Universidad de Los Andes
Facultad de Ingeniería
Postgrado de Computación

Técnicas de Localización de Objetos Móviles

Andrés Emilio Arcia Moret
Tutor: Dr. Leandro León

Mérida, 31 de enero de 2002

A Yoli y a Andrés Alejandro.

Agradecimiento

Este trabajo de tesis no hubiera podido ser desarrollado sin la valiosa colaboración de varias personas e instituciones.

En primer lugar, el Consejo Nacional de Ciencia y Tecnología (CONICIT) me brindó el soporte económico indispensable para realizar mis estudios de postgrado.

El Centro de Estudios en Microelectrónica y Sistemas Distribuidos (CEMISID) donde fue desarrollada enteramente la tesis. Pues, todos sus recursos, humanos y materiales, forman parte de lo que en este manuscrito se presenta.

También se utilizaron recursos del Centro Nacional de Cálculo Científico (CeCalCULA). Vaya entonces mi agradecimiento a Juan Luis Chaves y a Gilberto Díaz, quienes colaboraron en la administración de la red.

Carlos Nava, quien programo el sistema IPC, fue también de gran ayuda durante el desarrollo y las pruebas del sistema programado.

Finalmente, mi más sincero y especial agradecimiento va dirigido a mi tutor, el profesor Leandro León, quien durante todo el desarrollo de la tesis siempre tuvo el mismo entusiasmo y convicción para lograr lo que hoy ya se presenta como un prototipo operativo para la localización de objetos móviles. Muchas y valiosas lecciones sabré eternamente agradecerle.

Resumen

Este trabajo describe la utilización de técnicas efectivas de localización de objetos móviles en un sistema distribuido. El principio fundamental de la localización es el uso intensivo de *caches* distribuidos a través de todos los sitios del sistema. Para mantener los caches vigentes, se identifican los eventos que pueden actualizarlos y se utilizan dos técnicas: *prefetching* y *piggybacking*. En el caso extremo que el *caching* falle, se utiliza un protocolo de difusión en cascada que permite recuperar con certitud la dirección del objeto.

Las medidas de rendimiento demuestran que la combinación de estas técnicas resulta en un sistema de localización totalmente distribuido, de alta escala, tolerante a fallas y de alto desempeño. En ninguno de los experimentos efectuados fue necesario apelar a la difusión.

Palabras clave: sistemas distribuidos, localización de objetos, invocación remota, objetos, migración.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

El presente trabajo trata sobre el diseño y el desarrollo de un sistema distribuido de localización de objetos móviles. Un sistema distribuido a objetos está formado por máquinas autónomas interconectadas en red. En el sistema cada entidad es vista como un objeto que se encuentra unívocamente identificado.

Los objetos pueden comunicarse entre si, aún si se encuentran en distintos sitios, a través de invocaciones. Cuando un cliente desea acceder un objeto, éste hace uso de una *referencia*. Por medio de ésta, se efectúa la *invocación* especificando el método que realizará la operación requerida. Luego de procesar la invocación, el objeto retorna una respuesta al cliente.

En algunos sistemas, los objetos tienen la propiedad de moverse entre los procesos y los sitios. A este acto se le denomina migración. La movilidad de los objetos trae consecuencias. Cuando un objeto se mueve, todas sus referencias pierden validez.

A tal efecto, es necesario utilizar algún mecanismo para actualizar las referencias inválidas; este mecanismo se conoce como localización de objetos. La localización de objetos es, pues, el acto de actualizar las referencias inválidas de los objetos que han migrado.

Esta tesis propone y desarrolla un sistema de localización de objetos, aun si estos se encuentran en movimiento. El sistema exhibe escalabilidad y transparencia, características importantes en un sistema distribuido. El sistema es altamente distribuido, lo que lo hace escalable. Por otro lado, la transparencia es alcanzada mediante métodos que actualizan constantemente la información sobre la localización de un objeto. Por último, como se puede observar en las medidas, el sobrecoste para la invocación es pequeño.

1.1. Los problemas de la localización

Entre las técnicas de localización de objetos se encuentran la difusión, los lazos de persecución, la prueba y actualización, etc. Todas estas técnicas varían en costo, siendo la difusión la más costosa.

Un método cándido para actualizar una referencia a objeto es la difusión. Desde la perspectiva del objeto, consiste en realizar una difusión a sus clientes notificándole su nueva dirección.

También existe la difusión desde la perspectiva del cliente del objeto. Esto consiste en efectuar una difusión donde se pregunta sobre la dirección del objeto antes de invocar.

El propietario del objeto responde con la dirección correcta.

El método más popular es el de los lazos de persecución (*forwarding*). Esta técnica consiste en dejar una marca que apunte hacia el sitio donde se movió el objeto. Este procedimiento puede repetirse tantas veces como el objeto migre.

Otro método es la prueba y actualización, que consiste en hacer un test previo a la invocación para saber si efectivamente el objeto se encuentra donde señala la referencia.

Las técnicas mencionadas tienen debilidades; principalmente en escalabilidad, desempeño y en la sensibilidad a fallas. En la difusión, a medida que aumenta el número de sitios, crece el número de mensajes. Esto trae como consecuencia que el proceso de búsqueda sea más lento, pues se emplea un mayor número de mensajes.

Una cadena de lazos de persecución crece a medida que se aumenta el número de migraciones. Una cadena más grande implica un recorrido mayor para alcanzar al objeto. Los lazos de persecución hacen que el sistema sea sensible a fallas, pues un lazo que se pierda, implica la activación de un protocolo de recuperación.

1.2. Los objetivos de esta tesis

Como objetivo de este trabajo, se planteo desarrollar un sistema de localización de objetos móviles que exhibiera características deseables en un sistema distribuido: distribución, transparencia, tolerancia a fallas y versatilidad.

La distribución fue alcanzada a través de caches, los cuales mantienen información sobre los objetos a través del sistema. Además, el sistema se vale de la cooperación mutua para resolver una referencia inválida. Así, la localización de un objeto es un trabajo netamente distribuido.

Se ofrece una interfaz que da la ilusión de tener un sistema centralizado. Existen políticas de intervención de las invocaciones; así, transparentemente éstas se redireccionan al sitio donde haya migrado el objeto.

El sistema es tolerante a fallas, pues la información sobre la localización de los objetos es redundante. Varios localizadores están en capacidad de resolver una referencia inválida. Además, se provee de un protocolo de recuperación con el cual se pudiera resolver inconsistencias en el sistema.

Por último, el sistema desarrollado debe ser portátil entre las distintas versiones de Unix. Así, se tiene un sistema basado sobre IP, programado en $\texttt{C}^{++}$estándar y conforme a POSIX. Además se ofrece un sistema altamente parametrizado, pudiendo entonces adaptarlo a las necesidades del cliente.

1.3. Trabajo realizado

El sistema está basado en una técnica llamada actualización por fracaso de la invocación. Cuando una referencia efectúa una invocación hacia un objeto que ha migrado, ocurre un *fracaso de invocación*. Se emprende, entonces, una búsqueda del objeto. Se dispone de un servicio de localización al cual se puede preguntar y obtener la nueva dirección del objeto.

El servicio de localización es implantado por un conjunto de servidores localizadores que cooperan entre sí. Cada sitio ejecuta un demonio de localización. Los objetos residentes en cada máquina se registran en el servicio de localización local. El conjunto de demonios de localización lleva a cabo la localización de objetos móviles.

Los localizadores utilizan tres técnicas: caching, prefetching y piggybacking. Un cache es una estructura, con capacidad finita, donde se guardan resultados de cómputos para evitar volverlos a realizar. En los caches se encuentra la información sobre el paradero más reciente de un objeto. Para que éstos sean efectivos, hay que mantenerlos lo más actualizados posible. Esto se logra con las técnicas de prefetching, piggybacking y las propias actualizaciones que se hacen al migrar un objeto. Los caches tienen un límite en espacio; eventualmente, los registros en un cache son removidos conforme llegan otros registros más recientes.

Existen caches de entrada, salida, objetos eliminados, migración y nuevas referencias. Los caches de entrada y salida guardan información producto de las invocaciones. El cache de entrada registra los sitios desde donde se han realizado invocaciones. El cache de salida registra las referencias a objetos que han sido invocados. El cache de objetos eliminados registra objetos que han sido sacados del sistema. El cache de migración almacena la nueva dirección de un objeto que ha migrado. Por último, el cache de nuevas referencias guarda referencias de objetos que no se conocen en el sitio pero que probablemente sirvan para futuras localizaciones.

Para mantener actualizados los caches se usa prefetching. Las referencias guardadas en el cache de salida se revisan periódicamente con el fin de actualizarlas antes de que ocurra un fracaso. Por otro lado, también se utiliza el cache de entrada. Como en los registros de este cache queda la dirección de los sitios desde donde se invoca, éstos se pueden actualizar cada vez que un objeto migre.

Otra forma de actualización es la utilización de *piggybacks*. Un *piggyback* es un mensaje que se adjunta a los mensajes de invocación. Estos mensajes son procesados por los demonios localizadores y pueden ser de tres tipos: búsqueda de objeto, eliminación de objeto y anuncio de referencia. Estos mensajes actualizan los caches del conjunto de localizadores conforme se realicen invocaciones en el sistema.

Como último recurso en la localización, se encuentran las difusiones. Mediante difusiones se interroga a los demonios localizadores acerca de la existencia del objeto solicitado. En su versión más simple, consiste en esperar la respuesta del propietario del objeto que se busca.

Los costes asociados a las difusiones han sido amortizados mediante dos distintas técnicas: difusión débil y difusión fuerte.

La difusión débil es el envío de un mensaje de búsqueda no confiable. Por ello, se divide en dos etapas. La primera consiste en el envío de un mensaje de búsqueda pidiendo que se reporte sólo el sitio donde se encuentra el objeto. En la segunda etapa, se difunde un mensaje pidiendo toda la información del objeto que se encuentra en los caches.

La difusión fuerte se refiere al envío de mensajes confiables preguntando por la existencia del objeto. Esto implica un mayor costo, pues cada sitio debe responder al menos un mensaje de confirmación. De no encontrarse el objeto, se ordena un protocolo de reconstrucción de la tabla de objetos en cada demonio localizador. Si la reconstrucción fracasa, el objeto se declara como inexistente.

Todas estas técnicas se exportan a través de una biblioteca de funciones que cada cliente puede encadenar a su sistema y utilizar bajo su propio esquema. Aunque el demonio localizador tiene la capacidad de actualizar referencias inválidas, no se impone con una secuencia de utilización de las técnicas antes mencionadas.

1.4. Nota sobre la terminología

En esta tesis, se utilizan términos en su forma anglosajona. Según nuestras experiencias, el discurso es más comprensible que con sus equivalentes en castellano. Por ejemplo, usamos *cache*[1] en lugar de memoria escondida. Del mismo modo, usamos *piggyback* al no encontrar un término adecuado en castellano.

1.5. Estructura de este documento

El resto del documento está organizado como sigue. En el capítulo 2 se presenta un recorrido por las distintas técnicas de localización de objetos que hicieron hito. Como sección final de este capítulo, se hace una discusión de las técnicas investigadas y se introduce la propuesta de la técnica subyacente a este trabajo: la actualización por fracaso. En el capítulo 3 se hace un planteamiento formal del problema, se presenta la justificación de la solución y se detallan las distintas técnicas de localización. En el capítulo 4 se explica la arquitectura y la implantación de un servicio distribuido de forma genérica. En el capítulo 5 se muestra la interfaz del localizador. En el capítulo 6 se presenta la arquitectura y la implantación del sistema. En el capítulo 7 se expone el desempeño del sistema mediante medidas para diferentes configuraciones del localizador. Finalmente, en el capítulo 8, se presentan las conclusiones y recomendaciones.

[1]La palabra *cache* es de origen francés y significa escondido. No obstante es también utilizada en el idioma ingles.

Capítulo 2

Trabajos relacionados con la localización de objetos

Esta sección se consagrará a recorrer las diversas técnicas de localización que han sido propuestas. Para una mejor comprensión del problema, se utiliza una analogía de un evento más cotidiano: la búsqueda de una persona mudada de dirección.

