Gestión de procesos distribuidos

Miguel Angel Astor Romero

Versión 2 - 28 de septiembre de 2016

Introducción

Un sistema distribuido se define como "*una colección de computadoras independientes que dan al usuario la impresión de constituir un único sistema coherente*"[1]. Esta independencia física de las computadoras que componen al sistema (comúnmente conocidas como nodos) implica un incremento considerable en la complejidad del mismo en todas las áreas referentes a la gestión de los recursos del sistema, estando entre estas la gestión de los procesos. En un sistema distribuido el proceso sigue siendo la unidad mínima de ejecución y asignación de recursos. Sin embargo, en los sistemas distribuidos los procesos deben ser diseñados y desarrollados para explotar la distribución física y las capacidades de comunicación del sistema, lo que presenta nuevas posibilidades y complejidades con respecto a la gestión de los mismos.

El resto de este documento se estructura de la siguiente manera: en la Sección 1 se realiza un estudio detallado de un mecanismo que permite a un proceso ubicado en un nodo específico del sistema el poder detener su ejecución para reanudarla en un nodo diferente, llamado migración de procesos. En la Sección 2 se describe un mecanismo de comunicación entre procesos conocido como Llamadas a Procedimientos Remotos. En la Sección 3 se muestran distintas estrategias que se pueden tomar para la detección y resolución de interbloqueos en sistemas distribuidos.

1. Migración de procesos

Se entiende por migración de procesos a la capacidad de un sistema distribuido de trasladar un proceso activo de un procesador a otro de forma transparente. Esta capacidad corresponde a un subconjunto de las funcionalidades definidas por el proceso de migración de código.

1.1. Motivaciones

Las motivaciones de la migración de código se dividen en dos grupos fundamentales. El primero de estos se refiere a mejorar el desempeño del sistema al preferir migrar código antes que migrar datos. El segundo grupo se refiere a aumentar la flexibilidad del sistema.

1.1.1. Rendimiento

El enfoque de migración de código para mejorar el rendimiento global consiste en minimizar las transferencias de datos prefiriendo que los códigos que operan sobre estos se ejecuten de manera local a los mismos. Mediante este enfoque se optimiza el rendimiento al minimizar el tamaño de las comunicaciones entre nodos, siendo, en teoría, más económico transferir procesos y sus resultados que bloques de datos [2].

Otro enfoque de optimización del rendimiento consiste en migrar los procesos de nodos sobrecargados a nodos con menor carga de trabajo, de forma que se balancee la carga global del sistema [1].

1.1.2. Flexibilidad

Para mejorar la flexibilidad con migración de código se utiliza un mecanismo de configuración dinámica de clientes dentro de un esquema cliente-servidor. De esta manera, la implementación de ciertos detalles del funcionamiento de los clientes se mantiene en un repositorio de código ejecutable externo. En este repositorio se incluyen ciertos mecanismos que el cliente necesita para comunicarse exitosamente con el servidor [1].

1.2. Consideraciones

Para poder realizar la migración de código es necesario tomar en cuenta dos aspectos: el mecanismo de movilidad y la política de migración. Con mecanismo de movilidad nos referimos a las distintas formas en que una migración de código puede desarrollarse. La política de migración por su parte define cuando se puede migrar un proceso, que procesos migrar y como determinar hacia donde migrar dichos procesos.

1.2.1. Mecanismos de movilidad

Para poder establecer los mecanismos de movilidad, primero hay que definir que parte de los procesos se puede migrar. Para esto dividiremos un proceso en tres segmentos: el segmento de código, el segmento de recursos y el segmento de ejecución [1]. El segmento de código contiene únicamente las instrucciones ejecutables del proceso y, de ser necesario, información sobre posibles puntos de arranque para dicho proceso. El segmento de recursos contiene referencias a todos los recursos externos utilizados por el proceso. El egmento de ejecución contiene el estado de ejecución actual del proceso.

La definición de mecanismos de movilidad establece una clasificación que puede observarse en la Figura 1. Se pueden clasificar en dos grandes grupos, conocidos como mecanismos de movilidad débil y fuerte respectivamente.

1. Movilidad débil

   Por movilidad débil se entiende un mecanismo que permite migrar únicamente el segmento de código de un proceso, de forma que este pueda comenzar su ejecución en

