A Fault-Tolerant Algorithm For Distributed Resource Allocation

P. Pessolani, O. Jara, S. Gonnet, T. Cortés and F. G. Tinetti

***Abstract*— Resource allocation is a usual problem that must be faced during a distributed system design. Despite the large number of algorithms proposed in literature to solve this problem, most papers lack of detailed descriptions about how to turn these algorithms into real-world reliable protocols. This article presents a fault-tolerant algorithm for distributed resource allocation named *SLOTS* which is implemented as an executable protocol. It allocates resources among members in a fairly manner using simple heuristics and employing a *donation* approach. *SLOTS* supports the dynamic behavior of clusters and provides high availability services. It bases its fault-tolerance properties and membership changes in atomic sets of operations (like transactions) using services provided by an underlying Group Communication System.**

***Keywords*— Distributed Systems, Distributed Resource Allocation, Fault Tolerance.**

I.INTRODUCCION

LOS SISTEMAS distribuidos basados en el paradigma de *grupo de procesos* usualmente requieren asignar instancias de un mismo recurso entre todos miembros que conforman el grupo. Algunos ejemplos de problemas de este tipo se presentan en la asignación de ranuras de tiempo/frecuencia en un enlace compartido, en la asignación de canales de un medio de comunicación, en la asignación de un conjunto de bloques de disco en un sistema de archivos distribuidos, en asignación de rangos de puertos de protocolos TCP/UDP de un servidor distribuido, o en la asignación del rango de direcciones IP entre servidores [1]. Los algoritmos que resuelven estos problemas deben satisfacer la propiedad de exclusión mutua y evitar la inanición de los procesos por falta de asignación de recursos. Estas propiedades si bien necesarias, no son suficientes para una implementación práctica de un algoritmo que debería asegurar una alta disponibilidad del servicio y garantizar la consistencia en la asignación de recursos aún ante la ocurrencia de fallos (de procesos, de nodos o de red).

En este artículo se propone un algoritmo distribuido de asignación dinámica de recursos que pretende anticiparse a la demanda de aquellos miembros con mayores necesidades. Este problema, que a-priori parece sencillo de resolver, se complica al considerar la ocurrencia de fallos tales como la detención de un proceso miembro o, del nodo donde éste reside o, una partición de la red que une los nodos que conforman el cluster. Para que el algoritmo sea de utilidad práctica, debe mantener la consistencia de recursos asignados aún ante la ocurrencia de fallos o ante cambios en la topología de la red.

La asignación de recursos es un problema típico de los Sistemas Distribuidos que requiere de Consenso Distribuido (DC) o equivalentes (Consistencia Interactiva, Acuerdo Bizantino, Difusión Confiable con Terminación). Fischer, Lynch y Paterson [2] demostraron que es imposible alcanzar DC de manera determinística en sistemas asincrónicos aún ante la ocurrencia de un simple fallo de detención. Esta imposibilidad no significa que no se puede alcanzar DC en presencia de fallos, refiere a que no se puede garantizar un límite de tiempo para alcanzarlo. Ante estos resultados, podría ejecutarse DC cada vez que un miembro del grupo requiere de un recurso y volver a ejecutar DC cuando el recurso se libera. No obstante, para problemas donde la tasa de solicitud y liberación de recursos es alta, esta estrategia resultaría en un aumento considerable del tiempo de respuesta e incrementaría la utilización de la red como consecuencia de la complejidad de mensajes de los algoritmos de DC.

Debido a las características dinámicas de los clusters donde se ejecutan Sistemas Distribuidos, para el problema en cuestión, deben realizarse las siguientes consideraciones:

1. Durante la ejecución de la aplicación distribuida, pueden incorporarse nuevos miembros al grupo de procesos que requieren de la asignación de una mínima cantidad de recursos.
2. Los miembros más demandados requieren de una mayor cantidad de recursos asignados para satisfacer sus requerimientos temporales.
3. Los recursos asignados a procesos miembros que terminan en forma controlada o en forma abrupta (p.e. ante un fallo de software), pero cuyos nodos permanecen operativos, deberían considerarse libres para luego poder ser reutilizados.
4. Los recursos libres asignados a miembros con baja demanda temporal deberían poder reasignarse a aquellos miembros con mayor demanda.
5. Se pueden producir particiones en la red que une los miembros, aún ante esta situación, los sub-grupos de procesos resultantes deberían mantenerse operativos conservando la consistencia en la asignación de recursos. Cuando la red recupera su topología original, el procesamiento debe continuar tal como lo hacía previo a la partición.

P. Pessolani, Universidad Tecnológica Nacional, Santa Fe, Argentina, ppessolani@hotmail.com.

O. Jara, Universidad Tecnológica Nacional, Santa Fe, Argentina, oajara@gmail.com.

S. Gonnet, Instituto de Desarrollo y Diseño INGAR, CONICET-UTN, Santa Fe, Argentina, sgonnet@santafe-conicet.gov.ar.

T. Cortes, Barcelona Supercomputing Center (BSC) y Universitat Politècnica de Catalunya, toni.cortes@bsc.es.

F. G. Tinetti, Facultad de Informática, UNLP, CIC prov. Bs. As., Argentina, fernando@info.unlp.edu.ar.

Corresponding author: Pablo Pessolani.

El algoritmo distribuido que aquí se presenta se denomina *SLOTS*. Una implementación práctica del mismo se ejecuta actualmente como componente de un Sistema de Virtualización Distribuida [3] para brindar servicios Cloud tipo IaaS. En este sistema los recursos compartidos a asignar son *descriptores de procesos* que deben asignarse dinámicamente entre los nodos de un cluster de virtualización. Cada descriptor de proceso está contenido en un *slot.* Se asignan conjuntos de *slots* a los diferentes nodos para contener los descriptores de los procesos que se ejecutarán en esos nodos. Por la criticidad de los servicios Cloud IaaS, se requiere que los algoritmos toleren fallos para brindar servicios de alta disponibilidad, transformándose éste en un aspecto inevitable en las consideraciones de diseño. Si bien en la descripción del algoritmo propuesto se hacen referencias al caso de uso del mismo, éste debe considerarse de aplicación general para resolver otros tipos de problemas equivalentes en sistemas distribuidos que impliquen la asignación de recursos idénticos reutilizables. Como ejemplos de este tipo de recursos pueden mencionarse a un conjunto de direcciones IPs, a puertos TCP/UDP, a los bloques de discos de sistemas de almacenamiento, y a los canales (de tiempo o de frecuencia) de un enlace de comunicaciones.

SLOTS soporta las características dinámicas de los clusters y basa sus propiedades de tolerancia a fallos en un enfoque transaccional que utiliza los servicios de una plataforma subyacente de Servicios de Comunicaciones Grupales (GCS). La decisión de utilizar un GCS como base para implementar el algoritmo permite desacoplar el problema a resolver de los mecanismos de comunicaciones de grupo y de la implementación de detectores de fallos. Además de facilitar la resolución del problema de asignación de recursos en particular, el GCS permite resolver otros problemas típicos que se presentan en los sistemas distribuidos tales como la replicación de servicios (File Servers, Storage Servers), la implementación de colas de mensajes y logs distribuidos. Birman en [4] recomienda que "*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 de procesos, nodos y red.

El artículo está organizado de la siguiente manera: en la sección II, se exponen los antecedentes y trabajos relacionados a la asignación de recursos distribuida. En la sección III, se presenta una descripción del algoritmo *SLOTS*, en primer término, de una manera simplificada que no considera fallos, luego el algoritmo con tolerancia a fallos y soporte de cambios de membresía y completando la descripción con el algoritmo definitivo considerando optimizaciones. En la sección IV, se presentan detalles de la implementación de *SLOTS*. En la sección V, se realiza una evaluación de su rendimiento considerando los requerimientos de diseño. Finalmente, las conclusiones acerca de este trabajo y los desafíos a futuro se exponen en la sección VI.