Un problema parecido a la búsqueda de objetos, pero no igual, esta en la telefonía celular. Las diferencias entre la búsqueda de teléfonos celulares y objetos son identificadas y discutidas en una sección.

En el resto de las secciones, se muestran las técnicas más importantes en la localización de objetos. Finalmente, a través de la discusión, planteamos nuestra perspectiva de lo que debe ser un sistema distribuido de localización de objetos, que evidencia las debilidades de otros sistemas propuestos.

2.1. Búsqueda de una persona mudada de dirección

Como un primer ejercicio para introducir los problemas inherentes a la localización, intentaremos establecer una analogía entre la búsqueda de objetos y la búsqueda de una persona mudada de dirección. En nuestra opinión, la analogía es interesante, pues utiliza una situación de la vida real para introducirnos en los problemas relacionados con la búsqueda de objetos en un sistema distribuido.

Supongamos que es necesario ubicar a un conocido del cual ya no se conoce su dirección de habitación. Es probable que comencemos la búsqueda en algunos sitios donde solía coincidir esa persona. Podemos preguntar la nueva dirección a vecinos del sitio donde vivía, o a un amigo en común. En cualquier caso, existe un punto de referencia inicial desde el cual comenzaremos la búsqueda. Una vez iniciada la búsqueda, es probable que tengamos que recorrer una serie continua de referencias que guíen hasta su paradero. Es posible, también, que la persona buscada se haya mudado varias veces. Esto dificulta más la búsqueda.

Si estos métodos fracasan, quizá sea mejor apelar a métodos más costosos. Por ejemplo, publicar un aviso de solicitud en la prensa de circulación nacional, donde los costes dependen del tamaño del aviso y de otros parámetros. Si este método no sirviese, se pudiera comprar un espacio de televisión donde se anuncie la búsqueda, ampliando así la escala.

Por último y como un recurso muy costoso, podríamos enviar una carta de solicitud a todas las direcciones postales válidas.

También es posible recurrir a la base de datos del censo, teniendo así la última ubicación conocida de la persona de interés. Igualmente, es notorio el gran costo en el que se incurriría al hacer un censo exhaustivo.

Por otro lado, no habría que menospreciar el trabajo que un detective pudiera hacer en estos casos. La experiencia y especialización del detective podría ser mucho más eficaz y barato que otras técnicas.

El alcance de tales métodos influye directamente en el tiempo empleado para la ubicación de la persona. La urgencia de localización, dirige el costo de la búsqueda. Mientras se disponga de menos tiempo para realizar la búsqueda, será necesario emplear métodos más costosos para efectuar la ubicación de las personas.

2.2. Telefonía celular

La telefonía celular presenta símiles con los objetos móviles. La telefonía celular, tiene como función comunicar teléfonos celulares inalámbricos que comparten un medio de transmisión común. La premisa principal de diseño en la telefonía celular es que los teléfonos están en constante movimiento.

Una región está dividida por celdas o áreas que son cubiertas por un conjunto de estaciones base (antenas transmisoras/receptoras). Para efectos de comprensión, cada conjunto de celdas se puede llamar zona. Un conjunto de zonas están cubiertas por otras estaciones. Cada uno de los niveles de estaciones tienen sus respectivas funciones. El primer grupo se encarga de mantener un flujo constante en la comunicación de dos o más teléfonos. El segundo grupo, debe mantener la coherencia de la ubicación de los teléfonos en cada zona.

Como se puede ver, la comunicación dentro del sistema es de forma distribuida. Se puede apreciar la analogía existente entre el sistema de comunicación celular y el comportamiento de los objetos en un sistema distribuido. Los objetos, al igual que los teléfonos, se encuentran agrupados. Un proceso agrupa objetos y los sitios agrupan procesos. Se puede observar la similitud en los niveles de agrupación que plantea la comunicación celular.

Además, se observan comportamientos análogos en los procesos de migración. Mientras un teléfono puede migrar entre celdas y zonas, un objeto puede migrar entre procesos de una misma máquina o entre máquinas distintas.

Posteriormente, en la discusión § 2.8, señalaremos cuales son las diferencias entre un sistema de objetos móviles y la telefonía celular. Así, quedará claro por qué no es posible emplear los métodos de búsqueda de un sistema a otro.

2.3. Difusión

En este enfoque, el ente migrante efectúa una difusión dando a conocer su nueva localización. Existen dos tipos de difusión. El primero consiste en hacer la difusión hacia todo el conjunto de sitios. El segundo consiste en efectuar la difusión hacia los clientes del objeto migrante, con la implicación inmediata de que hay que conocer a todos los objetos enlazados al objeto migrante.

En general, este método sólo es considerado en el caso de que el número de clientes sea pequeño y si la migración es excepcional. Esto es debido a que produce considerables efectos adversos en una red.

En la migración de objetos la utilización pura de este mecanismo implica dos difusiones. La primera, indica que se está ejecutando una migración y así prohibir invocaciones durante el lapso de tiempo que dura la migración; la segunda anuncia la nueva localización y la finalización de la migración.

A pesar de estas objeciones naturales, este enfoque fue considerado pero no implementado por los sistemas **Demos/MP** [14] y **System V** [23].

Una variante de difusión fue utilizada en el sistema **Charlotte** [1]. Este sistema usaba lazos de comunicación que no eran actualizados por el núcleo. Cuando ocurría una migración, los núcleos que contenían procesos que referenciaban al proceso migrante eran informados acerca de la nueva localización. La migración no era ejecutada hasta tanto todas las referencias fueran actualizadas y reconocidas. Finalmente, los mensajes que estaban saliendo de los procesos que referenciaban al objeto migrante eran contenidos por su núcleo hasta que terminara la migración.

2.4. Prueba y actualización

Antes de que se efectúe una migración, el cliente verifica si el servidor realmente se encuentra en la localidad esperada. Si el servidor está ausente, el cliente pregunta por la nueva localidad a un servicio especial de localización. Este enfoque fue utilizado por los sistemas **Emerald** [10], **SOS** [17] y **Amber** [3] para determinar si la referencia era local o no. De cierta manera, este sistema es utilizado por los sistemas distribuidos que implantan algunos mecanismos globales de nombramiento. Por ejemplo, los puertos en los micro-núcleos también utilizan este tipo de solución.

Este enfoque presenta el inconveniente de que todas las invocaciones hacen el test, inclusive aquellas que referencien un objeto que no ha migrado. Este método sólo puede ser contemplado para tests locales en el mismo sitio.

2.5. Lazos de persecución

Un lazo de persecución es una indicación dejada en el sitio origen de la migración y que apunta al sitio destino. Cuando un cliente realiza una invocación, ésta llega normalmente al sitio origen, entonces, desde allí es enviado al sitio destino. Las cadenas de lazos se incrementan según el número de sitios nuevos que se van visitando en las migraciones sucesivas. Esto crea un problema denominado "de dependencias residuales" [5] que alegoriza el hecho de que el sistema va dependiendo de los lazos y que estos lazos devienen residuos cuando ellos no vuelven a ser utilizados.

Para alcanzar un objeto migrado, es necesario recorrer toda la cadena. Aparte del problema que representa un gran número de dependencias residuales, el encadenamiento también presenta el problema de que su gestión debe ser incorporada al protocolo de migración [11]. Los enlaces deben ser procesados antes de reanudar la ejecución en el sitio destino.

La nueva localización del servidor puede ser actualizada después de la primera invocación. Tres diferentes metodologías para el manejo de lazos de persecución han sido identificadas [7]. Estos enfoques difieren en la manera cómo se utiliza la localización del servidor.

Enfoque Perezoso: Los enlaces son actualizados sólo en el momento en el que ocurre una invocación. No obstante, una invocación repetida tendrá que cruzar toda la cadena de enlaces.

Enfoque del Corto Circuito: En el momento de una invocación, la localidad actual del servidor se envía al cliente incluida en el mensaje de respuesta. Las próximas invocaciones del mismo cliente son dirigidas directamente a la última localidad conocida del servidor.

Enfoque de Compresión de la Cadena: En el momento de la invocación, todos los enlaces que componen a una cadena son actualizados con la localidad actual del servidor. De esta manera, otras cadenas que hacen referencia a la cadena actualizada acortan su camino al servidor provocando una compresión.

Emerald [10] utiliza un enfoque perezoso donde los enlaces son recolectados localmente cuando los objetos se tornan inaccesibles. Cuando una invocación falla, un protocolo de localización es utilizado y los lazos son restaurados. El sistema **Guide 2** [4] usa el encadenamiento de una manera similar a **Emerald** .

Demos/MP [14] especificó un enfoque de corto circuito para la actualización de los clientes. Sin embargo, este enfoque nunca fue implementado. **Demos/MP** fue el primer sistema que utilizó lazos de persecución para encadenar procesos que habían migrado.

El sistema operativo distribuido **Galaxy** [20] utiliza identificadores únicos para todos los objetos del sistema (procesos, sitios, etc.). El acceso es transparente a la localidad, lo que facilita la actualización de los lazos. Antes del congelamiento [1] un servidor de mensajes es advertido de la nueva localidad. Esto facilita la futura entrega de mensajes a los sitios destinos. Durante la migración, la tabla de identificadores del sitio origen se marca como migrante y el nuevo sitio de residencia es añadido.

El sistema **DC**$^{++}$ [16] implementó un enfoque de cadena de compresión. Para ofrecer enlaces tolerantes a fallas, se introdujo el concepto de sitio de nacimiento. Esto consiste en mantener sistemáticamente un enlace entre el servidor y el primer sitio desde donde el servidor fue llamado por primera vez.

La selección del tipo de lazo de persecución depende del número de clientes, de la frecuencia de invocación, de la frecuencia de migración de los servidores y del número de sitios. El enfoque perezoso es el más sencillo de implementar y la compresión la más difícil.

Las experiencias ganadas a través de ciertas implementaciones de migraciones recomiendan evitar el gran número de dependencias residuales generadas por el encadenamiento [5, 8, 19, 21]. Las peores consecuencias pueden ser el deterioro del performance de las invocaciones, la sensibilidad a fallas y la necesidad de un mecanismo de recolección de basura (garbage collector) para determinar los lazos que no se necesitan.

[1] "Congelamiento" es un término de la jerga de migración que consiste en detener la ejecución de un proceso de manera tal que su estado pueda ser extraído y migrado. El término es traducido del vocablo en inglés "freezing".

2.6. Cadenas de pares tronco-rama

Un enfoque que lidia con los problemas de los lazos de persecución es llamado çadenas de pares de tronco-rama (stub-scion) (pss)"[13, 18]. Una rama (scion) es equivalente a un tronco del lado del servidor. Cuando un servidor migra, un nuevo par tronco-rama se crea en el sitio origen haciendo una conexión o binding con el sitio destino de la migración. La rama ancestral del sitio origen se actualiza haciéndola apuntar al nuevo tronco. De esta manera, el nuevo par puede ser visto como una forma de lazo de persecución.

Este enfoque actualiza la nueva localidad del servidor de manera parecida a la del corto circuito. La diferencia estriba en que un tronco puede apuntar a dos ramas diferentes. A uno de los apuntadores se le denomina "débil" y al otro "fuerte". El apuntador fuerte forma parte de la cadena originada como consecuencia de las migraciones y el apuntador débil es aquél que se actualiza cuando una invocación descubre una nueva actualización. Este apuntador débil siempre está en la misma posición o más avanzado que el apuntador fuerte.

Un enfoque donde los troncos son agrupados por recolectores distribuidos es propuesto. Las ramas son recolectadas mediante un protocolo distribuido de recolección que utiliza la información provista por los recolectores locales.

Se consideraron hipótesis fuertes de fallas. Se asumió una comunicación débil donde los mensajes pueden perderse arbitrariamente, retardarse o ser entregados en mal orden.