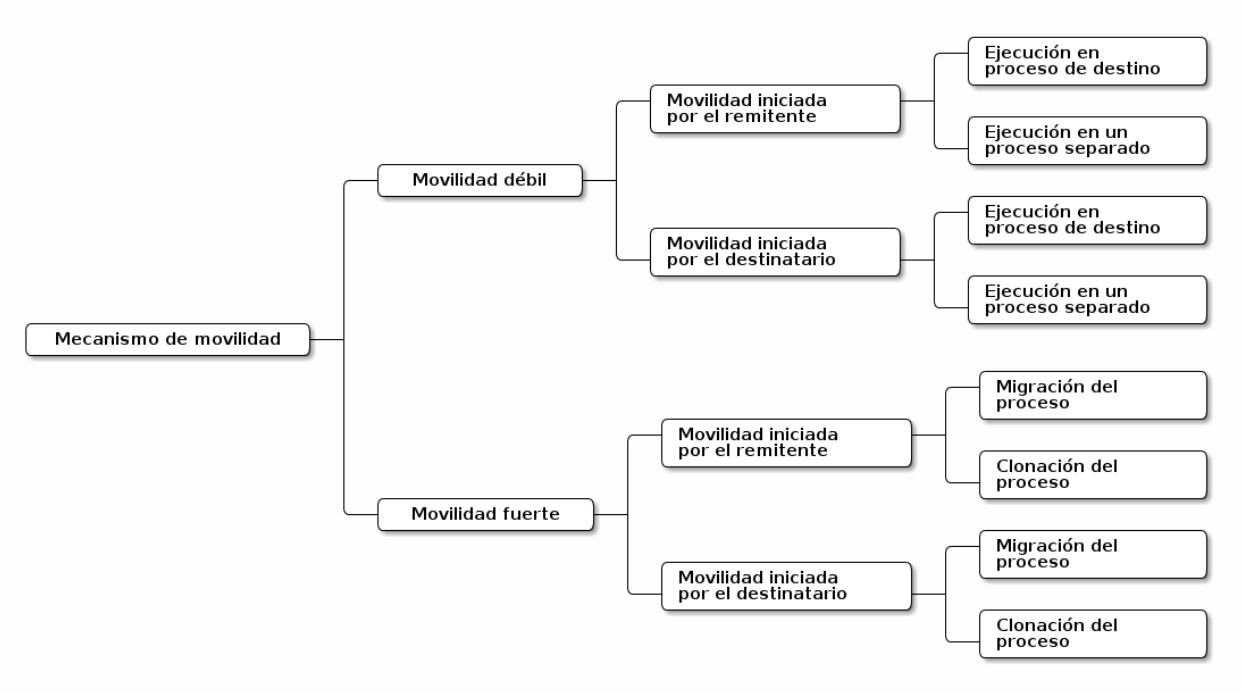

Figura 1: Modelos de migración de código. Figura recuperada de [1].

alguno de los puntos de inicio definidos por dicho segmento. Un ejemplo de un sistema que realiza migración de código con este mecanismo es el sistema de *Applets* de Java.

Independientemente de si la migración débil es iniciada por el remitente o el destinatario, el código migrado puede ser ejecutado como parte del proceso que inició la migración, o puede ser ejecutado en un proceso diferente creado en el nodo destino para tal fin, tal como se observa en la Figura 1.

2. Movilidad fuerte

   Con un mecanismo de movilidad fuerte se permite migrar tanto el segmento de código como el segmento de ejecución de un proceso. De esta forma un proceso puede retomar su ejecución en el nodo de destino desde el punto en que fue detenido en el nodo de origen, manteniendo (en teoría) todo su estado y espacio de memoria. Este mecanismo se implementa como un servicio al nivel del *kernel* del sistema operativo, o como una herramienta de lenguaje de programación al nivel del compilador o *runtime* del lenguaje en cuestión [2].

   La movilidad fuerte se divide en dos esquemas dependiendo de como se realice el traspaso del estado del proceso del nodo de origen al nodo destino. Si el proceso es trasladado completamente entre ambos nodos quedando únicamente una instancia del mismo en el nodo destino se dice entonces que se ha realizado una migración del proceso. Por otra parte, si al terminar la migración del código persiste una instancia del mismo

en el nodo origen junto a un duplicado exacto de dicho proceso en el nodo destino se habla entonces de clonación de procesos [1].

1.2.2. Políticas de Migración

Citando a Goscinski las políticas de migración "deberían definir **cuando** migrar **que** proceso y a **donde**, para lograr el mejor desempeño" [2]. Así mismo, las políticas pueden definir cuando un proceso que ha migrado debe ser desalojado del nodo que lo recibió [2]. Estas políticas pueden implementarse mediante un coordinador centralizado, o de forma descentralizada en cada nodo del sistema.

1.3. Migración del segmento de recursos

Independientemente de las políticas tomadas y los mecanismos utilizados para realizar la migración de los segmentos de código y ejecución, la migración del segmento de recursos debe tratarse con especial cautela utilizando mecanismos que dependerán de la clase de enlace a la que pertenezcan los recursos que esté utilizando el proceso a migrar. En general los recursos se pueden clasificar según tres clases de enlace [1]:

Enlace por identificador Los recursos son accedidos mediante nombres que hacen referencia a recursos específicos y únicos.

Enlace por valor Cuando un proceso utiliza recursos enlazados por valor es porque necesita del contenido específico del recurso independientemente de quien lo provea.

Enlace por tipo Los recursos enlazados por tipo tienen la propiedad de que pueden sustituirse por cualquier otro recurso equivalente.

Desde otro punto de vista, los recursos pueden clasificarse según el grado de relación que posean con el nodo al cual pertenecen. Dicha clasificación es la siguiente [1]:

Recursos no adjuntos Estos recursos no tienen relación directa con el nodo y pueden ser transferidos libremente al momento de migrar código.

Recursos adjuntos En este caso los recursos poseen una relación fuerte con sus nodos respectivos y resulta difícil o costoso el transferirlos entre nodos.

Recursos fijos Estos recursos forman parte integral del nodo al cual pertenecen y no es posible transferirlos o duplicarlos de ninguna manera.

Tomando en cuenta los tipos de enlace de los recursos locales y el grado de relación que poseen con sus nodos respectivos se pueden plantear 9 combinaciones de acciones que se pueden tomar al momento de migrar un proceso con respecto a los recursos locales que esté utilizando. Estas acciones se resumen en la Tabla 1, donde las siglas MV, GR, CP y RV toman el siguiente significado:

MV Trasladar el recurso completo.

GR Establecer una referencia global al recurso (ej.: un URL).

CP Copiar el valor del recurso en cuestión.

RV Enlazar el proceso a un recurso de valor equivalente en el destino.

Tabla 1: Acciones para la migración de recursos locales. Tabla recuperada de [1].