II. ANTECEDENTES Y TRABAJOS RELACIONADOS

Existen numerosos trabajos de investigación relacionados a la asignación de recursos, las comunicaciones de grupos de procesos y técnicas que brindan alta disponibilidad mediante la tolerancia a fallos. Se mencionan aquí solo aquellos trabajos más significativos por cuestiones de espacio.

En lo referente a la asignación distribuida de recursos mencionamos aquí algunas de las estrategias más utilizadas:

1. *Asignación Estática*: Al iniciar la ejecución, cada miembro del grupo tiene asignado un conjunto fijo de recursos. Esta solución trivial no considera el desbalance de carga en los diferentes miembros y tampoco las características dinámicas de los clusters.
2. *Gestor Centralizado de Recursos*: Se dispone de un gestor o coordinador que controla y supervisa la asignación de recursos [5]. Este gestor considera tanto la incorporación como la remoción de miembros al grupo y el desbalanceo de carga. Pero, por la inherente utilización de un componente centralizado, no puede considerarse una solución robusta ni escalable.
3. *Asignación Dinámica Distribuida de Recursos*: El algoritmo distribuido asigna dinámicamente más recursos a aquellos miembros con mayor demanda. La distribución de componentes disminuye la disponibilidad del conjunto, salvo que el algoritmo contemple mecanismos de tolerancia a fallos [6].

El problema del uso de un recurso compartido puede ser visto como equivalente a un clásico problema de exclusión mutua distribuida [7, 8, 9] donde la sección crítica comienza cuando un miembro del grupo adquiere el recurso y finaliza cuando éste lo libera. Existe una diferencia entre ambos y es que los algoritmos de exclusión mutua requieren la especificación de la identidad del recurso a ser asignado, mientras que en la asignación distribuida cualquier instancia de un recurso puede asignarse sin especificar su identidad. Al problema de asignación de múltiples instancia de un recurso se lo conoce como exclusión mutua h-de-k [10, 11, 12, 13, 14, 15] en donde existen múltiples instancias k de un mismo recurso y a cada nodo se le puede asignar hasta un número h de esas instancias ($1 \leq h \leq k$). Algunos de estos algoritmos no prevén la consideración de aspectos que son fundamentales para implementaciones prácticas. Por ejemplo, en [10] el algoritmo opera realizando múltiples intercambios de mensajes entre los miembros del grupo y almacena estados que liberan recursos de otros miembros. No están contemplando fallos de los miembros ni particiones de red, con un simple fallo del proceso solicitante pueden quedar múltiples recursos inhabilitados para ser utilizados.

Una forma de utilizar exclusión mutua distribuida para la asignación de recursos es la replicación de los descriptores de estados (libres u ocupados) de esos recursos. Con esta estrategia, cada vez que se requiere la asignación de un recurso se solicita acceso a la sección crítica que le permite acceder a los descriptores replicados. Cuando accede, se adquiere alguno de los recursos libres, se lo marca como ocupado, para luego salir de la región crítica. Por otro lado, cuando el recurso ya no se utiliza, se debe ingresar nuevamente a la sección crítica,

marcar al descriptor del recurso como libre para luego salir de la región crítica. Cada ingreso o salida de la sección crítica implica una multiplicidad de transferencia mensajes entre los miembros del grupo (dependiendo del algoritmo utilizado). Para el problema de la asignación de *slots*, este tipo de algoritmo fue descartado por la frecuencia de creación y finalización de procesos que tienen los servidores [16], lo que implica una mayor utilización de los recursos de red y un aumento de la latencia durante la creación de un proceso. Algo similar ocurre con el algoritmo de asignación dinámica de canales propuesto en [6] el cuál podría resolver el problema de la asignación de *slots,* pero éste presenta una latencia indeseada. Esto se debe a que requiere la autorización de todos los miembros del grupo antes de asignar el recurso incrementando además el tráfico en la red.

Existe una gran distancia entre la descripción teórica de un algoritmo que generalmente se presenta en una página de pseudo-código y la implementación en un sistema en condiciones de operar que puede tener miles de líneas en lenguaje C o C++ [17]. Los artículos de las referencias mencionadas acerca de otros algoritmos son de carácter puramente teórico y dejan aspectos ingenieriles sin resolver, o eventualmente mencionan a otros algoritmos que podrían ser utilizados para resolverlos. El resultado final termina siendo un algoritmo compuesto y complejo para el cual deja de ser válida la demostración de correcto funcionamiento del algoritmo original. *SLOTS* en cambio, es un algoritmo de una sencillez teórica muy intuitiva construido en dos capas, la superior que gestiona los recursos y la inferior que la brinda un GCS. Este conjunto se encuentra implementado en un prototipo en funcionamiento y en las siguientes secciones de éste artículo se describen las estrategias utilizadas para satisfacer los requerimientos establecidos.

Otro algoritmo en funcionamiento en sistemas en producción que podría resolver el problema de asignación de *slots* es el utilizado por *Wackamole* [1]. Este algoritmo es completamente distribuido y se utiliza para la asignación dinámica de un conjunto de direcciones IP entre los miembros de un grupo de servidores ante la detección de fallos o la incorporación de nuevos miembros. Se asegura consistencia porque ninguna dirección IP será utilizada por más de un servidor a la vez (exclusión mutua) y porque no habrá dirección IP que no sea asignada a un servidor (consistencia). Además, para mejorar el desempeño del cluster, periódicamente se balancean las direcciones IP entre los diferentes servidores. En Wackamole, los eventos que activan acciones en el algoritmo son los relacionados con los cambios en la conformación del grupo de servidores (incorporaciones, remociones, fallos y particiones de red). Si bien estos eventos son comunes en los clusters, no son tan frecuentes si el cluster es estable. Esta fue una de las razones por la cual no se adoptó *Wackamole* para resolver el problema de la asignación de *slots*. Dadas las características dinámicas del problema de asignación de *slots*, el evento que activaría acciones en *Wackamole* sería la falta de recursos suficientes en un miembro. Como efecto de este tipo de eventos se tendría una frecuencia mayor en el rebalanceo de recursos afectando la utilización de la red y la latencia en la asignación del recurso. De todos modos, *SLOTS* y *Wackamole* comparten un aspecto común que es la utilización de Spread Toolkit [18] como GCS. Ambos se basan en los servicios de difusión (multicast) confiable con Orden Total y las notificaciones de membresía de grupo que brinda el GCS [19]. Para la implementación de *SLOTS* se eligió Spread Toolkit por su madurez, adecuado rendimiento y abundante documentación, frente a otros GCS tales como Ring Paxos [20] en el que aún permanecen algunas características sin implementar, o QuickSilver [21], que solo está disponible para plataforma Windows o Corosync [22] que no está completamente documentado.

El Spread Toolkit [19] 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 [23]. Spread soporta el modelo de membresía de Sincronía Virtual Extendida (EVS) [24]. 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. *SLOTS* aprovecha las características para soportar fallos benignos (no bizantinos) y cambios de membresía que brinda EVS, y utiliza un enfoque transaccional para asegurar la consistencia en la asignación de recursos.

III. ALGORITMO DISTRIBUIDO DE ASIGNACIÓN DE RECURSOS – SLOTS

Para el diseño de *SLOTS* se establecieron requerimientos de tal forma de dotarlo de las siguientes propiedades:

1. *Seguro (safety [25, 26]):* Ningún recurso debe asignarse a más de un miembro a la vez (Exclusión Mutua - libre de colisiones) bajo todas las condiciones posibles.
2. *Consistente*: La cantidad de recursos en el sistema se mantiene constante. Cada recurso debe ser asignado a un miembro o estar disponible.
3. *Simétrico*: Todos los miembros siguen los mismos procedimientos para solicitar recursos.
4. *Activo (Liveness [25, 26])*: No debe existir condición lógica por la que el algoritmo detenga su funcionamiento. No deben existir estados donde ocurran *dead-locks* o *live-locks*.
5. *Sin Inanición:* Si existen recursos libres en el sistema, un miembro demandante obtendrá los recursos en un tiempo dado.
6. *Concurrente:* Múltiples miembros podrán solicitar recursos concurrentemente.
7. *Alto Rendimiento:* Ante una mayor demanda debe presentar una alta utilización de recursos y un bajo tiempo de respuesta promedio.
8. *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.
9. *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.