Las cadenas pss tienen dos grandes ventajas: la primera es que éstas pueden ser utilizadas en sistemas de muy gran escala donde el número de sitios es considerablemente grande, la frecuencia de invocación y de migración son muy altas, y la cantidad de clientes y de servidores muy grande. La segunda es que ellos facilitan la transparencia de la distribución, pues la centralización está completamente ausente. Sus inconvenientes son su complejidad, la interferencia con el protocolo de migración y la carga adicional que ellos añaden al sistema distribuido.

Las cadenas pss fueron implantadas por primera vez en el sistema **Soul** [13]. La recolección de lazos fue implantada por el sistema **Larchant** [6].

2.7. Actualización de los enlaces por recuperación de mensajes perdidos

En este enfoque, un mensaje enviado a un proceso falla si el proceso ha migrado. Se inicia, entonces, un mecanismo denominado recuperación del mensaje perdido [8]. La recuperación consiste en localizar el proceso y hacerle llegar el mensaje perdido. Varias técnicas son utilizadas para detectar la falla de la pérdida del mensaje [11]:

- Expiración de un tiempo limite de transmisión (timeout).
- Respuesta de un sitio antiguo que diga que el proceso requerido no ha sido encontrado.
- Excepción emitida por el sistema de comunicaciones (localidad desconocida, puerto inexistente, etc.).

Una vez que la falla ha sido detectada, un procedimiento para ubicar el nuevo destinatario del mensaje es puesto en marcha. Las maneras de determinar la nueva localidad van desde el uso de un servidor centralizado que guarde las nuevas localidades, hasta un protocolo totalmente distribuido de búsqueda entre todos los sitios.

Esta técnica es sencilla de implementar y no interfiere con el protocolo de migración [11]. El gran número de dependencias residuales es suprimido. Empero, es más difícil de adaptar a los sistemas de gran escala. Si un enfoque centralizado es utilizado, el desempeño se degradará conforme la cantidad de clientes que concurrentemente pregunten por la nueva localidad. La ubicación del servidor central dentro de la red también incide en el desempeño,

En el sistema **Amoeba** [22], las invocaciones que llegan al sitio origen son rechazadas. Si llegan durante la migración, un mensaje *"migración en proceso"* es respondido. Luego de la migración, la respuesta deviene: *"el proceso no esta aquí"*. En ambos casos un procedimiento de recuperación de mensajes es activado.

Para el prototipo de sistema a objetos **COOL-LDM** [11], se diseñó una variante denominada "actualización bajo excepción". Esta variante consiste en iniciar la localización de un objeto cuando ocurre una excepción del sistema de comunicación. **COOL-LDM** fue implementado sobre el sistema operativo **Chorus** [15].

2.8. Discusión

La telefonía celular presenta similaridades con los objetos móviles. Al igual que un objeto móvil, un teléfono celular cambia de posición dinámicamente; siendo entonces necesario un mecanismo que logre ubicar un celular.

Hay diferencias importantes entre los objetos y los celulares:

1. Cada celular dispone de un procesador que le permite enviar periódicamente su ubicación actual. Virtualmente, es posible incorporar esta funcionalidad a un objeto, pero ello conllevaría costes prohibitivos y posiblemente innecesarios.

2. Un celular puede ser invocado mientras éste se mueve, un objeto móvil no.

3. A causa del medio de transmisión, los celulares pueden recibir difusiones sin costes prohibitivos. Esto es posible en un sistema a objetos en detrimento del rendimiento.

4. Una vez que un objeto migre, es necesario que su nueva ubicación sea reportada solo una vez. Contrariamente, los teléfonos celulares deben estar enviando periódicamente su ubicación.

Los métodos de actualización cándidos están basados en la difusión. Todos los tipos de difusión existentes presentan el inconveniente de sobrecargar la red. El único uso de estos métodos está restringido a sistemas de pequeña escala donde hay pocos sitios, pocos objetos, y el movimiento de objetos es más bien excepcional.

El envío de mensajes a través de un mecanismo de difusión implica que todos los receptores procesarán el mensaje enviado, aun para descartarlo. Las difusiones hacen procesar en vano a quienes no están interesados en el mensaje.

Un mecanismo de difusión puede ser implementado de diversas maneras en una red. La implantación puede ir desde enviar el mensaje a cada miembro del grupo de forma secuencial, hasta los sofisticados mecanismos de difusión que poseen la mayoría de redes de área local. De cualquier manera, representa un sobrecargo en el tráfico de mensajes.

La difusión es el método a utilizar cuando otros métodos fallan en ubicar un objeto. Este método debe utilizarse como último recurso.

La prueba y la actualización sólo es aplicable en situaciones locales; es decir, cuando la referencia y el objeto se encuentren en el mismo sitio. En añadidura, la prueba debe ser ejecutada aun si un objeto no ha migrado.

Los lazos de persecución, tal como fueron presentados en § 2.5, presentan dos inconvenientes importantes. El primero es la presencia de dependencias residuales que degradan el desempeño y aumentan la sensibilidad a fallas conforme crece la cadena de lazos. El segundo inconveniente es la necesidad de implementar un mecanismo de recolección de basura (garbage collector).

Algunas técnicas han sido desarrolladas para paliar los diversos inconvenientes de los lazos de persecución, particularmente las cadenas tronco-rama. Estas técnicas son bastante prometedoras y actualmente están siendo probadas en sistemas persistentes de muy larga escala [13, 18]. A pesar de sus bondades, estas técnicas son difíciles de implementar, pues subyacen en algoritmos distribuidos considerablemente complejos para recolectar pares tronco-rama.

Todos los mecanismos de actualización basados en lazos de persecución tienen la desventaja de que deben ser considerados por el protocolo de migración. El problema estriba en que el protocolo de migración no puede reanudar la ejecución del objeto en el sitio destino hasta que no se haya creado el lazo de persecución. Esto puede ser un problema serio si la latencia de la migración es un factor prioritario.

El tamaño del mensaje de invocación es una consideración importante en los lazos de persecución. Mientras más grande sea el mensaje, más paquetes de red son necesarios. Para poder reenviar una invocación a través de un lazo, es necesario que todos los paquetes que compongan el mensaje de invocación hayan arribado. La razón de esto es que los mensajes de invocación son implementados con protocolos de transporte punto a punto IPC/RPC diseñados para un cliente y un servidor. Según las revisiones bibliográficas, la posibilidad de considerar un lazo de persecución en un protocolo de transporte aún no ha sido reportada. Posiblemente, hasta el momento de la revisión, jamás haya sido implementado un protocolo IPC que comience a reenviar paquetes, por el lazo de persecución, antes de que arriben el resto de los paquetes que componen el mensaje.

El último de los enfoques presentados fue la actualización por recuperación de mensajes perdidos. Según el sistema, la detección de pérdida de mensaje puede acarrear uno o dos mensajes de red. Si la detección de pérdida está basada en tiempos de expiración (timeouts), entonces se utiliza un sólo mensaje, pero también puede perderse una cantidad considerable de tiempo relacionada con el tiempo de expiración. La otra manera de detectar pérdida es una respuesta inmediata del destinatario del mensaje. Este es el caso de protocolos orientados a puertos, tales como las implantaciones IPC de los micro-núcleos y los protocolos basados en IP. En estos protocolos, un mensaje se pierde porque un nombre de puerto es inválido.

A nivel de transporte, la recuperación por mensajes perdidos no acarrea costes ligados a mensajes que ocupen varios paquetes de red. La pérdida es detectada al arribo del primer paquete. Bajo esta consideración, la actualización por recuperación de mensajes perdidos es menos costosa que el uso de los lazos de persecución. Resta ver cómo se podría implementar una actualización por mensajes perdidos en sistemas de gran escala. Puesto que la direcciones IP pueden ser interpretadas como puertos y que Internet ha devenido una red de gran escala, intuimos que tal implementación es factible.

2.9. Conclusión

En esta sección realizamos un recorrido por las diferentes técnicas de actualización. Luego planteamos una discusión analítica acerca las bondades e inconvenientes de cada una. En la próxima sección, presentaremos una propuesta para un mecanismo de actualización basado en una variante de recuperación de mensajes perdidos denominada "actualización por fracaso".

Capítulo 3

Planteamiento del problema

3.1. Fundamentos

En las siguientes subsecciones utilizaremos regularmente algunos términos comunes en un sistema distribuido a objetos.

Una **invocación** es el envío de un mensaje a un objeto cualquiera, en donde se especifica el método a ejecutar y sus parámetros. Este mensaje genera otro mensaje de respuesta por parte del objeto invocado.

Para poder ejecutar una invocación, hace falta tener una referencia del objeto. Una **referencia** , conocida también como medio binding, es la tupla formada por un identificador del sitio de residencia del objeto, un identificador del objeto y una estampilla lógica de tiempo; ésta última tiene la propiedad de darle unicidad a cada referencia, aún cuando se trate del mismo objeto.

Un objeto que debe responder a una invocación obtiene la información del cliente a través de un binding. Se denomina **binding** a la tupla formada por el identificador del proceso origen, el identificador del sitio desde donde se referencia al objeto y una referencia a un objeto.

3.1.1. Transporte de mensajes

Asumiremos un sistema de transporte punto a punto orientado a IPC/RPC, que en adelante llamaremos IPC. Protocolos de este tipo son comunes para sistemas y aplicaciones cliente-servidor y objetos. Podemos encontrar ejemplos en: ipcD [12], T/TCP (TCP Transaccional) [2] y x-kernel [9], entre otros. El IPC que usaremos utiliza un puerto opaco que es traducido a una dirección IP, un identificador de proceso y un identificador unívocamente universal basado en el tiempo. Se garantiza, entonces, que un puerto jamás será duplicado.

Cada sitio ejecuta un demonio IPC encargado de implantar dos operaciones básicas de pase de mensajes. La primera operación implanta el envío del mensaje de requerimiento y la recepción del mensaje de respuesta. Esta es una operación asíncrona en el sentido de que el remitente no se bloquea hasta la recepción de la respuesta. Para ello, las operaciones retornan un identificador único de mensaje, que corresponde de manera unívoca a cada par de mensajes requerimiento-respuesta. Con ello, el cliente del IPC puede administrar adecuadamente el envió y recepción de mensajes RPC. El protocolo de esta operación fue diseñado para implementar un RPC y garantiza una semántica "exactamente una vez".

La segunda operación implementa el envío fiable de un mensaje asíncrono sin respuesta.

3.1.2. Comunicación entre procesos y objetos

Para el direccionamiento de procesos y objetos, se asumen identificadores únicos globales administrados por el sistema IPC.

Un proceso maneja uno o más puertos IPC que contienen un identificador único. De esta forma, un proceso es unívocamente distinguido por el sistema IPC.

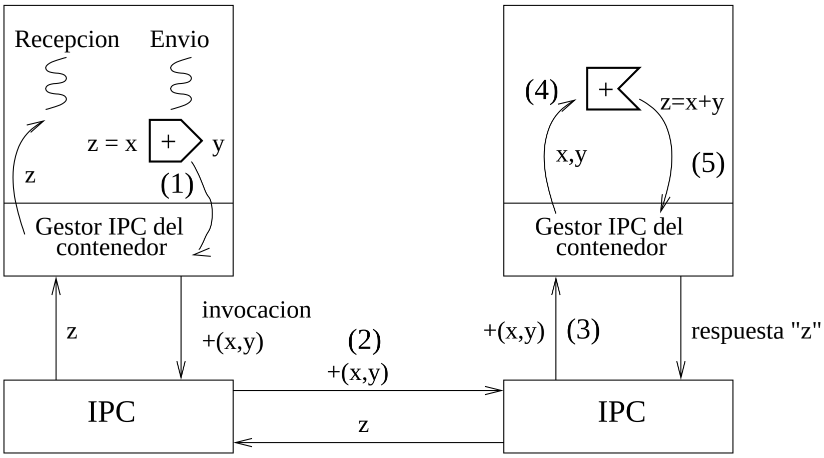

Figura 3.1: Estructura de un IPC en $\mathcal{ALEPH}$

La figura 3.1 esquematiza el proceso de invocación. Cuando una referencia efectúa una invocación (1), se envía un mensaje fiable IPC hacia el puerto del proceso destinatario (2). El sistema IPC garantiza la entrega del requerimiento al proceso. Al arribo del mensaje (3), el ejecutivo del proceso destino verifica la existencia del objeto y del método invocado. Si el objeto existe, entonces el ejecutivo llama a la implementación del objeto (4) quien, a su vez, llama al objeto. Posiblemente, el objeto invocado generará un mensaje de respuesta (5). De lo contrario, el ejecutivo responde que el objeto no ha sido encontrado.

