystem in Userspace, <http://fuse.sourceforge.net/> , accedido Enero 2018.
- [139] FatFs, http://elm-chan.org/fsw/ff/00index_e.html , accedido Enero 2018.
- [140] Nweb, <https://github.com/ankushagarwal/nweb> , accedido Enero 2018.
- [141] Rumpkernel, <http://rumpkernel.org/> , accedido Enero 2018.
- [142] V. Diekert, P. Gastin, "Local safety and local liveness for distributed systems", in Perspectives in Concurrency Theory, IARCS-Universities, pages 86-106. Universities Press, 2009.
- [143] B. Alpern, F. B. Schneider, "Recognizing Safety and Liveness", Distributed Computing, 1986.
- [144] P. Pessolani, T. Cortes, F. G. Tinetti, S. Gonnnet, O. Jara, "A Fault-Tolerant Algorithm For Distributed Resource Allocation", IEEE Latin America Transactions (Volume: 15, Issue: 11), Nov. 2017.
- [145] M. Alemandi, O. Jara, "Un driver de disco tolerante a fallos", Jornada de Jóvenes Investigadores Tecnológicos (JIT 2015), Rosario, 2015.
- [146] LZ4, <https://github.com/lz4/lz4> , accedido Enero 2018.
- [147] M. Kerrisk, "The Linux Programming Interface", No Starch Press, ISBN 978-1-59327-220-3, 2010
- [148] ipc-bench, <http://www.cl.cam.ac.uk/research/srg/netos/ipc-bench/> , accedido Enero 2018.
- [149] Google RPC, <https://grpc.io/blog/principles> , accedido Enero 2018.

-
- [150] S. Boyd-wickizer, A. T. Clements, O. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich, "An Analysis of Linux Scalability to Many Cores", Proceedings of the 9th USENIX, OSDI'10, 2010.
- [151] S. Gopal, N. Gupta, R. Jain, S. R. Kamatham, P. R. L. Eswari, "Experiences in porting TCP application over UDT and their performance analysis", 2013 IEEE International Conference in MOOC, Innovation and Technology in Education (MITE), Jaipur, pp. 371-374, 2013.
- [152] J. Fan, W. Zhou, "A Distributed Resource Allocation Algorithm in Multiservice Heterogeneous Wireless Networks", in Wireless Internet - 7th International {ICST} Conference, WICON 2013, China, 2013.
- [153] W. Schreiner, "A java toolkit for teaching distributed algorithms", In Proceedings of the 7th annual conference on Innovation and technology in computer science education (ITiCSE '02), ACM, 2002.
- [154] Z. Liu, N. Niclausse, C. Jalpa-Villanueva, "Traffic model and performance evaluation of Web servers", in Performance Evaluation, 46, 2-3 October 2001.
- [155] M. Harchol-balter, A. B. Downey, "Exploiting process lifetime distributions for dynamic load balancing", ACM Transactions on Computer Systems, 1997.
- [156] NBD, <http://nbd.sourceforge.net/>, accedido Enero 2018.
- [157] A User-space NFS Server, <http://unfs3.sourceforge.net/>, accedido Enero 2018.
- [158] HSFS - an NFS client via FUSE. <https://github.com/openunix/hsfs>, accedido Enero 2018.
- [159] D. Le, H. Huang, and H. Wang, "Understanding Performance Implications of Nested File Systems in a Virtualized Environment" in USENIX Conference on File & Storage Technologies (FAST), 2012.
- [160] B. Kumar, R. Vangoor, V. Tarasov, E. Zadok, "To FUSE or not to FUSE: performance of user-space file systems", in Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST'17), USENIX Association, Berkeley, CA, USA, 59-72, 2017.
- [161] SECure COMputing with filters, https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt, accedido Enero 2018.
- [162] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhanian. "The multikernel: a new OS architecture for scalable multicore systems". In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09). ACM, New York, NY, USA, 29-44. 2009.