10. *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.
11. *Equitativo*: Se asignarán mayor cantidad de recursos a aquellos miembros con mayores demandas.
12. *Estable*: Después de ser sometido a condiciones de cargas variables transitorias, el algoritmo debe converger a un estado estable.
13. *Elástico*: Si hay recursos suficientes, se debe satisfacer la demanda temporaria.
14. *Adaptativo*: Debe soportar la adición y remoción de miembros al cluster.
15. *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 aún ante particiones de la red y posteriores recomposiciones de la misma.

No hemos incluido entre los requerimientos la propiedad de escalabilidad, no porque no merezca su consideración sino porque está relacionada con las características de eficiencia y rendimiento del propio algoritmo y por estará afectado por las características de escalabilidad del GCS utilizado.

En los siguientes párrafos y secciones se harán referencias a las propiedades requeridas del algoritmo utilizando la siguiente nomenclatura: $\{\pi_1, \pi_4, \pi_7\}$. Este ejemplo indica, que determinada decisión o acción es tomada a favor de satisfacer el requerimiento de las propiedades 1, 4 y 7 antes mencionadas. Para la descripción de la operación del algoritmo se asume que:

A. Se utilizará un GCS como mecanismo para intercambio de mensajes en Orden Total entre los distintos miembros. El GCS refleja el estado actual del cluster y envía notificaciones cuando se producen cambios en membresía o modificaciones del estado de la red.
B. Se ejecuta un miembro del grupo de procesos en cada nodo del cluster (por cuestiones de simplicidad). Como el cluster está compuesto por hasta un máximo de *NR_NODES* nodos, el grupo de procesos estará compuesto de hasta *NR_NODES* miembros.
C. Se dispone de un número finito *NR_SLOTS* de *slots* para asignar entre los diferentes miembros del grupo. Todo *slot* es reutilizable, es decir una vez que se utiliza y se libera puede ser nuevamente utilizado.
D. Cada miembro dispone de una estructura de datos que denominamos *Process Slot Table* (PST) que se replica entre todos los miembros. La PST refleja la propiedad (miembro propietario) de cada uno de los *slots*, por lo tanto cada *slot* tiene un identificador y un identificador de su propietario. Si un *slot* esté asignado a un propietario no significa que el mismo se encuentre en uso. El miembro propietario de un *slot* puede disponer de él utilizándolo y liberándolo, sin alterar su propiedad.

Para una mejor comprensión del algoritmo, se describe *SLOTS* en tres etapas. En primer lugar se presenta una visión simple del mismo donde no son considerados los aspectos relacionados con la tolerancia a fallos ni cambios de membresía. Luego, se presenta un enfoque transaccional donde se presentan los diferentes escenarios de fallos y cambios en la conformación del cluster describiendo las soluciones adoptadas por *SLOTS* a los problemas que estos provocan. Finalmente se proponen algunas optimizaciones referidas a la reducción de mensajes en la red y a la reducción del tiempo de respuesta.

A. Algoritmo Simplificado

El algoritmo propuesto pretende que los *NR_SLOTS slots* disponibles se asignen equitativamente entre los distintos miembros en forma proporcional a la utilización temporaria de *slots* que ellos tuviesen $\{\pi_{11}\}$. Por otro lado, los miembros deberían disponer de cierta cantidad mínima de *slots* libres que le permita satisfacer las demandas inmediatas mientras se solicitan *slots* adicionales $\{\pi_{10}\}$.

El algoritmo *SLOTS* se basa en el concepto sencillo de *donación* y en las propiedades de la difusión (multicast) confiable con Orden Total [27]. El Orden Total en la entrega de mensajes asegura que el sistema no sufrirá de dead-locks ni live-locks $\{\pi_4\}$. Si un miembro requiere disponer de *slots* adicionales, difunde un mensaje de tipo *REQUEST_SLOTS* al resto de los miembros del grupo indicando la cantidad de *slots* requerida $\{\pi_3\}$. Esta cantidad es calculada aplicando una heurística simple que se describe más adelante. El resto de los miembros, al recibir mensajes *REQUEST_SLOTS*, calculan (también mediante una heurística simple) la cantidad de *slots* propios que están en estado libre en condiciones de ser donados. Todos los miembros deben responder difundiendo mensajes *DONATE_SLOTS.* Estos mensajes contienen: 1) el identificador del destinatario de la donación, 2) identificador del donante, 3) la cantidad de *slots* donados y 4) los identificadores de cada uno de los *slots* donados.

Cuando el miembro solicitante recibe los mensajes *DONATE_SLOTS,* por cada uno de ellos cambia el campo de propietario de los *slots* donados en su PST con su propio identificador e incrementa la cantidad de *slots* libres que ahora están en condiciones de ser utilizados. Dado que todos los miembros reciben los mensajes de donación *DONATE_SLOTS* en el mismo orden, todos conocen que los *slots* donados cambiaron de propietario, por lo tanto cada uno de ellos también actualiza su PST. Es decir, la PST es una estructura de datos replicada que se actualiza mediante la difusión con Orden Total. Si bien esto aparenta ser de poca utilidad aquí dado que a cada miembro solo le interesa conocer acerca de cuáles son los *slots* que le pertenecen, esta característica de la PST será utilizada para mantener la consistencia ante fallos $\{\pi_2\}$.

Se denomina *ciclo de donación* al tiempo transcurrido entre la difusión de un mensaje *REQUEST_SLOTS* hasta la recepción de los (*NR_NODES*-1) mensajes *DONATE_SLOTS* en el miembro solicitante. Un miembro solicitante no puede iniciar un nuevo *ciclo de donación* hasta tanto finalice el actual.

Una diferencia con respecto a otros algoritmos de asignación de recursos es que en *SLOTS* los recursos *siempre* se encuentran asignados a un miembro $\{\pi_2\}$ que es su propietario. Esos recursos pueden estar libres o siendo utilizados por el propietario. En caso de que se encuentren

libres, puede donarlos a otros miembros transfiriendo su propiedad. De esta manera el algoritmo permite que la asignación de los *slots* se modifique dinámicamente conforme la carga varía entre los diferentes miembros {π13}.

B. Cambios de Membresía y Tolerante a Fallos.

En primer lugar describiremos como *SLOTS* resuelve los *Cambios de Membresía* en el grupo {π14}. Estos cambios refieren a: 1) la incorporación de nuevos miembros al grupo, 2) la remoción administrativa de un miembro del grupo.

Cuando el cluster inicia, el primer miembro que se conecta al grupo, al no reconocer a otros miembros activos se convierte en el Miembro Primario o *primary_mbr*. El *primary_mbr* adquiere la propiedad de todos los *slots* disponibles en el grupo (*NR_SLOTS)*. Se verá más adelante que *primary_mbr* tiene funciones específicas en el grupo.

Cuando se inician el resto de los miembros, el GCS difunde mensajes *JOIN* por cada uno de ellos. Estos miembros aún no están en condiciones de participar en el intercambio de *slots* dado que no disponen de información de estado. Es responsabilidad del *primary_mbr* transferirle el estado de las variables replicadas por el grupo, entre ellas la PST. Esto se hace mediante la difusión de un mensaje *PUT_STATUS* emitido por el *primary_mbr*. Cuando el nuevo miembro finalizó con la sincronización de las variables replicadas, su estado cambia a *inicializado,* por lo que ahora puede solicitar donaciones de *slots* para utilizar. El nuevo miembro difunde un mensaje *REQUEST_SLOTS* solicitando una cantidad mínima de *slots* preestablecida. El resto de los miembros reaccionan como se explicó previamente.