3.1.3. Servicio abstracto de localización

El objeto de este trabajo es implantar un servicio de localización. En concreto, este servicio debe satisfacer dos servicios abstractos:

- **Registro de objeto o de proceso:** Todos los objetos y procesos del sistema deben registrarse en el servicio de localización. Del mismo modo, todo cambio de dirección de algún proceso u objeto debe ser anunciado a éste servicio.
- **Búsqueda de objeto o proceso:** El servicio de localización es capaz de procesar solicitudes de búsqueda de objeto.

3.1.4. Migración de objetos

Para enunciar este concepto, asumimos migración de objetos entre procesos remotos. La migración de objetos entre procesos locales es similar.

Antes de migrar el objeto, se establece un protocolo de negociación entre los procesos que intervienen. Si la migración es posible, entonces se copia el objeto hacia el proceso destino. Luego, se registra la nueva localidad en un servicio de localización. El identificador único del objeto nunca es modificado. Finalmente, el objeto es destruido en el proceso origen.

Invocaciones posteriores dirigidas hacia el objeto migrado en el sitio origen son retenidas por el servicio de localización; es decir, no se efectúa la llamada al objeto. Al culminar la migración, estas invocaciones son rechazadas con el mensaje "*el objeto ha migrado al sitio x*", si se dispone de la nueva dirección o, con el mensaje "*objeto inexistente*", si la dirección se desconoce.

3.1.5. Migración de procesos

Antes de que un proceso migre, se establece un protocolo de negociación que determine si el sitio puede albergarlo. Si esto es posible, entonces una imagen del proceso es creada en el sitio destino bajo un nuevo puerto IPC. Cuando la imagen ha sido creada, el sitio origen registra en el servicio de localización las nuevas coordenadas del proceso y todos sus objetos. Finalmente, la ejecución es reanudada en el sitio destino y la imagen del proceso en el sitio origen, junto con su puerto, son destruidos.

Las invocaciones dirigidas a cualesquiera de los objetos residentes en el proceso migrado son tratadas de la misma forma de cuando un solo objeto migra (3.1.4).

3.2. Actualización por fracaso de invocación

De las técnicas de actualización expuestas en el capítulo 2, la única alternativa que no presenta costes prohibitivos y escalable para la localización distribuida, es la de los lazos de persecución. Sin embargo, presenta el problema de las dependencias residuales y sensibilidad a fallas conforme crece la cadena de lazos. Esta sección se presenta una comparación más amplia de los lazos de persecución con el fracaso de invocación.

También formalizaremos la técnica fundamental de nuestro servicio de localización: la "actualización por fracaso de invocación" o, en corto, "actualización por fracaso".

Cuando un objeto o proceso migra, todas sus referencias se vuelven inválidas. Entonces, cualquier invocación desde alguna de estas referencias fracasará. En todo caso, el fracaso se hará saber a través de un mensaje de respuesta.

La actualización por fracaso requiere que, al momento de una migración, la nueva localidad sea notificada al servicio de localización.

La figura 3.2 ilustra el flujo de actividades a ejecutar cuando una invocación fracasa:

1. Un objeto O_1 migra desde el sitio 2 hacia el sitio 1. La nueva localidad es transmitida al servicio de localización.

2. Una referencia R_1, devenida inválida, efectúa una invocación. Esta invocación fracasa porque el objeto migró.

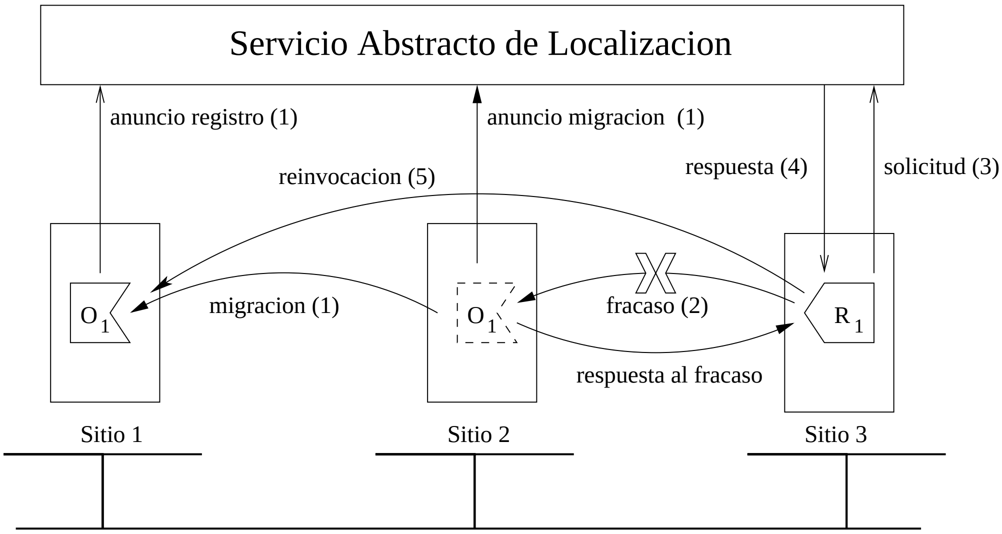

Figura 3.2: Flujo de actividades efectuadas en una invocación fracasada

3. El proceso que contiene la referencia emite una solicitud al servicio de localización solicitándole la nueva dirección del objeto.
4. El servicio de localización encuentra el objeto y retorna su dirección al proceso solicitante. Luego, la referencia R_1 es actualizada.
5. Actualizada R_1, la invocación es efectuada de nuevo.

Cualquier invocación fracasada provocará una solicitud al servicio de localización preguntando la nueva localidad del objeto. El servicio de localización se encarga, entonces, de encontrar el objeto y de notificar su dirección al solicitante. Luego, la referencia desde donde se hizo la invocación es actualizada con la nueva localidad. Una vez realizada la actualización, la invocación es repetida.

3.2.1. Mensajes de respuesta en el fracaso de una invocación

Existen dos escenarios posibles cuando se efectúa una invocación. Primero, la invocación puede tener éxito, el método remoto es ejecutado y una respuesta es recibida. Segundo, la invocación puede fracasar. El fracaso de la invocación puede detectarse mediante los siguientes tipos de respuesta:

1. "*El objeto no fue encontrado*": Con esta respuesta se concluye que el sitio destino de la invocación no conoce el objeto. Existe todavía la posibilidad de que se encuentre en cualquier otro sitio del sistema.
2. "*Existe una referencia más reciente*": En este caso, el sitio destino sabe que el objeto no está presente, pero conoce otro sitio donde el objeto puede encontrarse. Una respuesta de este tipo muestra una referencia más reciente. Esto sugiere la posibilidad de reinvocar con esta nueva dirección.

3. *"El objeto está eliminado"*: Cuando se obtiene esta respuesta no es posible ejecutar la invocación, pues el objeto ha sido eliminado del sistema.
4. El sistema de comunicación reporta un error; por ejemplo, puerto inexistente. Esta es la situación cuando un proceso migra con todos sus objetos. En el sitio destino de la migración, el proceso crea un nuevo puerto. Puesto que éste es unívoco, el sistema de comunicación será incapaz de procesar mensajes en el sitio origen de la migración, pues el puerto destino ya no existe.

En adelante estos tipos de mensajes serán conocidos como: mensaje de fracaso de tipo 1, 2, 3 y 4, respectivamente.

3.2.2. Duración de la invocación

De manera general y simplificada, la duración total de una invocación exitosa puede caracterizarse de la siguiente manera:

$$t_i = t_m(k_{in}) + t_m(k_{out}) \quad , \tag{3.1}$$

donde:

- k_{in} es el tamaño del mensaje de invocación.
- k_{out} es el tamaño del mensaje de respuesta de invocación.
- t_m es una función que caracteriza la duración de un mensaje punto a punto en función del tamaño del mensaje.

3.2.2.1. Duración de invocación con lazos de persecución

Por su naturaleza distribuida, el uso de lazos de persecución es el enfoque de actualización más popular. Si han ocurrido n migraciones, entonces, según (3.1), el costo de una invocación puede expresarse de varias maneras:

$$t_i = \begin{cases} (n+1)t_m(k_{in}) + t_m(k_{out}) & \text{Si la respuesta es enviada desde el sitio destino} \\ (n+1)t_m(k_{in}) + (n+1)t_m(k_{out}) & \text{Si la respuesta regresa por la cadena de lazos} \end{cases} \tag{3.2}$$

A efectos comparativos, favoreceremos la primera expresión, pues ofrece menos duración.

3.2.2.2. Duración de invocación con fracaso

En su forma abstracta, la actualización por fracaso de invocación requiere cuatro mensajes adicionales para actualizar una referencia. El primer mensaje es la invocación misma que fracasa. El segundo es la respuesta de fracaso. El tercero es el requerimiento emitido al localizador. Finalmente, el cuarto es la respuesta del localizador con la nueva dirección. Podemos, pues, caracterizar la duración de una invocación de la siguiente forma:

$$t_i = \underbrace{t_m(k'_{in})}_{\text{Invocación que fracasa}} + \underbrace{t_m(k_{fail})}_{\text{Anuncio de fracaso}} + \underbrace{t_m(k_{req})}_{\text{Solicitud localizador}} + \underbrace{t_m(k_{rep})}_{\text{Respuesta localizador}} + \underbrace{t_m(k_{in}) + t_m(k_{out})}_{\text{Invocación exitosa}} \tag{3.3}$$

k'_{in} es el tamaño del mensaje de invocación que fracasa. Asumimos $k'_{in} \leq k_{in}$ porque, según el sistema de comunicación, el fracaso de la invocación puede detectarse antes de que arribe la totalidad del mensaje de invocación. k_{fail} es el tamaño de un mensaje de cualquiera de los tipos presentados en § 3.2.1 que indica el fracaso de la invocación. k_{req} y k_{rep} son, respectivamente, los tamaños de los mensajes de requerimiento y respuesta del localizador.

k_{fail}, k_{req} y k_{rep} son mensajes pequeños que no dependen del tamaño de la invocación. Por simplicidad, podemos asumir $t_m(k_{fail}) = t_m(k_{req}) = t_m(k_{rep}) = T_c$, donde T_c es la duración de un mensaje pequeño en condiciones de carga relajadas. Igualmente, por simplificar, asumiremos $k'_{in} = k_{in}$; es decir, el peor caso. A partir de estas simplificaciones podemos escribir (3.3) como sigue:

$$t_i = 3T_c + 2t_m(k_{in}) + t_m(k_{out}) \tag{3.4}$$

3.2.2.3. Comparación

Podemos vislumbrar una comparación del desempeño de la invocación ante una migración mediante las expresiones (3.2) y (3.4). En lo que concierne la invocación, el fracaso de la invocación tendrá un mejor desempeño cuando (3.4) < (3.2); es decir:

$$\begin{aligned} 3T_c + 2t_m(k_{in}) + t_m(k_{out}) &< (n+1)t_m(k_{in}) + t_m(k_{out}) \Longrightarrow \\ (n-1)t_m(k_{in}) - 3T_c &> 0, \quad n \geq 2 \end{aligned} \tag{3.5}$$

Claramente, hasta en un máximo de 2 migraciones, la invocación se desempeña mejor con un lazo de persecución. Esto es normal, pues a nivel de mensajes de invocación, los dos enfoques tienen el mismo número de mensajes. La diferencia estriba en que el fracaso le añade los mensajes al localizador.

Para un número de migraciones mayor que dos, el desempeño depende de la forma de la función t_m, la cual depende del sistema de comunicación que se utilice. Como medio de comparación, utilizaremos datos experimentales tomados de un sistema IPC desarrollado en el CEMISID [12]. El sistema en cuestión implanta un protocolo IPC, completamente fiable, con semántica de a lo más una vez, sobre paquetes UDP. Medidas reportadas en [12] demuestran que este protocolo es superior a la alternativa confiable TCP/IP.