	No adjunto	**Adjunto**	**Fijo**
Por identificador	MV (o GR)	GR (o MV)	GR
Por valor	CP (o MV, GR)	GR (o CP)	GR
Por tipo	RV (o MV, CP)	RV (o GR, CP)	RV (o GR)

1.3.1. Recursos de comunicación

Una clase de recursos particularmente difíciles de migrar son los recursos creados por los distintos mecanismos de comunicación entre procesos. Cada caso particular debe tratarse de manera distinta. Por ejemplo, los segmentos de memoria compartida pueden tratarse con un esquema de referencias globales, lo cual implica que el sistema completo debe tener soporte para mecanismos de memoria compartida distribuida [1]. Por otra parte los puertos locales pueden ser enlazados de nuevo en el nodo destino; sin embargo, es necesario establecer algún mecanismo mediante el cual los procesos remotos que mantienen comunicación con el proceso migrante sean capaces de ubicar a dicho proceso en su nuevo nodo en el sistema. Esto se puede resolver estableciendo un proceso puente en el nodo de origen el cual se encargaría de redirigir los mensajes enviados al nodo migrante hacia el nodo destino. En el caso de las señales, dos estrategias a utilizar son la replicación de las señales en el nodo destino, o sencillamente descartar todas las señales pendientes al momento de realizar la migración.

1.4. Negociación de la migración

Es necesario establecer un procedimiento que permita que dos nodos en un sistema distribuido sean capaces de acordar la realización de un proceso de migración. Este procedimiento se conoce como negociación de la migración, y en el caso de la migración fuerte se establece con un protocolo al nivel de los *kernels* de los sistemas operativos locales a cada nodo [2]. Cada sistema operativo que da soporte a migración de procesos define sus propios protocolos de la misma manera que establecen sus propios mecanismos y políticas de migración de procesos. En esta sección se examina el protocolo de negociación de migración utilizado por el sistema operativo distribuido Charlotte.

1.4.1. Negociación de la migración en Charlotte

Charlotte es un sistema operativo distribuido que proporciona mecanismos de paso de mensajes independientes de la ubicación de los procesos [3]. Este sistema operativo utiliza un mecanismo de migración basado en tres etapas: negociación, transferencia y limpieza. El procedimiento es implementado mediante procesos livianos asociados al *kernel* conocidos como *kernjobs* y se ilustra en la Figura 2. En dicha Figura se puede apreciar el proceso de migración del proceso P1 desde el segundo nodo identificado de izquierda a derecha, el cual va a ser transportado al cuarto nodo del sistema.

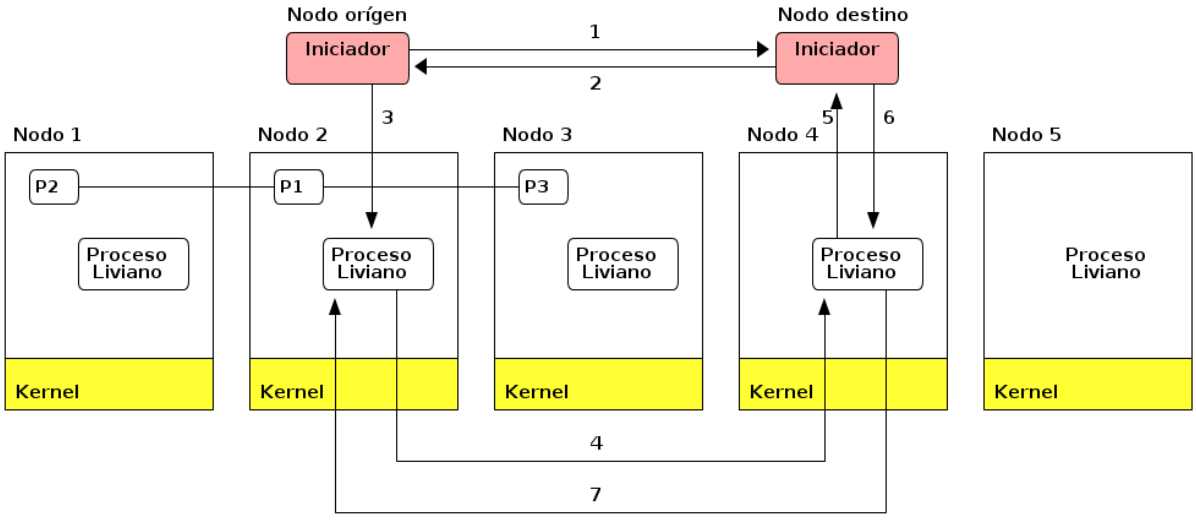

Figura 2: Migración de procesos en Charlotte. Figura recuperada de [2].

Siguiendo la numeración de los mensajes dada en la Figura 2, podemos ver que el proceso de negociación comienza con un mensaje de solicitud de migración enviado por un proceso liviano llamado iniciador (mensaje 1). Este proceso iniciador se coloca separado de los nodos en la Figura 2 porque este puede pertenecer a cualquier nodo en el sistema, sin necesidad de estar asociado al mismo nodo que desea migrar uno de sus procesos. El mensaje de solicitud de migración es recibido por otro proceso iniciador el cual puede aceptar o rechazar la solicitud, enviando su respuesta en el mensaje 2. De ser aceptada la migración, el iniciador correspondiente redirige dicha aceptación al *kernel* al cual pertenece el proceso a migrar, el cual puede consistir en una llamada al sistema si el iniciador se encuentra en el mismo nodo que el *kernel* de origen, o puede ser un mensaje propiamente dicho en caso de que el iniciador sea externo al nodo de origen.