Cuando un miembro es removido administrativamente del grupo, el GCS difunde un mensaje *LEAVE* con el identificador del miembro finalizado. Si el miembro fuese el *primary_mbr*, entonces entre el resto de los miembros inicializados se elige el nuevo *primary_mbr.* El GCS informa a todos los miembros de la composición del actual grupo, por lo que el nuevo *primary_mbr* es aquel de menor identificador. El nuevo *primary_mbr* adquiere entonces, la propiedad de todos los *slots* que pertenecían al miembro recién finalizado. Todos los miembros actualizan la propiedad de dichos *slots* y la nueva conformación del grupo. Si la terminación del *primary_mbr* hubiese ocurrido posteriormente a la recepción de un mensaje *JOIN* de otro miembro, pero previamente a que el *primary_mbr* hubiese respondido a ella con un mensaje *PUT_STATUS*, el nuevo *primary_mbr* debe difundir el mensaje *PUT_STATUS* al nuevo miembro (Fig. 1). Por esta razón todos los miembros inicializados registran los mensajes *JOIN* de nuevos miembros y los mensajes *PUT_STATUS* del actual *primary_mbr.* Este registro les permite conocer cuáles son aquellos miembros nuevos que están pendientes de recibir mensajes *PUT_STATUS*, para el caso eventual de que alguno de ellos pueda transformarse en el próximo *primary_mbr* por la finalización del actual.

Si el miembro removido no es el *primary_mbr*, entonces la propiedad de todos sus *slots* se transfiere al *primary_mbr.* Todos los miembros actualizan la propiedad de dichos *slots* y la nueva conformación del grupo.

Una característica del GCS utilizado en la implementación (Spread Toolkit) es que informa cambios de topología del grupo mediante lo que se denominan cambios de *vista* (Fig. 1). El GCS difunde diferentes mensajes según sea una desconexión de la aplicación miembro del grupo (fallo de proceso) o la desconexión del nodo donde ella ejecuta. Para el primer caso el GCS difunde un mensaje *DISCONNECT*, en tanto la desconexión del nodo se informa difundiendo un mensaje *NET_PARTITION (*cambio de topología de la red). Esto se debe a que el GCS no puede distinguir entre un nodo que finalizó abruptamente de aquel que dejó de estar conectado a la red.

Para resolver el problema de tolerancia a fallos {π15} se aplicó un enfoque transaccional similar al utilizado por los motores de Bases de Datos. Se define aquí a una transacción como un conjunto indivisible de operaciones. Si este conjunto finaliza correctamente se dice que se hizo un COMMIT de la transacción. Pero, si alguna de dichas operaciones no finaliza correctamente y/o es interrumpida por algún evento tal como un fallo, se invalida la transacción y el estado del sistema debe volver (ROLLBACK) al estado estable y consistente previo a iniciar la transacción {π2}. Para poder retornar a un estado estable, los motores de Bases de Datos registran las operaciones que componen las transacciones en almacenamiento no volátil. En SLOTS se descartó esa técnica por el costo en latencia que implicarían dichos registros. Por otro lado, el conjunto de transacciones que se ejecutan en SLOTS son preestablecidas, por lo que se toman los recaudos necesarios para completarlas o para realizar ROLLBACK sin necesidad de utilizar almacenamiento no volátil. Durante las transacciones que dan tratamiento a los fallos y cambios de membresía se inhibe en todos los miembros la posibilidad de solicitar o donar slots hasta tanto la transacción finalice o se aborte.

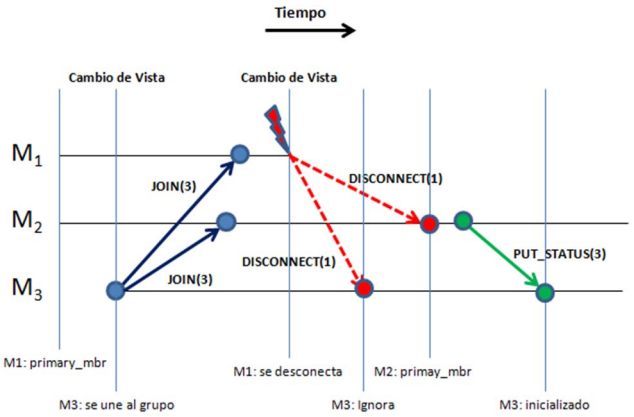

Figura 1. El nuevo *primary_mbr* envía *PUT_STATUS* al nuevo miembro.

A continuación se describen los siguientes escenarios de fallos (benignos): 1) la detención abrupta de un miembro del grupo (fallo de proceso), 2) la partición de la red 3) la reunificación de la red. Para cada uno de ellos se detallan las acciones ejecutadas por el algoritmo para tolerar esos fallos a fin de lograr una mayor disponibilidad del servicio.

Cuando se produce una detención abrupta de un miembro (fallo de proceso) el GCS difunde un mensaje *DISCONNECT.* Las acciones que toma *SLOTS* son idénticas a las descriptas para el caso de la detención administrativa de un miembro

(mensaje *LEAVE)*.

Cuando se produce una partición de red el GCS difunde un mensaje *NET_PARTITION*. Estos tipos de fallos generan un problema conocido como "*cerebro partido*" (Split-brain) [28] porque, eventualmente, pueden permanecer operativos dos o más subconjuntos de nodos o *particiones*. Cada una de esas particiones ignora el estado en el que se encuentran el resto de las particiones. Existen dos enfoques para resolver el problema: A) *Optimista*: cada partición asume que las otras particiones no están activas y por lo tanto se pueden reutilizar los recursos que ellas tenían asignados, B) *Pesimista*: cada partición asume que las otras particiones permanecen activas y se limita la utilización de los recursos a aquellos asignados a los miembros de la propia partición.

Para el caso A), si luego la red se recompone, podría haber inconsistencia en la asignación de recursos dado que las particiones podrían haber utilizado recursos asignados previamente a otras particiones. En cambio, si los nodos de las otras particiones se hubiesen efectivamente desconectado, la partición activa hubiese utilizado los recursos en forma más eficiente. Para el caso B), si la red se recompone, no habría inconsistencia en la asignación de recursos. En cambio si los nodos de las otras particiones se hubiesen efectivamente desconectado, la partición activa hubiese subutilizado los recursos. Por las características de la aplicación donde se utiliza actualmente *SLOTS,* se decidió que la consistencia de asignación de recursos $\{\pi_2\}$ es más importante que la disponibilidad de un mayor número de recursos y por lo tanto se adoptó la opción B). Esta estrategia evita inconsistencia $\{\pi_1, \pi_2\}$ por fallos de particiones de red.

Si eventualmente uno de los miembros que era considerado integrante de otra partición hace un *JOIN* para incorporarse al grupo, esto estaría indicando al grupo que el nodo de ese miembro en realidad se desconectó y luego se reinició. Ante esta situación, todos los miembros del grupo cambian la asignación de los *slots* que pertenecían al nodo reiniciado al *primary_mbr*. Es decir, el nodo reiniciado no mantiene la propiedad de los *slots* que tenía asignados al momento de la desconexión $\{\pi_{14}\}$ como tampoco conserva información alguna de su estado a ese momento, solo mantiene su identificación (*node_id*).

Cuando se produce una partición de red, los miembros de cada partición deben elegir su propio *primary_mbr* para continuar la ejecución y brindar el servicio $\{\pi_{15}\}$. Es decir, se ejecuta un algoritmo *SLOTS* independiente en cada una de las particiones $\{\pi_4\}$ operando sobre el sub-conjunto de *slots* cuyos propietarios conforman esas particiones (Fig. 2).