La figura 3.3 ilustra la expresión (3.5) para varios valores de n y valores experimentales de t_m en una red de área local Ethernet de 10 Mbits/sec. Se consideró un valor de $T_c = 2,8$ msec, que es el desempeño del sistema IPC para un mensaje de 64 bytes; longitud más que suficiente para cualquier mensaje al localizador. Podemos apreciar que para dos migraciones y un tamaño de mensaje de invocación menor o igual a los 512 bytes, los lazos de persecución aún son mejores que la actualización por fracaso.

Los mensajes de invocación contienen información de control y los parámetros de entrada de un método remoto. En este sentido, 512 bytes es una cantidad aceptable para muchas de las aplicaciones existentes que no utilicen arreglos o secuencias. Así pues, podemos intuir que los lazos de persecución aún son buenos para dos migraciones.

Para tres migraciones, los lazos todavía son buenos si $k_{in} < 128$ bytes; tamaño factible pero menos probable, para un mensaje de invocación. Para $n = 4$, la actualización por fracaso de invocación es superior que los lazos de invocación para todo k_{in}.

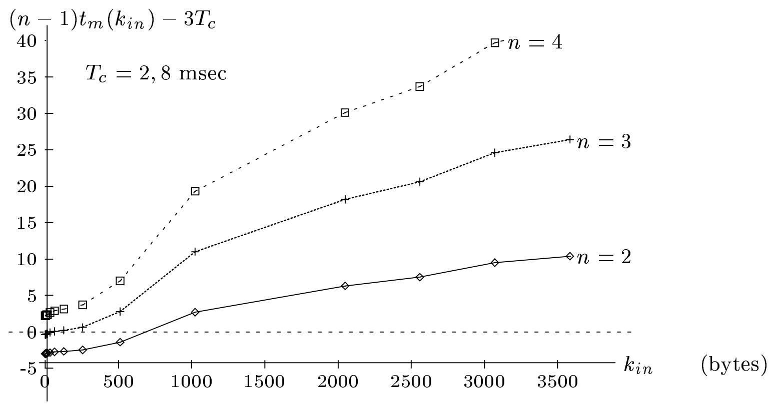

Figura 3.3: Valores estimados de (3.5) en una red Ethernet de 10 Mbits/sec sobre el sistema IPC de [12]

Este análisis supone que el localizador es un servidor centralizado que se encuentra en condiciones de carga relajadas. Por supuesto, si la frecuencia de migración es muy alta, el valor de T_c aumentaría radicalmente y nuestro análisis devendría inválido. Además, el análisis también predispone baja escala.

Bajo los paradigmas actuales de desarrollo de sistemas distribuidos, parecería posible asumir que una migración es mucho menos frecuente que una invocación. Así pues, es plausible suponer condiciones relajadas para un servidor central de localización.

Si la escala del sistema aumenta, sea en número de sitios, cantidad de objetos, en frecuencia de migraciones, etc., entonces una solución de localización distribuida, basada en el fracaso de invocación, que haga uso intensivo de caches, podría eliminar el problema de sobrecarga al servidor de localización.

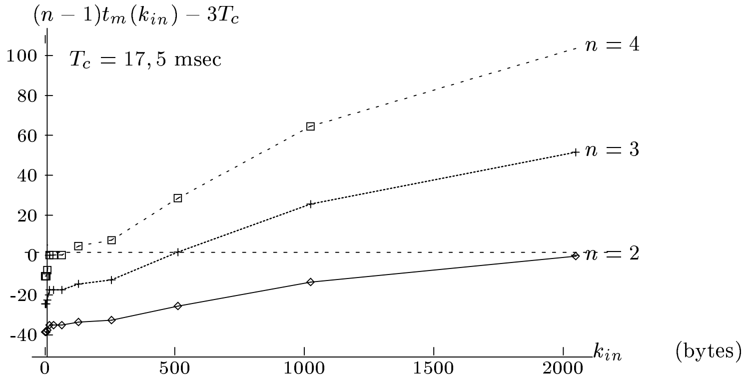

Figura 3.4: Valores estimados de (3.5) en una red metropolitana IP con tres enrutadores en el sistema IPC de [12]

Podría pensarse que para una red de menor latencia y ancho de banda, nuestro análisis podría cambiar. La figura 3.4 muestra la expresión (3.5) para valores de t_m medidos sobre una red IP de área metropolitana [12].

De la figura 3.4 podemos apreciar que nuestro análisis aún sería válido para redes de menor latencia y ancho de banda. La figura 3.4 asume $T_c = 17{,}5$ msec, que es la latencia reportada para un mensaje de 64 bytes [12]. Sin embargo, esto no es necesariamente correcto, pues T_c dependería del lado de la red donde se encuentre el localizador central y la referencia inválida.

3.2.2.4. Percepción de la migración

Tanto en los lazos de persecución como en la actualización por fracaso, la acción de actualizar es iniciada por una invocación. Desde la perspectiva del sitio invocante, el tiempo en que se tarde en efectuar la actualización es un coste importante. Cuanto antes se conozca la nueva localidad del objeto, más referencias pueden aprovechar este conocimiento y efectuar directamente una invocación. En este sentido, la actualización por fracaso ofrece una latencia menor que los lazos de persecución. En otras palabras, con actualización por fracaso, el sitio invocante se percata más rápido de la migración que con los lazos de persecución.

Según la variante de lazos de persecución que se use, la cantidad de sitios que se percatan de la migración puede ser mayor. En efecto, con el enfoque de compresión de cadena explicado en § 2.5, todos los sitios incluidos en una cadena de lazos pueden informarse de la migración una vez que se completa la invocación. En este sentido, los lazos de persecución favorecen la cantidad de sitios que se percatan de una migración.

Un localizador distribuido, que redunde información de migración a través de los sitios, podría cooperar para mantener información reciente de las últimas migraciones. De este modo, podríamos alcanzar una proporción de sitios que se percaten de la migración equivalente a la de los lazos de persecución.

3.2.2.5. Incidencia sobre el protocolo de migración

Un lazo de persecución afecta la latencia de la migración, pues la migración no puede ser efectiva hasta no asegurar la presencia del lazo en el sitio origen de la migración. De lo contrario, se corre el riesgo de que llegue una invocación que nunca podrá ser procesada. En añadidura, la implantación de la migración es más compleja, pues la creación del lazo de persecución debe ser incluida y sincronizada por el protocolo de migración.

La actualización por fracaso requiere que se le notifique la migración al localizador. Esta notificación puede ser realizada de forma asíncrona y temprana respecto al protocolo de migración. Esto hace más sencillo el protocolo y su implantación.

Si una invocación arriba durante la migración, entonces, con buena suerte, la nueva dirección del objeto habrá arribado al localizador antes que la solicitud de búsqueda. De todos modos, la invocación puede ser programada para efectuar varios intentos. Así, en el peor de los casos, el objeto podrá ser localizado después de varios intentos.

3.2.2.6. Sensibilidad a fallas

La tolerancia a fallas es uno de los principales aspectos de todo sistema distribuido. Bajo esta perspectiva, hay dos factores a considerar: la detección de la falla y la acción para enmendarla.

Una cadena de lazos puede romperse si algo malo sucede en cualquiera de los sitios de la cadena. En este caso, la acción más común para recuperar la falla consiste en realizar difusiones de actualización [10]. Lamentablemente, esta solución es excesivamente costosa, pues, aparte los inconvenientes ligados a la difusión, es muy difícil detectar rápidamente la pérdida de un lazo de persecución. Por esa razón, a expensas de costes mayores en espacio y tiempo, algunos enfoques de lazos de persecución más recientes están basados en la redundancia de los lazos [13, 18].

La actualización por fracaso requiere la presencia de un servicio de localización. Si el servicio es centralizado y éste falla, entonces todo el sistema es comprometido. La ventaja de la centralización es que la falla puede ser detectada muy rápidamente. Además, un localizador centralizado podría utilizar técnicas de alta disponibilidad y recuperación. A pesar de estas medidas, es muy difícil tratar con la escala. De hecho, un servicio centralizado sobrecargado puede dar una falsa impresión de falla y provocar acciones vanas para enmendarla.

Si el servicio de localización es distribuido, entonces es posible redundar información de localización a través de componentes repartidos. Si uno de los componentes falla, entonces otro más puede tomar el relevo. La detección de falla es mejor que en el caso central.

3.2.2.7. Costo de la actualización

La presencia de movilidad añade costes adicionales al desempeño de un sistema distribuido. En la sección § 3.2.2 aprehendimos los costes de red que los lazos de persecución acarrean a la invocación. Aparte de estos costes, la invocación tiene un sobrecoste de procesamiento. Cada CPU donde se encuentre un lazo gastará algunos ciclos adicionales en procesar el lazo.

Con la actualización por fracaso de invocación, la estructura de un despachador de objetos es la misma que con los lazos de persecución e, inclusive, que la de un sistema sin migración. Independientemente de que el sistema sea estático o móvil, el despachador debe verificar si el objeto se encuentra en el sitio. Así pues, desde esta perspectiva, el único costo de procesamiento de la actualización por fracaso está dado por los mensajes enviados al servicio de localización. La actualización no acarrea, pues, costes de procesamiento sobre la invocación.

El coste más serio de los lazos de persecución es mantener un mecanismo de recolección de lazos que ya no se usen más. Aunque soluciones modernas que usan mecanismos distribuidos de recolección no reportan estos costes [6], es evidente que hay que consumir ciclos de CPU y mensajes adicionales para efectuar la recolección.

La actualización por fracaso, centralizada o distribuida, no requiere recolección.

3.2.2.8. Localización distribuida mediante fracaso de invocación

En las subsecciones anteriores hemos destacado las ventajas y desventajas de la actualización por fracaso respecto a los lazos de persecución. Estamos listos, entonces, para formular las bases del servicio de localización distribuido.

Nuestro servicio estará basado en servidores de localización. Cada servidor mantiene una “tabla propietaria” que almacena localidades de objetos. Cuando un objeto migra, sus coordenadas pueden ser eliminadas de la tabla propietaria de un servidor y guardadas en la tabla propietaria de otro. En todo caso, la tabla propietaria ofrece información correcta de la localización de un objeto.

Resultados de búsquedas previas pueden guardarse en caches. De este modo, requerimientos de búsqueda pueden ser satisfechos sin necesidad de encontrar el servidor propietario del objeto. Para que el uso de caches sea efectivo, es necesario:

- Que las localidades de los objetos redunden a través de los caches de los diversos servidores.
- Que la información contenida en el cache sea vigente; es decir, que no corresponda a búsqueda de objetos que migraron de nuevo.

La información contenida en los caches es redundante a través de los servidores del sistema. Esta redundancia ofrece un soporte para el procesamiento de fallas.

Requerimos, entonces, de políticas que refresquen los caches con información reciente y pertinente. Además, necesitamos ser capaces de recuperar la dirección de un objeto si, por mala suerte, los caches y la tabla propietaria fallan. A tales efectos, contamos con las siguientes técnicas:

1. **Prefetching**

   Cuando ocurre una migración de objeto, algunos servidores pueden ser notificados de la nueva localidad. De este modo, los caches son refrescados con la información más reciente.

2. **Piggybacking en invocaciones**

   Puesto que la invocación es una actividad muy frecuente y fundamental en un sistema a objetos, podemos aprovechar los mensajes de invocación para intercambiar información entre los servidores de localización.

   Una indicación de localidad implica información de corto tamaño. Así pues, el impacto de este intercambio no debería ser muy grande. En añadidura, los protocolos de red y de transporte actuales trabajan en función de un grano de mensaje. La latencia y ancho de banda de tales protocolos no es exactamente lineal. Para tamaños de mensajes aproximados, pero no iguales, el desempeño del protocolo es el mismo.