Posteriormente, el proceso liviano encargado de realizar la migración de procesos en el nodo de origen envía un mensaje con toda la información relevante del proceso a migrar hacia su par en el nodo destino (mensaje 4). Este mensaje contiene estadísticas sobre el uso del CPU y la red por parte del proceso a migrar, así como el tamaño de su espacio

de memoria y sus segmentos. Si el proceso liviano en el nodo destino acepta la migración, entonces redirige este mensaje al iniciador asociado a dicho nodo (mensaje 5). Este iniciador tiene la posibilidad de rechazar el proceso de migración. En caso de que el iniciador acepte la migración en esta etapa, este envía un mensaje de inicio de migración de vuelta al proceso liviano en el destino (mensaje 6), el cual es reenviado al proceso liviano equivalente en el nodo de origen (mensaje 7).

1.5. Desalojo de procesos

De la misma forma que un nodo de un sistema distribuido puede aceptar un proceso externo para su ejecución, este nodo pudiera querer desalojar dicho proceso externo cuando su carga se torna muy alta, o cuando se desea dar prioridad a los procesos originados localmente. Para implementar este mecanismo se define al nodo originario de cada proceso como su nodo hogar. Todo proceso ejecutándose en un nodo diferente a su nodo hogar es conocido como un proceso foráneo. En principio, el desalojo de un proceso consiste en retornarlo a su nodo hogar, acción que no puede ser rechazada por el *kernel* de dicho nodo. Que acciones tomar con respecto al proceso desalojado en el nodo de origen dependerán de cada sistema operativo distribuido particular.

1.6. Automigración

Por automigración se entiende a la capacidad de un proceso de poder iniciar espontaneamente y por decisión propia su migración hacia otro nodo del sistema. Esta funcionalidad fue inventada y desarrollada para el sitema operativo distribuido LOCUS y luego heredada por la herramienta TCF (*Transparent Computing Facility* - Instalaciones para Cómputo Transparente) del sistema operativo AIX de IBM [4]. Este mecanismo era implementado en estos sistemas mediante la llamada al sistema `migrate`, la cual permitía al proceso que realizaba la llamada el solicitar un proceso de migración a un destino específico [5].

2. Llamadas a procedimientos remotos

En lo que se refiere a comunicación entre procesos en un sistema distribuido existen muchas alternativas, la más común de las cuales es el mecanismo de paso de mensajes basado en las primitivas `send` y `receive` [1]. Bajo este esquema un proceso cliente solicita la ejecución de un servicio a un proceso servidor (usualmente ubicado en un nodo diferente de la red) enviándole un mensaje de solicitud con la primitiva `send` y luego bloqueándose hasta tanto no reciba una respuesta del servidor con la primitiva `receive`. En este caso el servidor realiza la operación análoga invirtiendo el orden en que ejecuta las primitivas mencionadas. Mediante este mecanismo la comunicación entre el cliente y el servidor se realiza de forma explícita.

Una alternativa a esto es el mecanismo de RPC (*Remote Procedure Call* - Llamadas a Procedimientos Remotos), el cual permite abstraer la comunicación desde el punto de vista de los desarrolladores, los cuales pueden escribir el código fuente que ejecutarán estos procesos

utilizando llamadas a procedimientos clásicas. Estas llamadas son convertidas en solicitudes de ejecución remota de forma transparente por el compilador.

2.1. Definición y sutilezas

El mecanismo de RPC se define como un acercamiento lingüístico al paradigma de comunicación cliente-servidor [2]. En términos de lenguajes de programación, el servidor se puede visualizar como un módulo de biblioteca el cual exporta una interfaz pública que los procesos clientes pueden utilizar como si de un módulo local se tratara, siendo responsabilidad del compilador el resolver la correcta ejecución de estos métodos remotos.

Sin embargo, la semántica de una llamada a un procedimiento remoto tiene que forzosamente diferir de una llamada a procedimiento local. En efecto, los mecanismos de RPC funcionan bajo las siguientes consideraciones semánticas [6]:

- Los procedimientos además de definir que parámetros reciben deben a su vez especificar cuales de estos son parámetros de entrada, cuales son parámetros de salida y cuales son parámetros de entrada/salida.
- Es necesario que el procedimiento remoto especifique el tipo de paso para todos los parámetros, sea por valor, por referencia o por copia-restauración [1].
- El procedimiento remoto no puede tener acceso a ningún estado global o público del cliente.
- No tiene utilidad el pasar direcciones de memoria entre el servidor y el cliente.

Los sistemas basados en RPC se suelen agrupar en dos categorías [6]:

1. Mecanismos integrados al lenguaje de programación.
2. Mecanismos que utilizan un IDL (*Interface Definition Language* - Lenguaje de Definición de Interfaces) para la definición de la interfaz del servidor.

En el caso de los mecanismos integrados al lenguaje, se utilizan las herramientas propias del lenguaje en cuestión para definir la interfaz del mecanismo RPC. En ambos casos se utiliza un compilador especializado para generar las implementaciones del mecanismo a partir de la definición de la interfaz. Esto se detalla en la sección 2.2.1.

2.2. Implementación

Todo sistema de RPC debe cumplir con las siguientes tres tareas [6]:

Procesamiento de la interfaz La integración del mecanismo de RPC con el lenguaje de programación.

Manejo de la comunicación El envío y recepción de solicitudes y respuestas.

Enlazado Se refiere a localizar al servidor adecuado para atender una solicitud específica.

Para llevar a cabo estas tareas los sistemas de RPC suelen ser implementados mediante un tipo especial de procedimientos locales llamados *stubs*. Se deben definir *stubs* tanto para el servidor como para el cliente.

2.2.1. Implementación por *stubs*

En pocas palabras un *stub* es un procedimiento local generado de forma automática, el cual se encarga de preparar los parámetros de la llamada remota en el caso del cliente, y de seleccionar el método a ejecutar pasándole los parámetros recibidos en el caso del servidor [6]. Estos procedimientos se generan con un compilador especial a partir de la definición de interfaz. El funcionamiento de este mecanismo se muestra en la Figura 3. Los procesos de *marshalling* y *unmarshalling* se describen en la Sección 2.3.