Cuando se recompone el estado de la red, el GCS difunde un mensaje *NET_MERGE* con la composición del nuevo grupo. Los *primary_mbr* de cada una de las particiones difunden los estados solo de los *slots* asignados a miembros de su partición con mensajes de tipo *MERGE_STATUS.* Esto les permite a todos los miembros construir una nueva PST resultante de la fusión de las PST de las particiones constituyentes. Posteriormente, se elige entre todos los miembros el nuevo *primary_mbr* del grupo para luego continuar con la ejecución normal $\{\pi_4\}$ con el cluster completo.

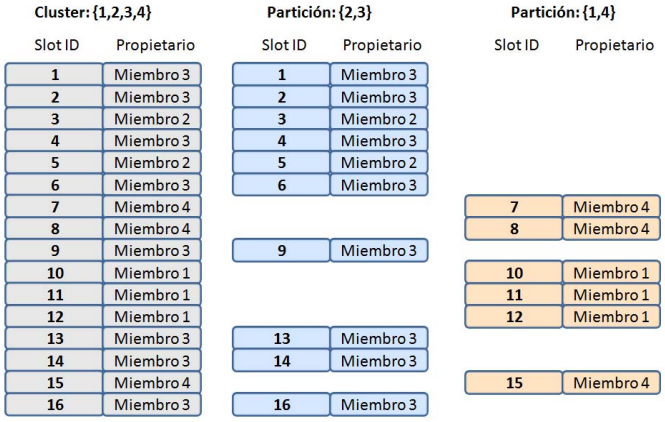

Figura 2. Asignación de recursos antes y después de una partición de red.

De los escenarios de fallos, cambios de membresía y protocolo de donación de *slots* presentados se pueden distinguir aquellas secuencias de operaciones que deben ser consideradas como transacciones:

1) *Transacción JOIN-PUT_STATUS*

Si el GCS difundió un mensaje *JOIN*, e inmediatamente después difunde otro mensaje de cambio de vista tal como *NET_PARTITION, DISCONNECT, NET_MERGE, JOIN*, se interrumpe la transacción por la cual el *primary_mbr* debió difundir el mensaje *PUT_STATUS* destinado al nuevo miembro. Finalizado el tratamiento del segundo (o último) evento, el *primary_mbr* debe emitir el *PUT_STATUS.* Este *primary_mbr* no es necesariamente el mismo que el *primary_mbr* que daría tratamiento al primer *JOIN*, dado que el segundo cambio de vista pudo haber sido provocado por la desconexión del *primary_mbr* original. Por ésta razón, todos los miembros se encuentran sincronizados y en condiciones de asumir el rol de *primary_mbr* para, eventualmente finalizar con aquellas transacciones que hubiesen quedado pendientes por ser interrumpidas. Con esta estrategia se satisface la propiedad de la continuidad de actividad $\{\pi_4\}$.

2) *Transacción REQUEST_SLOTS-DONATE_SLOTS*

Cuando un miembro solicita donación de *slots* difundiendo un mensaje *REQUEST_SLOTS* comienza a registrar todos aquellos miembros donantes que le responden con mensajes *DONATE_SLOTS*. Como condición se estableció que un miembro solicitante no puede emitir una nueva solicitud hasta tanto obtenga la respuesta de todos los otros miembros. Si se produce un cambio de vista antes de que todos los miembros respondan, el miembro solicitante elimina de la lista de miembros pendiente de respuesta a aquellos que quedaron en otra partición o fueron desconectados $\{\pi_4\}$. En tanto que si el miembro solicitante fue el desconectado, todos los miembros donantes cuyos mensajes *DONATE_SLOTS* no fueron entregados antes de la desconexión hacen un *ROLLBACK* de la propiedad de los *slots* donados al miembro desconectado. Un ejemplo de esto se presenta en Fig. 3 donde el miembro M1 solicita *slots* difundiendo un mensaje *REQUEST_SLOTS.* El miembro M3 recibe este mensaje y difunde un mensaje de

donación *DONATE_SLOTS* el cual es entregado a todos los miembros del grupo. Todos los miembros cambian la propiedad de los *slots* donados de M3 a M1. Cuando M2 recibe el mensaje *REQUEST_SLOTS* de M1, difunde un mensaje de donación *DONATE_SLOTS* el cual no se entrega a todos los miembros como consecuencia de que se produce un cambio de vista por desconexión de M1 (*DISCONNECT*).

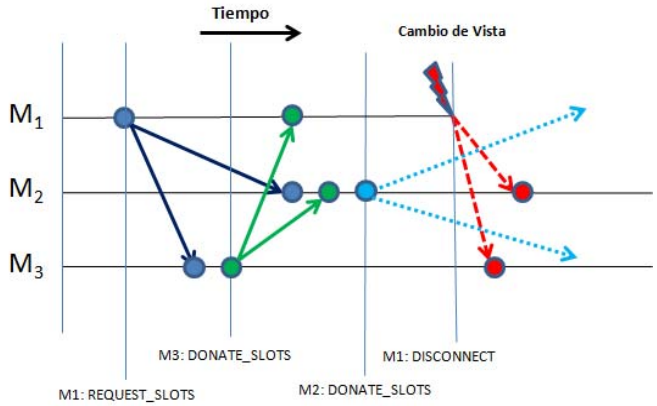

Figura 3. *ROLLBACK* de *slots* donados.

La propiedad de los *slots* donados por M3 fue transferida a M1 antes de la desconexión, por lo que la propiedad de esos *slots* se transferirá al *primary_mbr* después de la desconexión de M1. En el caso de los *slots* donados por M2, como no se entregó el mensaje *DONATE_SLOTS* antes de la desconexión de M1, el miembro M2 recupera la propiedad de los mismos (*ROLLBACK*). De esta forma se asegura la continuidad de actividad $\{\pi_4\}$ y la consistencia de la PST $\{\pi_2\}$.

3) *Transacción NET_MERGE-MERGE_STATUS*

Cuando el GCS reporta una reunificación total o parcial de la red mediante un mensaje *NET_MERGE*, dos o más particiones deben fusionarse. Cada una de esas particiones tiene su propio *primary_mbr* y su propio sub-conjunto de *slots* asignados. Cada *primary_mbr* difunde un mensaje *MERGE_STATUS* informando el estado de sus variables replicadas y su PST. La transacción se considera desde el momento en que se recibe el mensaje *NET_MERGE* hasta que se reciben todos los mensajes *MERGE_STATUS* de los *primary_mbr* de todas las particiones que se reunifican. Si entre ellos se produce un nuevo cambio de vista, todos los miembros hacen un *ROLLBACK* y vuelven al estado previo al *NET_MERGE*, esto asegura la consistencia de la PST $\{\pi_4\}$.

C. *Optimizaciones*

Durante las etapas de diseño, implementación y evaluación de *SLOTS* se detectaron al menos tres posibles optimizaciones al algoritmo básico presentado.

1. Cuando un miembro se une al grupo, el GCS lo informa con *JOIN*. El *primary_mbr* difunde un *PUT_STATUS* para sincronizarlo. Como ese mensaje llega a todos los miembros, el nuevo miembro no necesita difundir un *REQUEST_SLOTS* para solicitar *slots*, esto ya está implícito al considerarse un nuevo miembro inicializado, por lo que se evita la difusión de ese mensaje.
2. Si concurrentemente a la solicitud de donación de un miembro $\{\pi_6\}$ éste recibe un mensaje *REQUEST_SLOTS* de otro miembro, ninguno de los dos miembros responde a la petición dado que si ambos están solicitando *slots* se asume que no están en condiciones de donar.
3. Durante períodos de alta demanda, la probabilidad de que muchos miembros realicen donaciones nulas aumenta. Como todos los miembros deben responder a la solicitud de donación, podrían llegarle al solicitante mensajes de donaciones nulas antes que donaciones efectivas de *slots* aumentando el tiempo de respuesta. Para mejorar este comportamiento, los miembros que no tienen *slots* para donar, retrasan sus respuestas hasta recibir el primer mensaje de donación emitido por otro donante. Para evitar el bloqueo $\{\pi_4\}$ del caso extremo en que ningún miembro dispone de *slots* para donar, el miembro siguiente en orden rotativo al miembro solicitante no retrasa su respuesta, aún en el caso de que su donación sea nula asegurando la continuidad de actividad.