3. **Protocolo de recuperación**

   Si a pesar de las técnicas anteriores no es posible recuperar una localidad, entonces es necesario encontrar el servidor propietario. Para ello, como último recurso, establecemos un protocolo que efectúa difusiones en cascada, desde la de menor hasta la de mayor costo. En líneas generales, tal protocolo consiste en:

   a) Se envía un mensaje no fiable preguntando por el localizador propietario del objeto, quien debería responder con un mensaje fiable.

   b) Si el paso anterior fracasa, se difunde un mensaje no fiable a todos los servidores. Cualquier servidor que encuentre información sobre el objeto responde con un mensaje fiable.

c) Si el paso anterior fracasa, se envía un mensaje fiable preguntando por el localizador propietario del objeto, quien debería manifestarse con un mensaje fiable de vuelta.

d) Si el paso anterior fracasa, entonces se difunde un mensaje fiable ordenando a todos los servidores reconstruir su tabla propietaria. Consecuentemente, si el objeto existe, el localizador propietario responde un mensaje fiable con la localidad.

e) Si el paso anterior fracasa, el objeto es declarado inexistente.

3.3. Arquitectura del servicio de localización

El enfoque propuesto del servicio de localización es enteramente distribuido. En cada sitio existe un demonio de localización que registra todos sus procesos y objetos. En adelante, este demonio será denominado "localizador".

También existe una parte ejecutiva de localización en cada proceso. Este ejecutivo es denominado "localizador proceso".

Para facilitar el discurso, es conveniente enunciar algunas definiciones. Dado un sitio x, definimos:

Invocación entrante: Una invocación proveniente de un sitio remoto que arriba al sitio x.

Invocación saliente: Una invocación emitida desde el sitio x hacia un objeto remoto.

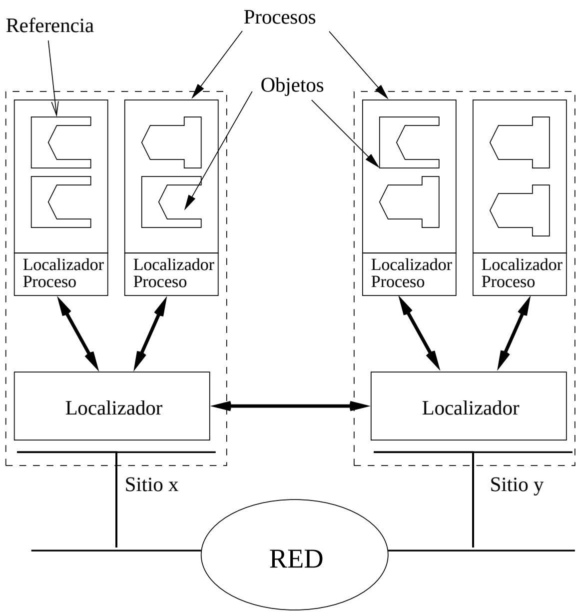

Figura 3.5: Arquitectura con el sistema de localización

La figura 3.5 ilustra la arquitectura distribuida del servicio de localización. En principio, los localizadores interactúan de forma cooperativa durante la migración, durante las invocaciones o durante solicitudes de indagación y actualización por algún ente interesado en buscar un objeto.

3.3.1. Localizador

Las características básicas de un localizador son:

- Conoce las coordenadas de localización de todos los procesos y objetos residentes en el sitio.
- Almacena y mantiene la información necesaria para la localización distribuida de objetos.
- Responde a solicitudes de búsqueda locales y distribuidas, pertinentes a la localización y control distribuido.
- Interviene las invocaciones entrantes y salientes. Ambas invocaciones pueden ser leídas, modificadas o rechazadas.
- Dirige el flujo de las invocaciones entrantes y salientes.

El localizador conoce los procesos y los objetos mediante la tabla propietaria. Esta tabla guarda toda la información necesaria para localizar los objetos y procesos residentes en el sitio. Cuando un proceso u objeto migra, su registro en la tabla propietaria "migra" hacia la tabla propietaria del localizador del sitio destino de la migración.

Dado un objeto residente sobre un sitio x, el localizador del sitio x es denominado su "localizador propietario".

3.3.2. Localizador Proceso

Las características básicas de un localizador proceso son:

- Mantener información de los objetos residentes en el proceso.
- Validar y despachar las invocaciones entrantes y salientes.
- Cooperar con el protocolo de migración de objetos.

En resumen, el localizador proceso es la parte ejecutiva de un proceso que contiene objetos. Este ejecutivo gestiona las invocaciones a objetos del proceso y efectúa llamadas al demonio localizador. El localizador proceso es implantado por una biblioteca a encadenar a todo proceso que desee manipular objetos.

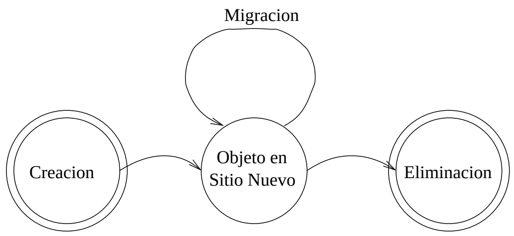

Figura 3.6: Máquina de estados del ciclo de vida de un objeto

3.4. Caching

La manera más efectiva de amortizar los costes inherentes al fracaso en la invocación es el caching de búsquedas previas. Diversos tipos de caches son habilitados en cada localizador para almacenar cálculos pertinentes a una localización. La idea es guardar un conjunto de búsquedas previas y eventualmente utilizarlas en futuras búsquedas.

Para todos los tipos de caches que trataremos, se asume la existencia de una política de reemplazo a usar cuando el cache esté lleno. La entrada en cada uno de los caches es débil en el sentido de que puede ser incorrecta. Esto sucede, por ejemplo, si el objeto migra dos veces y el cache mantiene el registro de la primera migración.

Dentro de nuestro sistema, la conducta de un objeto se abstrae en el diagrama de estados presentado en la figura 3.6. Luego de su creación, el objeto puede estar en constante movimiento por todo el espacio de máquinas, hasta que es eliminado del sistema, donde su identificador no vuelve a ser utilizado jamás. Esto trae ciertas consecuencias:

1. Un objeto no está limitado en su recorrido por el sistema.

2. Una referencia a un objeto es siempre potencialmente inválida, pues un objeto no tiene restricciones respecto al momento de migrar.

Siempre que se vaya a mover un objeto, las invocaciones dirigidas a éste serán retenidas por el localizador que aloja al objeto. Luego, será respondido un mensaje con la nueva localidad del objeto a los sitios origen de las invocaciones. De esta forma la invocación puede repetirse.

Asumiremos que los localizadores poseen mecanismos para enviar y recibir mensajes asíncronos confiables.

3.4.1. Cache de Migración

Cuando un objeto en un sitio x migra hacia un sitio y, éste deja su nueva localidad en el cache de migración del sitio origen.

Cuando una invocación hacia el objeto migrado arriba al sitio x, el localizador propietario busca en el cache de migración. Si una referencia más reciente es encontrada, entonces se le responde a la invocación un mensaje de fracaso del tipo 2, con la nueva dirección del objeto conocida por el cache. De no encontrarse ningún registro en este cache, podría obtener un mensaje de fracaso del tipo 1 o del tipo 3.

El acto de migrar causa una revisión y eventual actualización de todos los caches de los sitios origen y destino de la migración. Estos caches serán explicados en el resto de las secciones subsiguientes.

Un registro en este cache contiene la tritupla $< A, z, t >$. A es un identificador de objeto y z es el último sitio válido conocido donde estuvo A en el tiempo lógico t. En adelante, t es una estampilla de tiempo lógico que indica el número de migraciones del objeto.

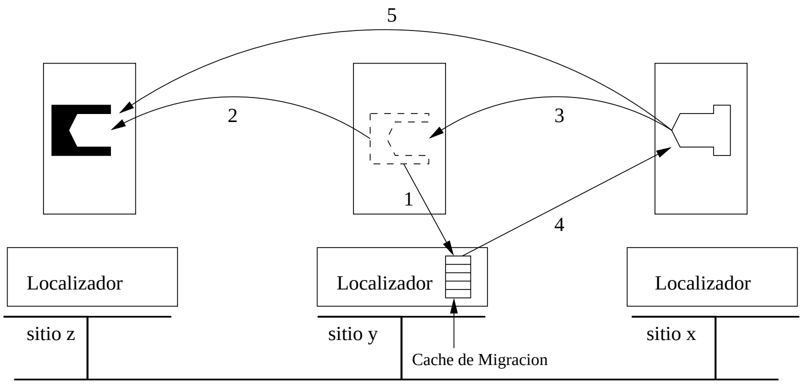

Figura 3.7: Uso del cache de migración

La figura 3.7 ilustra el uso del cache de migración. Las etiquetas numeradas representan los eventos siguientes:

1. El objeto registra su futura localidad en el cache de migración del sitio origen y.
2. El objeto migra hacia el sitio z.
3. Una referencia con la antigua dirección efectúa una invocación. La invocación arriba al antiguo sitio y.
4. El localizador busca en el cache de migración; si encuentra una entrada más reciente, entonces le responde al sitio invocante un mensaje de fracaso del tipo 2.
5. La referencia es actualizada y la invocación es reenviada hacia el sitio z.

3.4.2. Cache de Entrada

Dado un localizador, el cache de entrada guarda referencias remotas correspondientes a invocaciones entrantes (ver figura 3.8).

Un registro de éste cache contiene la tritupla $< A, x, t >$. A es un identificador del objeto invocado, x es un sitio donde existe una referencia hacia A y t es la estampilla de tiempo lógico de llegada del objeto x.

El cache de entrada puede ser utilizado para hacer prefetching en los sitios que poseen las referencias remotas. Cuando un objeto migra, algunas entradas del cache que indiquen una referencia al objeto migrante, son utilizadas para notificar a otros localizadores la nueva localidad.

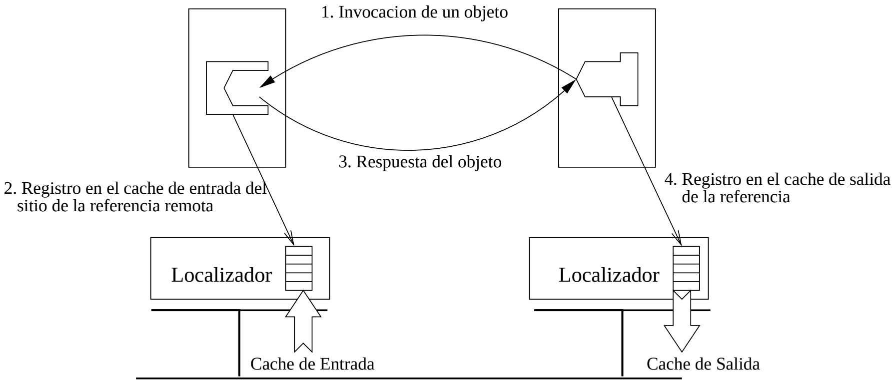

Figura 3.8: Uso de los Caches de Entrada y Salida

3.4.3. Cache de Salida

Dado un localizador, el cache de salida almacena referencias locales correspondientes a invocaciones salientes (ver figura 3.8). Toda invocación que sale de un sitio es revisada por el localizador. Si el localizador encuentra una referencia más reciente en el cache de salida, entonces la invocación es enviada hacia el sitio cuya referencia es más reciente. Si esta invocación no fracasa, entonces la referencia es actualizada con la nueva dirección.

El localizador puede recibir información acerca de nuevas localidades. Cuando un localizador recibe una nueva localidad, ésta queda actualizada en el cache de salida.

El cache de salida puede usarse para hacer prefetching de referencias locales. Periódicamente, se puede supervisar la validez de las referencias guardadas en el cache de salida. Si se detecta alguna referencia inválida, entonces, con suerte, la referencia podría actualizarse antes de que ocurra la invocación. La actualización es insertada en el cache de salida, de manera tal que las subsecuentes invocaciones salientes no fracasarán.

Un registro de este cache contiene la tritupla $< A, x, t >$. A es un identificador de objeto y x es un sitio donde residió el objeto en el tiempo lógico t. Esta tritupla representa una referencia.