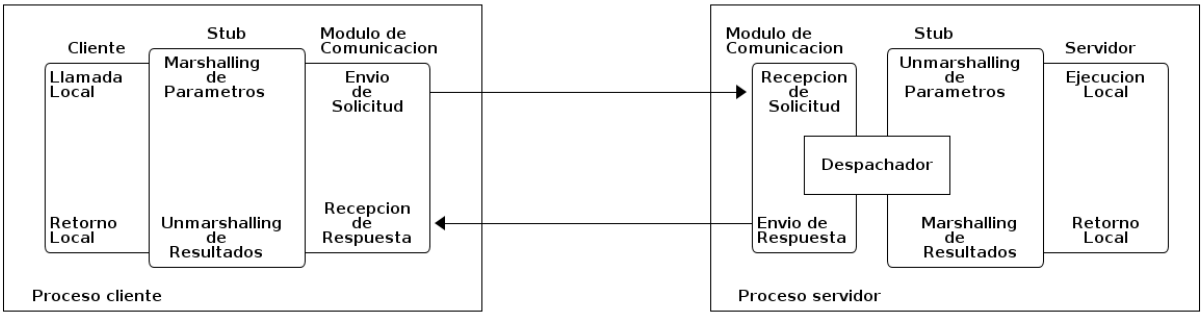

Figura 3: Implementación de RPC mediante *stubs*. Figura recuperada de [6].

2.3. Paso de parámetros y resultados

Debido a que el servidor y el cliente probablemente residirán en computadoras diferentes las cuales pueden tener arquitecturas diferentes, se hace necesario establecer un mecanismo mediante el cual ambos procesos sean capaces de comunicar los parámetros y los resultados de las llamadas RPC. Se identifican tres problemas fundamentales en este respecto [1]:

- La recepción de los parámetros en el orden correcto del lado del servidor.
- Mantener el significado de los parámetros y los resultados independientemente del ordenamiento de bytes utilizado.
- Conservar la semántica de paso de parámetros definida por la interfaz.

2.3.1. Tipos de paso de parámetros

Existen 3 semánticas de paso de parámetros utilizados en los mecanismos de RPC [1]:

1. Paso de parámetros por valor.
2. Paso de parámetros por referencia.
3. Paso de parámetros por copia-restauración.

Para pasar parámetros por valor estos sencillamente se copian en los mensajes de solicitud y se envían directamente al servidor. El proceso de *marshalling* es el encargado de llevar a cabo esta acción.

El paso de parámetros por referencia es considerablemente más complejo dado que una referencia a un objeto en memoria solo tiene sentido dentro del espacio de memoria del proceso que generó dicha referencia, y las soluciones suelen utilizar directamente el paso de parámetros por copia-restauración para poder permitir el paso de parámetros por referencia en lo absoluto. El paso de parámetros por copia-restauración realiza el paso del parámetro del cliente al servidor por valor; sin embargo, si el proceso servidor realizó alguna modificación en el valor del parámetro, esta modificación es reenviada al cliente y luego copiada en su espacio de memoria. Esta copia es responsabilidad del *stub* del cliente [1].

2.3.2. *Marshalling* y *unmarshalling*

El proceso de *marshalling* y su operación inversa, el *unmarshalling*, se refieren al proceso de transformar la representación de los datos transferidos entre el cliente y el servidor entre una representación canónica y la representación local de cada proceso involucrado en la comunicación, además de realizar el empaquetado y desempaquetado de los datos enviados de forma que estos sean recibidos en el orden correcto [1].

El *marshalling* se encarga de convertir todos los datos a ser enviados de los tipos de datos utilizados por el lenguaje de programación a tipos de datos especificados por el protocolo de comunicación subyacente al mecanismo de RPC. Así mismo, este proceso se encarga de empaquetar los parámetros y los resultados en el orden especificado por la semántica de paso de parámetros del protocolo de comunicación. Se suelen utilizar hasta tres semánticas de paso de parámetros:

Paso de parámetros tipo C Los parámetros se empaquetan en el orden inverso en el cual están declarados.

Paso de parámetros tipo Pascal Los parámetros se empaquetan en el orden en el cual están declarados.

Paso de parámetros por nombre Los parámetros tienen un nombre asociado el cual es utilizado por el *stub* del servidor para colocarlos en el orden correcto.

El proceso de *unmarshalling* realiza las acciones inversas al proceso de *marshalling*.

2.4. Clases de errores y tolerancia a fallos

Como todo mecanismo de comunicación en red, las llamadas a procedimientos remotos pueden fallar por diversas razones. En esta Sección se distinguen 5 tipos de fallos posibles al utilizar este mecanismo de comunicación [1].

2.4.1. El cliente no puede encontrar al servidor

En este caso el cliente no puede contactar al servidor. Existen dos soluciones a este problema: se puede retornar un valor especial para indicar el error (por ejemplo -1) o levantar una excepción si el lenguaje de programación lo permite.

2.4.2. Se pierde la solicitud

Si la solicitud se pierde en tránsito es posible realizar una retransmisión. En caso de que las retransmisiones se pierdan en el camino también, entonces el cliente puede considerar que el servidor es inalcanzable por alguna razón y tomar las medidas explicadas anteriormente.

2.4.3. Se pierde la respuesta

En este caso la solución depende de si la solicitud es idempotente o no. En el caso de las solicitudes idempotentes es posible utilizar retransmisiones. Un ejemplo de una solicitud idempotente es leer una cantidad de bytes de un archivo.