IV. IMPLEMENTACION DE *SLOTS*

El algoritmo *SLOTS* ha sido implementado en lenguaje C sobre Linux 2.6.32 x86 de 32 bits. El tipo de cluster objetivo consiste básicamente en computadores conectados por una red de area local (LAN), confiable y de alta velocidad. El algoritmo en sí no impone límite respecto al tamaño del cluster por lo que su escalabilidad está limitada entre otros aspectos, por el rendimiento de transferencia de mensajes del GCS subyacente y de la infraestructura que constituye el cluster.

Cada ciclo de donación requiere *NR_NODES* difusiones (sin considerar optimizaciones). Si simbolizamos con Δ_{mcast} el tiempo medio de difusión de un mensaje desde que éste es emitido hasta que es entregado a todos los miembros, entonces límite máximo del rendimiento del algoritmo se puede estimar como:

$$(1)\ N_{dc} = (NR\_NODES * \Delta_{mcast})^{-1}$$

Siendo N_{dc} es la cantidad de ciclos de donaciones por segundo.

A. *Arquitectura Implementada*

En la implementación actual, cada proceso miembro está compuesto de dos hilos de procesamiento (*threads*). Uno de ellos administra la utilización de los *slot* que le pertenecen al miembro y se lo denomina *Process Management Thread* (*PMT*). El otro hilo gestiona la asignación de *slots* entre los diferentes miembros del grupo y se lo denomina *Slots Management Thread* (*SMT*) (Fig. 4). Entre ambos hilos existen estructuras de datos compartidas por lo cual el acceso a ellas se protege con *mutexes* para evitar condiciones de competencia. En el *SMT* es donde principalmente se implementa *SLOTS,* en tanto que el *PMT* es el componente que utiliza los *slots* asignados al miembro y que solicita la donación cuando se encuentra escaso de recursos.

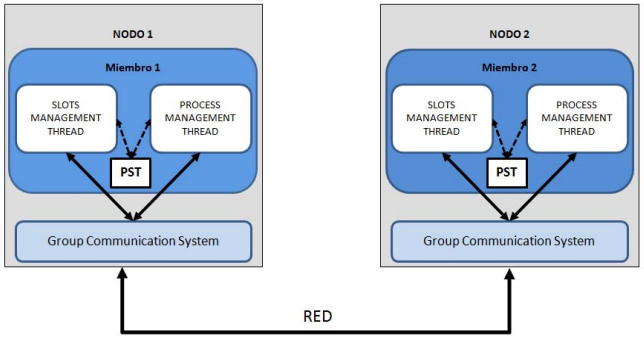

Figura 4. Modelo de arquitectura de implementación de *SLOTS*.

B. Heurística Utilizada

Para completar la descripción del funcionamiento del algoritmo se requiere establecer criterios por los cuales un miembro solicita donación de *slots* y otros, por los cuales se encuentra en condiciones de donar *slots*.

La heurística utilizada es dependiente del problema concreto para el cual se diseñó *SLOTS*. Para este caso aplican criterios heurísticos muy simples, que probablemente podrían replantearse cuando se utilice en entornos de producción fuera del laboratorio. Particularmente, cuando un miembro solicita *N slots*, la donación total puede ser mayor, menor o igual a la cantidad *N* solicitada $\{\pi_8\}$. Este resultado podría no ser aplicable a otros tipos de problemas en donde se requiere satisfacer exactamente una cantidad de recursos determinada.

Como criterios para solicitar *slots* se establecieron:

1. El número de *slots* libres asignados al miembro (*s_free*) debe ser inferior a un límite especificado (*FREE_LOW*).
2. El miembro no debe tener peticiones de donación pendientes de respuesta (no se puede iniciar otro ciclo de donación).
3. La cantidad de *slots* a solicitar (Fig. 5) dependerá del promedio de asignación de *slots* que le corresponden a cada miembro (*max_owned*) de los *init_nodes* miembros inicializados. Si el miembro es propietario de *s_owned slots* y aún no ha alcanzado *max_owned slots*, entonces solicita una donación con la cantidad necesaria para alcanzar ese valor. Si por el contrario, el miembro tiene asignado mayor cantidad de *slots* que *max_owned,* solicita una donación de *slots* que le permita alcanzar el número mínimo de *slots* libres establecido (*FREE_LOW*).

Para la donación de *slots* en respuesta a la solicitud de otro miembro se establecieron los siguientes criterios:

1. Si el miembro donante también es un solicitante con peticiones pendientes, su donación será de cero *slots*.
2. El número de *slots* propios del miembro en estado libre (*s_free*) debe ser superior a un límite especificado (*FREE_LOW*).
3. La cantidad de *slots* a donar (Fig. 6) dependerá de la cantidad requerida (*s_req*) y del excedente de *slots* libres (*s_free*) en el miembro. Si no tiene excedentes no donará *slots*. De lo contrario donará una cantidad de *slots* inversamente proporcional al número de miembros donantes (*donors*). Si dispone de excedente suficiente, donará todo su excedente.

Con la heurística utilizada se logra la equidad $\{\pi_{11}, \pi_{12}\}$ en la asignación de los *slots* después de unos pocos ciclos de donación. Esto se puede comprobar con los resultados que se presentan en la sección Evaluación.

```
int slots_to_request(void ){
  max_owned = CEILING(NR_SLOTS, init_nodes);
  if(max_owned > s_owned)
     return(max_owned - s_owned);
  return(FREE_LOW - s_free);
}
```

Figura 5. Pseudo-código de la heurística aplicada a la petición de *slots*.

```
int slots_to_donate(int s_req){
   donors = init_nodes - 1;
   surplus = (s_free – FREE_LOW);
   if( surplus <= 0) return(0);
   if(surplus > CEILING(s_req, donors))
        return(CEILING(s_req, donors));
   return(surplus);
}
```

Figura 6. Pseudo-código de la heurística aplicada a la donación de slots.

V. EVALUACION DE RENDIMIENTO

Como metodología para la evaluación de rendimiento se utilizó la simulación del algoritmo, siguiendo un enfoque similar a [29] 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 de *SLOTS* para el simulador DAJ [30], 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 y se lo puede solicitar por e-mail a cualquiera de los autores.

En esta sección los recursos estarán representados por *slots* 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 *slot* libre en miembro para el nuevo descriptor de proceso.

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

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

En [31] 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 en la simulación. En tanto que para μ se adoptó una distribución normal inversa [16].

El GCS se modeló como un nodo servidor de difusión (*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 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 *slots* libres para utilizar realiza una difusión (a través del nodo *SPREAD*) de un mensaje tipo *MSG_FORK* indicando la cantidad requerida de *slots* (*FREE_LOW*). Cuando un miembro (incluido el emisor) recibe este mensaje, busca los primeros *FREE_LOW* libres de la PST 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 los mismos *slots* al emisor. Cuando un miembro dispone de una cantidad de *slots* mayor o igual a un valor establecido *FREE_HIGH* libera *slots* hasta quedarse solo con *LOW_SLOTS.* Para ello difunde un mensaje tipo *MSG_EXIT*, especificando los identificadores de los *slots* liberados.

Los indicadores de rendimiento evaluados son:

A. *Utilización*: Es la suma de *slots* utilizados por todos los miembros respecto al total (*NR_SLOTS*).
B. *Tasa de Peticiones Fallidas/Exitosas*: Es la tasa entre las peticiones para utilizar un recurso cuando no hay recursos 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 Multicast* (Δ_{mcast}). En una implementación realista, el *Tiempo de Multicast* dependerá del GCS utilizado y de la infraestructura del cluster. El Tiempo de Respuesta considera el número de multicast de mensajes que se requieren desde que un miembro emite una solicitud de *slots* hasta que recibe un mensaje de donación de al menos un *slot*.