3.4.4. Cache de nuevas referencias

Por alguna razón, un localizador puede llegar a conocer la nueva referencia de un objeto. Supongamos que un localizador l_x en sitio x recibe un mensaje notificándole: "*en el tiempo lógico t, el objeto A residía en el sitio y*". El localizador busca en el cache de salida una entrada para el objeto A. Si la entrada es encontrada y el tiempo lógico de la nueva referencia es más reciente que la del cache, entonces éste es actualizado con la nueva dirección.

Si en el sitio x no hay referencias al objeto A, o si el cache de salida de l_x está frío[1] respecto al objeto A, entonces, con toda certitud, el cache de salida no tendrá entradas que refieran al objeto A. Para paliar éste inconveniente, o para mantener una información

[1] En la jerga de caching, "frío" significa que: o una entrada para el objeto nunca ha sido ingresada, o que la entrada para el objeto fue extraída del cache por la política de reemplazo.

que después podría ser útil, una referencia que aún no ha sido utilizada, es decir, que no haya sido encontrada en el cache de salida, es insertada en un cache llamado “cache de nuevas referencias”.

Un registro de éste cache contiene la 3-tupla $< A, x, t >$. A es un identificador de objeto y x es un sitio donde residía el objeto A en el tiempo lógico t. Esta tupla representa una referencia.

3.4.5. Cache de eliminaciones

Cuando un objeto es eliminado del sistema, su identificador único queda registrado en este cache. Así, podrá ser utilizado para actualizar el resto de los caches.

Si un objeto se encuentra en este cache, entonces significa que ha sido eliminado de la tabla propietaria y del resto de los caches donde existiera alguna entrada.

Un registro de este cache sólo contiene el identificador único del objeto.

3.4.6. Caches e invocaciones

Una invocación saliente hacia el objeto A es procesada como sigue:

1. Buscar en el cache de eliminados. Si se encuentra una entrada en este cache, entonces se declara inexistente el objeto y el proceso de búsqueda finaliza.
2. De lo contrario, si fracasó el paso anterior, buscar una referencia en el cache de salida. Si una referencia más reciente es encontrada, entonces actualice la invocación para que apunte al sitio señalado por la referencia más reciente y finalice el proceso de búsqueda.
3. De lo contrario, si fracasó el paso anterior, buscar la entrada más reciente entre el cache de migración y en el cache de nuevas referencias.
4. Si en ninguno de los caches del paso anterior ha sido encontrada una referencia más nueva, entonces se procede a invocar. En el caso de haber encontrado una referencia más reciente, se actualiza la referencia y se procede a invocar.

Por otro lado, una invocación entrante se procesa de la manera siguiente:

1. Buscar el objeto en la tabla propietaria. Si el objeto es encontrado, entonces se procede con la invocación.
2. De lo contrario, si fracasó el paso anterior, buscar en el cache de eliminados. Si se encuentra una entrada en este cache, se responde un mensaje de fracaso del tipo 3, y el proceso de búsqueda finaliza.
3. De lo contrario, si fracasó el paso anterior, buscar una referencia en el cache de salida. Si una referencia más reciente es encontrada, entonces responda un fracaso de invocación con la referencia más reciente y finalice el proceso de búsqueda.
4. De lo contrario, si fracasó el paso anterior, buscar la entrada más reciente en el cache de migración y en el cache de nuevas referencias.

5. Si en el paso 2 o 3 se encontró una referencia más nueva del objeto, se responde un mensaje de fracaso del tipo 2. De otra forma se responde un mensaje de fracaso del tipo 1.

3.5. Prefetching

En este trabajo, prefetching es la acción de actualizar una referencia inválida antes de que ocurra una invocación. Para determinar los sitios hacia donde se puede realizar el prefetching, se utilizan los caches de los sitios origen y destino de migración.

3.5.1. Prefetching desde el sitio origen de la migración

El sitio origen de la migración puede realizar prefetching con el cache de entrada. Para ello, se buscan referencias hacia el objeto migrante y se les notifica a los sitios remotos la nueva localidad del objeto.

Todas los registros referentes al objeto migrado que están dentro del cache de entrada, son eliminados cuando la migración se hace efectiva. Esto es útil para favorecer la política de reemplazo con una entrada que con certeza está disponible.

3.5.2. Prefetching desde el sitio destino de la migración

Para que el sitio destino de la migración pueda efectuar futuros prefetching, se deben efectuar las siguientes operaciones con los caches:

Con el cache de entrada: Antes de eliminar del cache de entrada del sitio origen las referencias hacia el objeto migrante, algunos registros de este cache pueden "migrar" hacia el cache de entrada del sitio destino. De esta manera, el sitio destino puede efectuar prefetching tal como se describió en § 3.5.1.

En la figura 3.9 se observa como al migrar O desde el Sitio A hacia el Sitio B (1), también lo hacen sus referencias R_1, R_2 y R_3 (2). Luego, desde el Sitio B se puede ejecutar el prefetching (3) como se hizo en el Sitio A (4).

Con el cache de salida: Al arribo del objeto al sitio destino, se busca en el cache de salida una referencia hacia el objeto migrante. Si una entrada es encontrada en el cache, entonces ésta es actualizada.

De haber referencias en el sitio destino que apunten hacia el objeto migrante, éstas deben ser actualizadas. Esto permite interceptar una invocación y redirigirla como una invocación local.

3.5.3. Prefetching explícito con el cache de salida

Las entradas del cache de salida pueden ser periódicamente verificadas. La verificación consiste en probar si el objeto continúa en el sitio donde dice la referencia. Si alguna referencia es inválida, entonces se inician acciones de búsqueda del objeto con la esperanza de encontrarlo antes de que ocurra una invocación.

La figura 3.10 ilustra el prefetching con el cache de salida. Se observan las referencias que se encuentran en un momento cualquiera en el cache de salida de localizador del

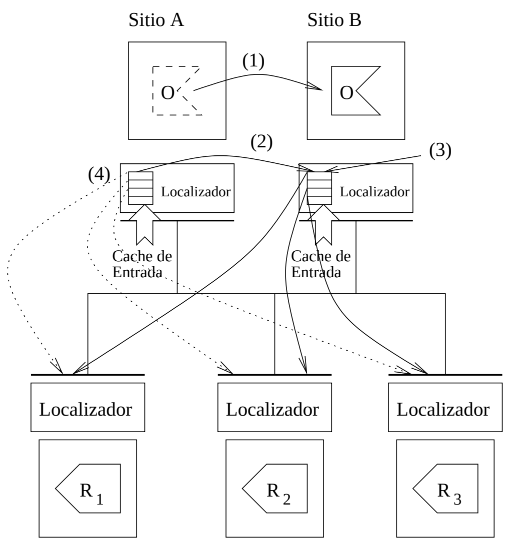

Figura 3.9: Prefetching con el Cache de Entrada

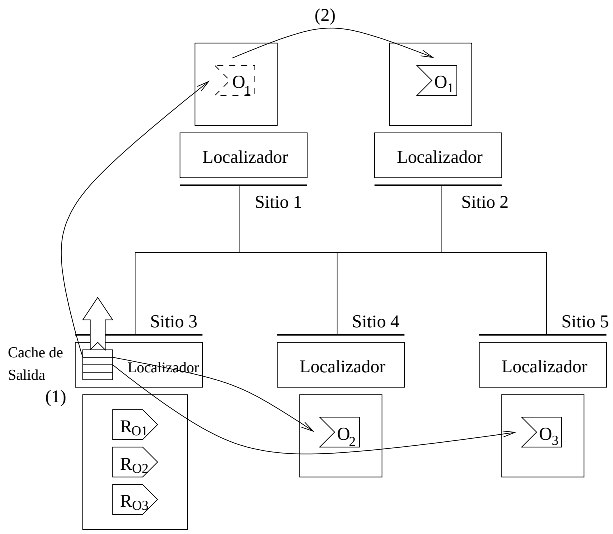

Figura 3.10: Prefetching con el Cache de Salida

sitio 3 (1). Al hacer la verificación de las referencias, se descubrirá que R_{O_1} es inválida, pues O_1 ha migrado (2). Luego, serán emprendidas acciones de búsqueda con el objeto de actualizarla.

3.6. Invocación como vehículo de localización

Esta técnica se sirve de la actividad de invocación como vehículo de localización. Las secciones subsiguientes desarrollarán los conceptos necesarios para comprender los mecanismos de localización basados en ésta técnica. En el argot de redes el acto de aprovechar mensajes explícitos para cargar otros mensajes clandestinamente se denomina "*piggybacking*".

3.6.1. Mensajes explícitos y *piggybacks*

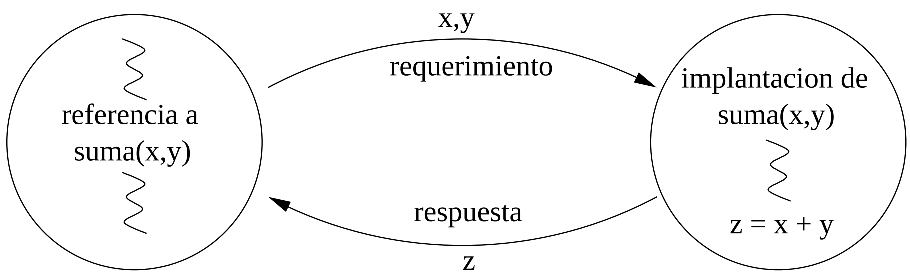

Figura 3.11: Mensajes explícitos en un RPC a suma(x, y)

Un mensaje explícito es un mensaje fiable que es parte de algún cálculo distribuido. Como ejemplo, considere la llamada a procedimiento remoto *suma(x, y)*. La implementación de ésta llamada utiliza dos mensajes explícitos: uno para el requerimiento, y otro para la respuesta (ver figura 3.11).

La noción de *piggyback* consiste en aprovechar implícitamente un mensaje explícito para enviar otro mensaje. Dicho de otro modo, el formato final del mensaje contendría los datos pertinentes al mensaje explícito, más una sección destinada a colocar el *piggyback*. La escritura del *piggyback* es realizada por alguna capa ejecutiva situada por debajo de la capa que origina el mensaje explícito. El remitente y el receptor de un *piggyback* son las correspondientes capas inferiores situadas debajo de las capas remitente y receptora del mensaje explícito (ver figura 3.12).

3.6.2. Suposiciones acerca de la arquitectura general del sistema de invocación

La invocación a objeto remoto es un caso extendido de una llamada a procedimiento remoto (RPC). Tradicionalmente, la arquitectura de un RPC es orientada a capas. El empilamiento de protocolos es el método que permite tratar con la complejidad inherente al desarrollo de un sistema RPC. Del mismo modo, el empilamiento es el mecanismo a usar en el desarrollo del sistema de invocación remota.

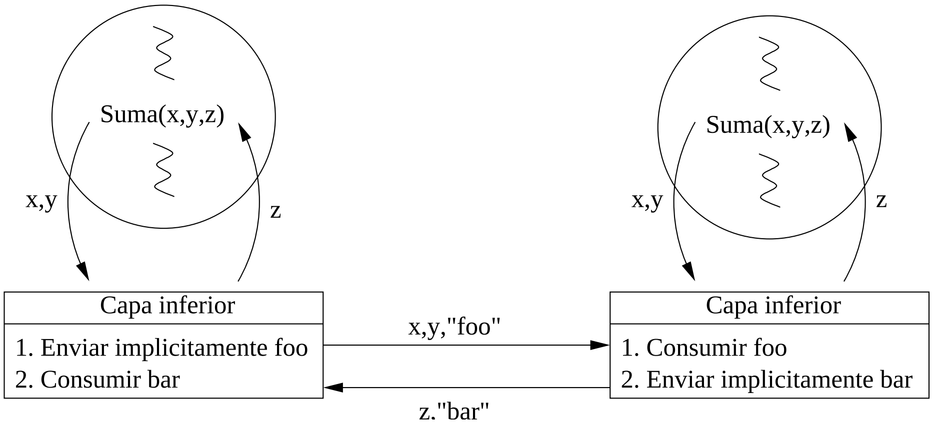

Figura 3.12: Interacción con la capa inferior de mensajes

La figura 3.13 bosqueja de forma resumida, una arquitectura general para el sistema de invocación remota.