En el caso de que las solicitudes no sean idempotentes (por ejemplo transferir fondos de una cuenta bancaria a otra), se puede utilizar un mecanismo de identificadores y números de secuencia, de forma que el servidor tenga posibilidad de reconocer solicitudes duplicadas para no ejecutarlas nuevamente.

2.4.4. El servidor falla después de recibir una solicitud

Todo servidor RPC debe realizar 3 pasos para atender una solicitud: primero debe recibir la solicitud, luego debe ejecutar la solicitud y finalmente debe enviar una respuesta. Sin embargo, el servidor puede fallar ya sea inmediatamente después de recibir una solicitud, o durante la ejecución de la misma. Para resolver este problema se presenta la dificultad de que el cliente no puede conocer en que punto falló el proceso servidor.

Se conocen tres tipos de soluciones que afectan la semántica de la ejecución de la llamada RPC:

Semántica "al menos una vez" El cliente puede retransmitir las solicitudes y el servidor garantiza que la llamada RPC se ejecutará completamente por lo menos una vez.

Semántica "a lo sumo una vez" El cliente debe reportar un error si la primera solicitud falla de cualquier manera.

Semántica sin garantías El cliente puede retransmitir las solicitudes pero el servidor no da garantías con respecto a la cantidad de veces que se puede haber ejecutado una solicitud.

2.4.5. El cliente falla antes de que se envíe la respuesta

Para solucionar este problema se han propuesto múltiples soluciones. Una de estas es un mecanismo de expiración, el cual asocia a cada solicitud de RPC un tiempo T para ser ejecutada. Si el servidor necesita más tiempo para resolver la solicitud entonces debe pedirlo explícitamente al cliente. Si el cliente no responde o no es alcanzable entonces el servidor puede abortar la ejecución de la solicitud en cuestión.

3. Manejo de *deadlocks* distribuidos

Un *deadlock* o interbloqueo es una situación de error que ocurre en los sistemas operativos como consecuencia de ciertas propiedades del mecanismo de asignación de recursos [1]. Formalmente un interbloqueo se define como "el estado dependiente del tiempo de bloqueo permanente de un conjunto de procesos los cuales interactúan sea directamente comunicándose entre si, o indirectamente al competir por recursos" [2]. Es necesario que el sistema operativo cumpla con las siguientes 3 condiciones para que sea posible la ocurrencia de un interbloqueo [8]:

Exclusión mutua Un proceso del sistema puede tener acceso exclusivo a los recursos.

Retención y espera Un proceso del sistema puede retener recursos mientras esta bloqueado en espera de que se le asignen otros recursos.

No apropiación El sistema no puede revocar a un proceso el acceso a un recurso que ya le ha sido otorgado.

3.1. Modelos de interbloqueos distribuidos

En los sistemas distribuidos pueden presentarse dos tipos de interbloqueo: los interbloqueos por recursos y los interbloqueos por comunicación [7].

3.1.1. Modelo de recursos

Los interbloqueos por recursos son aquellos que se presentan cuando existen relaciones circulares de dependencia entre los procesos de un conjunto, donde cada proceso está en espera a que otro proceso del conjunto libere un recurso que este necesita, mientras que al mismo tiempo retiene recursos que otros procesos del conjunto necesitan. Un ejemplo de un interbloqueo por recursos puede observarse en la Figura 4. En dicha Figura, los circulos representan procesos mientras que los cuadrados recursos. Una arista de un proceso a un

recurso representa que el proceso está solicitando ese recurso, mientras que una arista de un recurso a un proceso indica que dicho proceso tiene posesión de ese recurso.

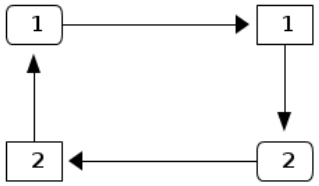

Figura 4: Interbloqueo por recursos.

3.1.2. Modelo de comunicación

Los interbloqueos por comunicación son aquellos que se producen cuando todos los procesos de un conjunto están permanentemente bloqueados a la espera de mensajes que solo pueden ser enviados por otros procesos del conjunto [2]. Este modelo de interbloqueo es más general que el modelo de interbloqueos por recursos y, de hecho, es posible modelar cualquier interbloqueo por recursos como un interbloqueo por comunicación [7].

Se define como conjunto dependiente CD de un proceso bloqueado P al conjunto de todos los procesos que pueden provocar que P se desbloquee mediante el envío de un mensaje, es decir, CD_p contiene todos los procesos de los cuales P está esperando una respuesta. Se dice entonces que un conjunto no vacío C de procesos se encuentra en interbloqueo cuando [2]:

1. Todos los procesos pertenecientes a C están bloqueados.
2. Para todo proceso P perteneciente a C se cumple que: $CD_p \subset C$.
3. No hay mensajes en tránsito entre los procesos de C.

3.2. Estrategias clásicas

Los interbloqueos son un problema ampliamente estudiados en el contexto de los sistemas centralizados, y por lo tanto se han desarrollado una gran variedad de técnicas para poder tratar este problema. Estas se agrupan en 3 grandes categorías que son la evasión, la prevención y la detección y recuperación de interbloqueos. Una técnica adicional, conocida como el algoritmo del avestruz se utiliza para evitar tener que lidiar con los interbloqueos.

3.2.1. Avestruz

La llamada estrategia del avestruz consiste en ignorar la posibilidad de que pueden ocurrir interbloqueos en un sistema cuando la probabilidad de que estos ocurran es tan baja que no se justifica el costo de tratar de proveer una solución real al problema [8].