Por las características de DAJ, los modelos utilizados en las simulaciones realizan rondas o *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 un *round* 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 de recurso, se utilizó como unidad de medición del tiempo al *round* de simulador.

En la Fig.7 se presenta la utilización de *slots* para la carga de trabajo impuesta al cluster de una de las simulaciones con los siguientes parámetros en cada nodo *i*:

- $NR\_SLOTS = 768$
- $NR\_NODES = 32$
- $FREE\_SLOTS = 4$
- $0{,}5 \leq \lambda_i \leq 0{,}8$
- $1 \leq \mu_i \leq 100$

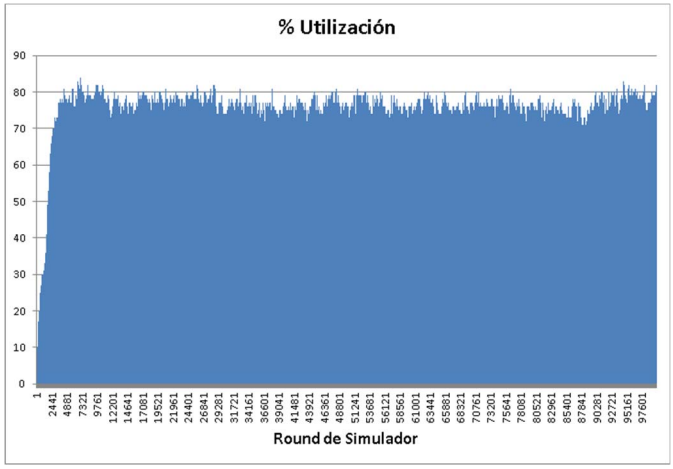

Figura 7. Utilización de recursos.

En la simulación se introdujeron variaciones (cada 10000 rounds de simulador) del λ_i asignado a cada miembro. De todos modos, la gráfica revela la estabilidad del algoritmo $\{\pi_{12}\}$. Una vez alcanzado un estado de régimen donde los *slots* fueron equitativamente distribuidos entre los diferentes miembros, la utilización media es del 73%. El algoritmo de exclusión mutua tuvo una utilización 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 multicasts (Tabla I).

TABLA I
DESEMPEÑO DE *SLOTS*

ALGORITMO	*fork()*Exitosos/ Multicasts	*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 multicast requerido por el protocolo. En la misma simulación este indicador arrojó un valor de 11,45 frente a 6,28 del algoritmo de exclusión mutua.

En la Fig. 8 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 *slots* libres para asignar. En esa figura también evidencia la estabilidad del algoritmo $\{\pi_{12}\}$.

En forma resumida, la Tabla I 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%.

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 *slots*, no se bloquea al 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 *slot* libre o, finalice en el miembro un proceso liberando un *slot*. Si se optara por una implementación bloqueante, entonces no habría peticiones de *fork()* fallidas sino peticiones demoradas que bloquearían al solicitante.

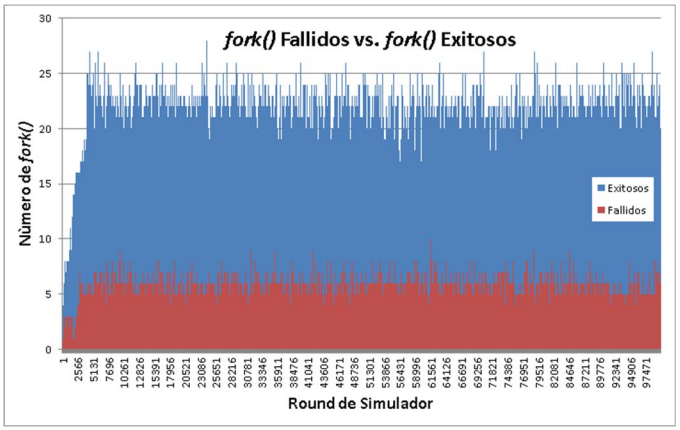

Figura 8. *fork()* Fallidos vs. *fork()*Exitosos.

En la Tabla I, se tomó como parámetro para ambos algoritmos la cantidad mínima de *slots* libres (*FREE_LOW*) que debe tener un miembro. Este parámetro se utiliza para decidir cuándo un miembro debe solicitar la donación de *slots*, y para decidir si un miembro está en condiciones de donar *slots*. 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. 9 se puede observar que los valores son mayoritariamente dos (79%) o tres (16%) tiempos de multicast (Δ_{mcast}). Esto significa que un miembro solicitante probablemente recibirá una donación de al menos un *slot* en el primer o segundo mensaje de donación $\{\pi_7\}$.

Los resultados de la simulación demuestran que *SLOTS* cumple con los requerimientos de rendimiento, eficiencia, estabilidad y elasticidad establecidos $\{\pi_5, \pi_7, \pi_9, \pi_{12}, \pi_{13}\}$.

VI.CONCLUSIONES Y TRABAJOS A FUTURO

Se ha presentado un algoritmo distribuido tolerante a fallos para la asignación de múltiples instancias de un mismo recurso. Sus características lo hacen apto para la utilización en ambientes de Clusters, Grid y Cloud Computing.

Tanto en el diseño como implementación del algoritmo se utilizó un modelo de capas en donde *SLOTS* basa su lógica en los servicios que le brinda un GCS. El GCS subyacente es responsable de notificar a *SLOTS* sobre los cambios de membresía y fallos en los miembros del cluster o en la red. Todas estas situaciones son tratadas por *SLOTS* para alcanzar su objetivo principal que es la asignación distribuida de recursos tolerante a fallos. Para tratar con los fallos (benignos) se adoptó un enfoque de tipo transaccional que permite garantizar un estado final consistente posterior al fallo y a su recuperación. Su diseño por capas y su sencillez facilitan su comprensión e implementación reduciendo así la probabilidad de cometer errores de lógica y/o de programación.

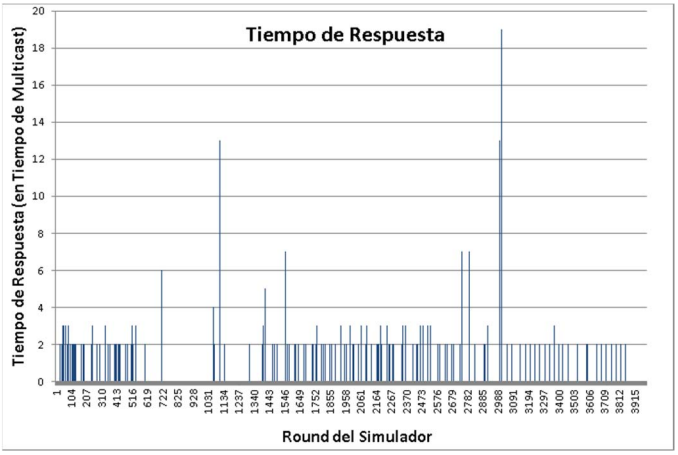

Figura 9. *Tiempo de Respuesta*.

Claramente, el algoritmo podría implementarse sin utilizar un GCS, pero esto implicaría que se deberían resolver aspectos tales como la difusión con Orden Total, los cambios de membresía y la implementación de detectores de fallos que son requeridos para realizar una implementación práctica y realista del mismo.

En cuanto al rendimiento, *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 de un prototipo de un Sistema de Virtualización Distribuida en un cluster de 8 nodos de laboratorio. El código de *SLOTS* en lenguaje C para Linux se encuentra disponible y se lo puede solicitar por e-mail a cualquiera de los autores

AGRADECIMENTOS

La participación de Toni Cortes en este trabajo ha sido financiada por el Gobierno de España (subvención SEV2015-0493 del programa Severo Ochoa) por el Ministerio Español de Ciencia e Innovación (contrato TIN2015-65316) y la Generalitat de Catalunya (contrato 2014-SGR-1051).

La participación de Fernando G. Tinetti en este trabajo ha sido financiada por la UNLP (Facultad de Informática) y la CIC

Provincia de Buenos Aires, Argentina.

REFERENCIAS

[1] Yair Amir, Ryan Caudy, Ashima Munjal,Theo Schlossnagle, Ciprian Tutu; "N-way fail-over infrastructure for survivable servers and routers", *IEEE International Conference on Dependable Systems and Networks*, DSN03, 2003.

[2] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson, "Impossibility of distributed consensus with one faulty process", *J. ACM*, 32(2):374–382, 1985.

[3] Pablo Pessolani, Toni Cortes, Silvio Gonnet, Fernando G. Tinetti, "Sistema de Virtualización con Recursos Distribuidos", in *WICC 2012,* Argentina, 2012.

[4] Kenneth P. Birman, Robert Cooper, Barry Gleeson, "Design Alternatives for Process Group Membership and Multicast", Department of Computer Science, Cornell University, 1991.

[5] M. Zhang and T.-S. P. Yum, "Comparisons of Channel-Assignment Strategies in Cellular Mobile Telephone Systems"*, IEEE Transactions on Vehicular Technology*, November 1989.

[6] Ravi Prakash and Niranjan G. Shivaratri and Mukesh Singhal, "Distributed Dynamic Fault-Tolerant Channel Allocation for Mobile Computing", IEEE Transactions On Vehicular Technology, 1999.

[7] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks", *Commun. ACM*,vol. 24, pp. 9–17, January 1981.

[8] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm"*, ACM Trans. Comput. Syst.*, vol. 3, no. 4, pp. 344–349, 1985.

[9] K. Raymond, "A tree-based algorithm for distributed mutual exclusion", *ACM Trans. Comput. Syst.*, vol. 7, no. 1, pp. 61– 77, 1989.

[10] Raynal, M., "A Distributed Solution for the k-out of-m Resources Allocation Problem", in *Lecture Notes in Computer Sciences*, Vol. 497. Springer Verlag. 1991.

[11] Baldoni, R., Manabe, Y., Raynal M., Aoyagy, S., "k-Arbiter: A Safe and General Scheme for h-out of-k Mutual Exclusion", Theoretical Computer Science, 1998.

[12] Manabe, Y., Tajima, N., "(h-k)-Arbiter for h-out of-k Mutual Exclusion Problem", in *Theoretical Computer Science*, 2004.

[13] Jiang, Jehn-Ruey, "A Fault-tolerant H-out Of-k Mutual Exclusion Algorithm Using Cohorts Coteries for Distributed Systems", in *Proceedings of the 5th International Conference on Parallel and Distributed Computing: Applications and Technologies*, 2004.

[14] Chaudhuri, Pranay and Edward, Thomas, "An Algorithm for K-mutual Exclusion in Decentralized Systems", in *Computer Communications*. September, 2008.

[15] J. Lejeune, L. Arantes, J. Sopena, and P. Sens., "Reducing synchronization cost in distributed multi-resource allocation problem", in *44th International Conference on Parallel Processing*, 2015.

[16] Mor Harchol-balter and Allen B. Downey. *Exploiting process lifetime distributions for dynamic load balancing*, ACM Transactions on Computer Systems, 1997.

[17] T. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective", in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC '07)*, 2007.

[18] The Spread Toolkit. http://www.spread.org.

[19] Yair Amir, Claudiu Danilov, Michal M. Amir, John Schultz, and Jonathan 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.

[20] P. Marandi, M. Primi, N. Schiper, F. Pedone, "Ring Paxos: A high-throughput atomic broadcast protocol", in *DSN*, 2010.

[21] 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.

[22] S. Dake, C. Caulfield, A. Beekhof, "The Corosync Cluster Engine", in *Linux Symposium*, 2008.

[23] Y. Amir, C. Danilov, and 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.

[24] Moser, L. E., Amir, Y., Melliar-Smith, P. M., and Agarwal, D. A. "Extended Virtual Synchrony", In *Proceedings of the IEEE 14th International Conference on Distributed Computing Systems* (Poznan, Poland, June). IEEE Computer Society Press, 1994.

[25] V. Diekert and P. Gastin, "Local safety and local liveness for distributed systems", In *Perspectives in Concurrency Theory*, IARCS-Universities, pages 86-106. Universities Press, 2009.

[26] Bowen Alpern and Bowen Alpera and Fred B. Schneider and Fred B. Schneider, "Recognizing Safety and Liveness", *Distributed Computing*, 1986.

[27] Xavier Défago and André Schiper and Péter Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey", in *ACM Computing Surveys*, 2004.

[28] John Burke, Rasmus Rygaard, Suzanne Stathos, "Split-Brain Consensus. On A Raft Up Split Creek Without A Paddle", Stanford Secure Computer Systems Group.

[29] Juan Fan,Wuyang Zhou, "A Distributed Resource Allocation Algorithm in Multiservice Heterogeneous Wireless Networks", in *Wireless Internet - 7th International {ICST} Conference, WICON 2013,* China, 2013.

[30] Wolfgang 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.

[31] Zhen Liu, Nicolas Niclausse, and César Jalpa-Villanueva, "Traffic model and performance evaluation of Web servers", in *Performance Evaluation*, 46, 2-3 October 2001.

Pablo Pessolani is a Ph.D. candidate in Informatics at the Universidad de La Plata, La Plata, Argentina. In 1986 he received the degree in Electrical Engineering from the Universidad Tecnológica Nacional (UTN), Santa Fe, Argentina. In 2006 he received the MSc. degree in Computer Networks from Universidad de La Plata, Argentina. He is a professor in UTN Santa Fe since 1992. His main research and professional interests involve Operating Systems, Real Time Systems, Distributed Systems, Computer & Network Security and Virtualization.

Oscar Jara got his Information Systems Engineering degree at Universidad Tecnológica Nacional, Santa Fe, Argentina, in year 2006. He has been working in that same institution as Assistant Professor in the Advanced Operating Systems course for the last five years. He is currently involved with the industry working as cloud architect and software engineer for several firms.

Silvio Gonnet received an Engineering degree in Information Systems from Universidad Tecnológica Nacional (UTN), Santa Fe, Argentina, in 1998, and obtained his PhD. degree in Engineering from Universidad Nacional del Litoral (UNL) in 2003. He currently holds a Researcher position at the National Council for Scientific and Technical Research of Argentina (CONICET), to work at Instituto de Desarrollo y Diseño (INGAR). Also, he works as an Assistant Professor at Universidad Tecnológica Nacional. His research interests are about models to support the design process, software architectures, and semantic web.

Toni Cortes is the manager of the storage-system group at the BSC and associate professor at UPC. He received his PhD in computer science in in 1997 from UPC. Since 1992, Toni as been teaching operating system and computer architecture courses at the Barcelona school of informatics (UPC) where he also served as vice-dean for international affairs. His research concentrates in storage systems, programming models and operating systems. He has published more than 125 technical papers. In addition, he has also advised 10 PhD thesis. Dr. Cortes has been involved in several EU and industry projects).

Fernando G. Tinetti received a Licentiate degree in Computer Science from Universidad Nacional de La Plata, La Plata, Argentina, in 1992. In 1997 he received a MSc. degree, and in 2004 received a PhD. degree in Computer Science from Universidad Autónoma de Barcelona, Barcelona, España. He

is a profesor in UNLP since 1990. Since 2005 holds a Researcher position at Comisión de Investigaciones Científicas de la Provincia de Buenos Aires. His main research and professional interests involve HPC, Parallel computing, Distributed computing, Real Time Systems and Robotics.