3.2.2. Evasión

Esta estrategia consiste en llevar un registro de las asignaciones de recursos a procesos, de forma que sea posible determinar si una asignación futura es segura (no conduce a un interbloqueo) o insegura. Un ejemplo de esta estrategia es el algoritmo del banquero [8]. Esta estrategia se caracteriza por necesitar la construcción de un mapa global de asignación de recursos lo que implica la difusión de estas asignaciones por la red y la necesidad de lograr convergencia en la construcción de este mapa, además de que suele ser necesario conocer las necesidades de recursos de cada proceso a priori. Estas razones hacen a la evasión de interbloqueos inviable en los sistemas distribuidos [2].

3.2.3. Prevención

La estrategia de prevención de interbloqueos consiste en eliminar alguna de las condiciones establecidas al comienzo de la Sección 3 responsables de la posibilidad de interbloqueos [8]. Eliminar dichas condiciones implica eliminar características deseables del sistema, o la necesidad de imponer cierta estructura a los procesos, como por ejemplo:

1. Eliminar la condición de exclusión mutua implica crear la posibilidad de corrupción de datos dado que múltiples procesos serían capaces de utilizar simultaneamente recursos compartidos que no se prestan a esto.

2. Eliminar la condición de retención y espera se puede lograr haciendo que un proceso solicite todos los recursos que va a necesitar al comenzar, lo que no siempre es posible. Alternativamente se puede lograr obligando a los procesos a solicitar recursos por grupo, siendo necesario que liberen todos los recursos que poseen si no es posible asignarles todos los recursos que solicitan simultaneamente. Esto último disminuye el nivel de concurrencia del sistema.

3. Eliminar la condición de no apropiación se logra permitiendo que el sistema pueda revocar el acceso a algún recurso a cualquier proceso en cualquier momento, lo cual no siempre es seguro.

Por estas razones la prevención de interbloqueos es considerada impráctica [2].

3.2.4. Detección y resolución

Bajo esta estrategia se permite que ocurran interbloqueos, los cuales deben ser detectados mediante algún algoritmo y luego resueltos de alguna manera. La detección puede realizarse con estrategias centralizadas, distribuidas o jerárquicas [2].

Las técnicas distribuidas utilizan distintas estrategias para determinar la existencia de interbloqueos. Una de estas estrategias es la constucción del grafo de asignación de recursos en cada nodo evitando la necesidad de tener un nodo dedicado a su construcción, lo que elimina el punto de falla de las estrategias centralizadas. Sin embargo, la construcción del

grafo en cada nodo introduce problemas de consistencia, lo que puede provocar la detección de interbloqueos falsos. Otra estrategia de detección distribuida utiliza un modelo de comunicación conocido como cómputo de difusión (del inglés *diffusing computation*). Esta estrategia utiliza mensajes especializados llamados "sondas" los cuales se utilizan para realizar la detección de ciclos de dependencia sin necesidad de construir el grafo de asignación de recursos. El algoritmo presentado en la Sección 3.4 es un ejemplo de cómputo de difusión.

Finalmente, las estrategias jerárquicas dividen la detección de interbloqeos en tres niveles: local, regional y global. Cada nodo del sistema está encargado de detectar interbloqueos internos, esto define el nivel local. Si uno o más procesos de un nodo necesitan recursos asignados a procesos de otros nodos, estos nodos envían la información de estas asignaciones y solicitudes a un coordinador regional, el cual construye un grafo de asignación de recursos que involucra a todos los nodos que este coordina, y realiza la detección del interbloqueo en dicho grafo. Múltiples coordinadores regionales pueden enviar información a un coordinador global el cual realizar el mismo proceso. Esta jerarquización puede extenderse al infinito [2].

3.3. Algoritmos de detección centralizados

Las estrategias centralizadas utilizan un ente coordinador central encargado de realizar la detección y consisten en la construcción de un grafo de asginación de recursos global y la busqueda de ciclos en este. Estas estrategias se caracterizan por tener un punto de falla y cuello de botella en la figura del coordinador central [2].

3.4. Algoritmo de Chandy-Misra-Haas

El algoritmo de Chandy-Misra-Haas, propuesto originalmente en [7], es un algoritmo clásico, correcto y representativo de las estrategias tomadas para realizar la detección de interbloqueos distribuidos [2]. Este algoritmo propone dos variantes, una para interbloqueos por recursos pensada para sistemas de bases de datos distribuidas, y otra más general para interbloqueos por comunicación.

3.4.1. Modelo de recursos

Este modelo asume que cada proceso del sistema esta asociado a un proceso especial llamado controlador, el cual es el encargado de detectar los interbloqueos. Un controlador puede ser responsable por más de un proceso. Los controladores utilizan un tipo de mensaje llamado "sonda" el cual es un vector de tres elementos (i, j, k). Una sonda (i, j, k) indica que se está tratando de detectar si el proceso P_i está en interbloqueo, siendo la sonda un mensaje del proceso P_j al proceso P_k. La recepción de una sonda (i, j, i) por parte del controlador de P_i, indica que dicho proceso se encuentra en interbloqueo [7].

3.4.2. Modelo de comunicación

En este modelo no existen controladores y todo el cómputo del algoritmo lo realizan los procesos directamente. El proceso que inicia un cómputo se conoce como iniciador. En este modelo existen dos tipos de sondas, las solicitudes o *queries* y las respuestas o *replies*. Las sondas independientemente de su tipo son de la forma $query(i, m, j, k)$ y denotan que esta sonda pertenece al m-ésimo cómputo comezado por el proceso P_i y es enviada del proceso P_j al proceso P_k.

Se distinguen las siguientes propiedades:

1. Si un proceso P_i está en interbloqueo al comenzar el cómputo de difusión, entonces recibirá una respuesta de la forma $reply(i, m, j, i)$ por cada solicitud $query(i, m, i, j)$ que envíe.

2. Si el iniciador P_i ha recibido una respuesta $reply(i, m, j, i)$ a cada solicitud $query(i, m, i, j)$ que ha enviado, entonces está en interbloqueo.

Cada proceso P_k debe mantener los siguientes arreglos, los cuales tendrán una posición por cada proceso del sistema:

$ultimo(i)$ El mayor número de secuencia m visto por P_k en cualquier solicitud recibida o enviada por si mismo. Incialmente es 0 para todo i.

$modificador(i)$ El identificador del proceso P_j que provocó el cambio más reciente en $ultimo(i)$. El valor inicial es arbitrario para todo i.

$num(i)$ Es el número total de mensajes de la forma $query(i, m, k, j)$ enviados por P_k menos el total de respuestas $reply(i, m, j, k)$ recibidas por P_k, donde $m = ultimo(i)$.

$espera(i)$ Este arreglo booleano es verdadero si y solo si P_k si P_k ha estado inactivo continuamente desde que $ultimo(i)$ fue actualizado por última vez. Inicialmente es falso para todo i.

El Algoritmo 1 es ejecutado por un proceso P_i para iniciar el cómputo de difusión.

$ultimo(i) \leftarrow ultimo(i) + 1$;
para todo $P_j \in S$ *donde* S *es el conjunto dependiente de* P_i **hacer**
  Enviar una sonda $query(i, ultimo(i), j, k)$;
fin
$num(i) \leftarrow\mid S \mid$;

Algoritmo 1: Algoritmo de inicio de cómputo para el modelo de comunicación.

El Algoritmo 2 es ejecutado por un proceso P_i al comenzar a ejecutarse.

El Algoritmo 3 es ejecutado por un proceso inactivo P_k al recibir una sonda de la forma $query(i, m, j, k)$.

El Algoritmo 4 es ejecutado por un proceso P_k al recibir una respuesta de la forma $reply(i, m, r, k)$.

para todo i **hacer**
  $espera(i) \leftarrow$ **false**;
fin
Descartar toda solicitud o respuesta recibida;

Algoritmo 2: Algoritmo de fin de cómputo para el modelo de comunicación.

si $m > ultimo(i)$ **entonces**
  $ultimo(i) \leftarrow m$;
  $modificador(i) \leftarrow j$;
  $espera(i) \leftarrow$ **verdad**;
  para todo $P_r \in S$ *donde* S *es el conjunto dependiente de* P_k **hacer**
    Enviar sonda $query(i, m, k, r)$;
  fin
  $num(i) \leftarrow \mid S \mid$;
en otro caso
  si $espera(i)$ ***y*** $m = ultimo(i)$ **entonces**
    Enviar respuesta $reply(i, m, k, j)$;
  fin
fin

Algoritmo 3: Algoritmo de recepción de solicitud para el modelo de comunicación.

Conclusiones

En este trabajo se presentó un panorama de tres tópicos en gestión de procesos distribuidos. Primero se presentó el tema de migración de procesos, el cual consiste en un mecanismo que permite a un proceso en un sistema distribuido el pasar de un nodo del sistema a otro a través de la red. Se describieron así mismo las principales dificultades que se presentan al migrar un proceso, junto a un mecanismo de negociación de la migración.

En segunda instancia se detalló el mecanismo de comunicación entre procesos conocido como llamadas a procedimientos remotos, o RPC. Este mecanismo permite a un proceso el poder ejecutar de forma transparente un procedimiento definido en un proceso diferente, posiblemente en un nodo distinto del sistema e independientemente de los lenguajes de programación utilizado y la arquitectura de los mismos.

Finalmente se realizó un estudio sobre las técnicas para tratar interbloqueos en sistemas distribuidos. Se hizo especial énfasis en las técnicas de detección de interbloqueos por ser estas las más factibles de ser implementadas de forma distribuida [2].

Referencias

[1] A. Tanenbaum y M. Van Steen, *Sistemas Distribuidos: Principios y Paradigmas*, 2^a Edición, Pearson, 2007.

```
si m = ultimo(i) y espera(i) entonces
    num(i) ← num(i) − 1;
    si num(i) = 0 entonces
        si i = k entonces
            Declarar interbloqueo;
        en otro caso
            Enviar respuesta reply(i, m, k, j) a P_j donde j = modificador(i);
        fin
    fin
fin
```

Algoritmo 4: Algoritmo de recepción de respuesta para el modelo de comunicación.

[2] A. Goscinski, *Distributed Operating Systems: The Logical Design*, 1^a Edición, Addison-Wesly Publishing Company, 1991.

[3] R. Finkel y Y. Artsy, *The Process Migration Mechanism of Charlotte*, Operating Systems Technical Commitee Newsletter, pp. 11-14, 1989.

[4] B. Walker y R. Mathews, *Process Migration in AIX's Transparent Computing Facility (TCF)*, IEEE Technical Committee on Operating Systems Newsletter, vol. 3, No. 1, pp. 5-7, 1989.

[5] G. Popek, *The LOCUS distributed system architecture*, 1^a Edición, MIT Press, 1985.

[6] G. Colouris, J. Dollimore y T. Kindberg, *Distributed Systems: Concepts and Design*, 2^a Edición, Addison-Wesley, 1994.

[7] K. Mani Chandy, J. Misra y L. Haas, *Distributed deadlock detection*, ACM Transactions on Computer Systems (TOCS), vol. 1, No. 2, pp. 144-156, 1983.

[8] A. Tanenbaum, *Sistemas Operativos Modernos*, 3^a Edición, Pearson, 2008.