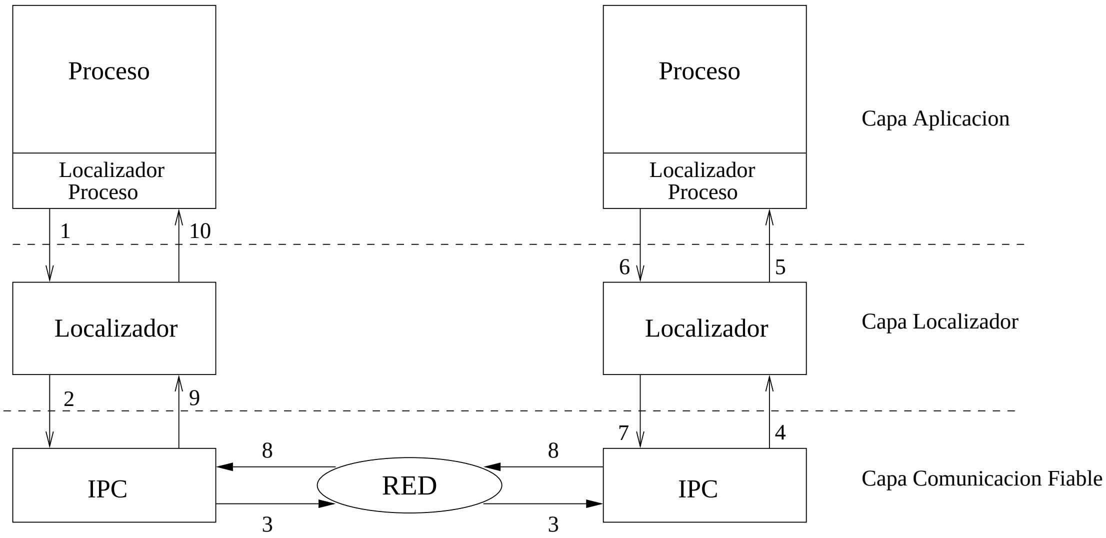

Figura 3.13: Mensajes explícitos en el localizador

En la figura 3.13 se aprecian los siguientes eventos:

1. Llamada externa al localizador.

2. Mensaje explicito del localizador al IPC que lleva *piggybacks*.

3. Comunicación fiable de una solicitud en la capa IPC.

4. Mensaje del IPC al localizador con el mensaje en bruto. En este momento el localizador puede recibir *piggybacks*.
5. Upcall que despierta la llamada bloqueante hecha en el localizador a la espera de una solicitud de invocación (lado servidor).
6. Respuesta del proceso al requerimiento hecho en la solicitud de invocación. En esta fase el localizador puede insertar *piggybacks*.
7. Pase de la respuesta al IPC, que lleva *piggybacks* puestos por la capa localizador.
8. Comunicación fiable de un mensaje de respuesta al mensaje del paso 3.
9. Pase de la respuesta del IPC al localizador. Al igual que en el paso 4, el localizador puede recibir *piggybacks*.
10. Upcall con el mensaje de respuesta a la invocación.

3.6.3. Uso de la invocación para envío de *piggybacks*

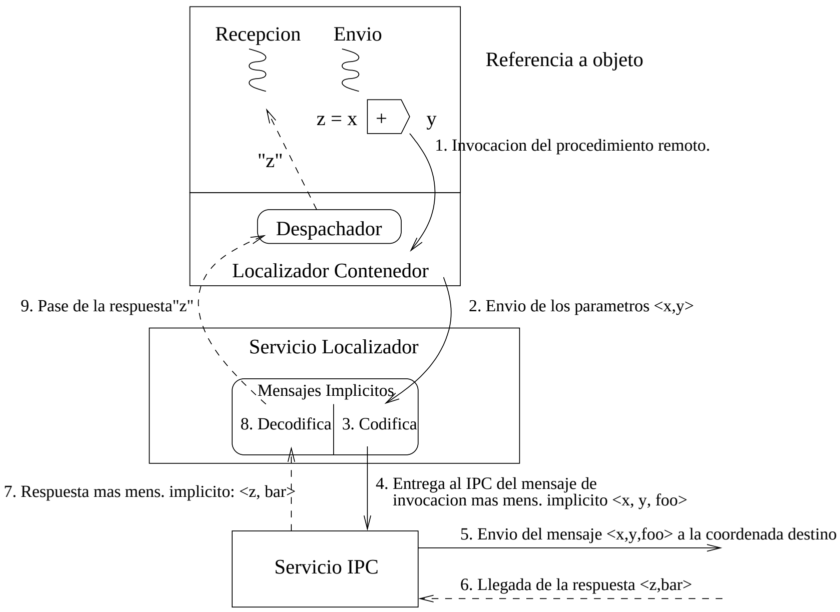

Figura 3.14: Mensajes explícitos en localizador remitente

Existen distintas formas de implantar los *piggybacks*. Considerando la primera, si se coloca la capa del localizador al mismo nivel de la capa IPC, entonces el localizador podría solicitarle a la capa IPC envío y recepción de *piggybacks*. En añadidura, se puede programar la capa IPC para que solicite permisos al localizador acerca del envío y recepción de mensajes. El localizador también puede instruir a la capa IPC para que extraiga parte del mensaje explícito y se lo entregue al localizador del sitio destino. Algo que se debe considerar en este esquema es la factibilidad de modificar el código de un sistema IPC existente. Al añadir código al sistema IPC se compromete la cohesión y se aumenta la probabilidad de errores. La tarea de programación y prueba del sistema localizador sería mucho más ardua. También, al ser el sistema más complejo, se tornaría más propenso a fallas de programación.

Por otra parte, si la capa del localizador se coloca entre el proceso y la capa IPC, se tendría la ventaja de no modificar el código del sistema IPC. Solamente se estaría utilizando la interfaz que ofrece el IPC para el envió y recepción de mensajes. Esto representa una solución más sencilla, elegante y modular al problema de la incorporación del sistema IPC al localizador. Es por ello que se prefiere utilizar este esquema al planteado en el párrafo anterior. Lamentablemente, esta solución tiene la desventaja de introducir la duplicación de mensajes.

La figura 3.14 esquematiza el proceso de envío y escritura de un *piggyback* desde el sitio remitente.

Cuando una referencia a objeto efectúa una invocación, ésta es capturada por el localizador. El localizador responde con un mensaje indicando el estado de la recepción de la invocación. El localizador puede, entonces, escribir un *piggyback* e, inclusive, modificar las coordenadas del mensaje explicito para que este sea redireccionado a otro sitio. Asimismo, en el localizador se puede gestionar algún mecanismo de permisología que determine si el envió es autorizado o no.

Una vez que el localizador haya procesado el mensaje, éste es entregado a la capa IPC para que se haga efectiva la transmisión. Cuando la capa IPC recibe un mensaje, se lo entrega a la capa localizador.

A la llegada de un mensaje al localizador (Figura 3.15) se extraen los identificadores de proceso y objeto destino, el identificador del servicio solicitado y la lista de *piggybacks*. El localizador será capaz de autorizar la entrega de un mensaje explicito a un proceso.

En adelante, asumiremos que todos los procesos pueden comunicarse mediante *piggybacks* contenidos en las invocaciones. Dado un lapso de tiempo t, el número de localizadores que pueden ser alcanzados mediante *piggybacks* en el lapso t depende de las invocaciones que ocurran durante ese lapso y del grado de conectividad dado por las referencias existentes. Durante el tiempo t, la cantidad de *piggybacks* que pueden ser enviados de un localizador a otro es proporcional a la cantidad de invocaciones que circulen entre ellos.

Antes de enunciar los tipos de *piggybacks*, es necesario plantear las suposiciones siguientes:

- Todo mensaje explicito, inclusive los de requerimiento y de respuesta, es utilizado por el localizador para enviar *piggybacks*.

- El localizador puede propagar los *piggybacks*. Si un localizador recibe un *piggyback*, éste puede decidir enviarlo a otros localizadores mediante otros mensajes explícitos.

- Una acción de búsqueda involucra un binding, lo que significa que se conoce el sitio desde donde se inició la búsqueda.

- Un *piggyback* puede ser identificado unívocamente por el identificador del objeto al que éste se refiere.

- Los *piggybacks* referentes a un objeto, serán actualizados conforme llegue información más reciente o de mayor prioridad. Las prioridades de los mensajes están descritas en § 3.6.5.

La figura 3.15 esquematiza el proceso de recepción y lectura de un *piggyback* en el sitio destino.

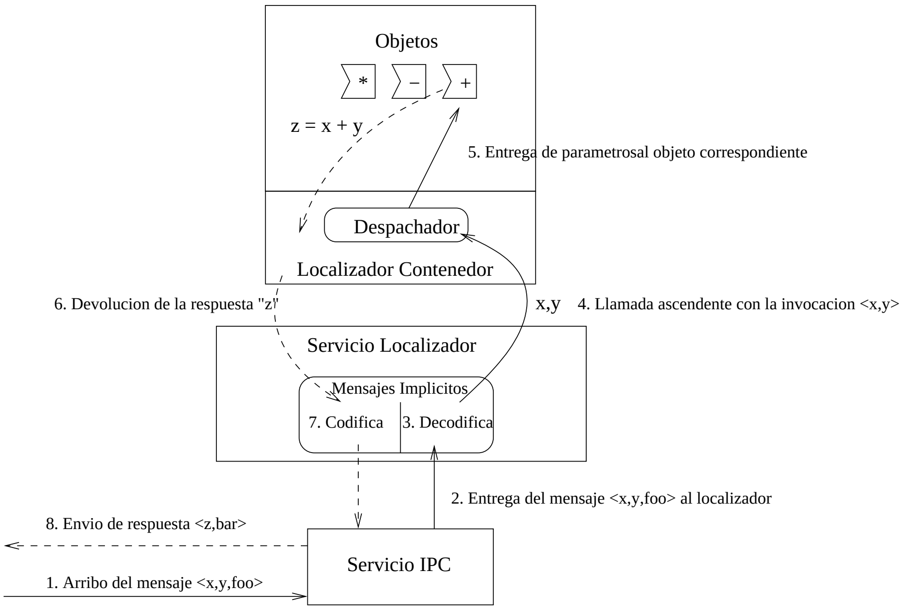

Figura 3.15: Mensajes explícitos en el localizador receptor

3.6.4. Tipos de *piggyback*

Existen tres tipos de *piggyback* que se describirán a continuación.

3.6.4.1. Búsqueda de objeto

Asumiremos un binding inválido $< x, y, R >$ que causa la emisión de un requerimiento de búsqueda. Este requerimiento es procesado inicialmente por el localizador l_x en el sitio x.

Cuando un localizador l_x recibe una solicitud de búsqueda por *piggyback* del objeto A, l_x intercepta invocaciones salientes del sitio y les incluye el *piggyback* "*el localizador l_x requiere actualizar el binding $< x, y, R >$* ". Este mensaje lo llamaremos "*se busca*".

Cuando un localizador l_z recibe un "*se busca*", l_z busca el objeto A en su tabla propietaria. Si lo encuentra, l_z envía un mensaje fiable a l_x notificándole la nueva referencia del objeto encontrado, que como ya se ha dicho, es potencialmente invalida, pues nada garantiza que en el momento en el que llegue la nueva referencia, el objeto se haya movido. El mensaje es explícito desde l_z hacia l_x.

Si l_z no encuentra el objeto A en su tabla propietaria, entonces l_z busca en sus caches una referencia más reciente. Si la encuentra, l_z envía un mensaje a l_x con la referencia más reciente.

Si la búsqueda tuvo éxito en los caches, y no en la tabla propietaria, l_z propaga el mismo "*se busca*" a otros localizadores. Esto se debe a que una entrada en los caches puede ser incorrecta.

3.6.4.2. Actualización de referencia

Cuando un objeto A migra hacia un sitio x, su tiempo lógico t es incrementado en una unidad, el localizador l_x intercepta invocaciones salientes y les incluye el *piggyback* "*En el tiempo lógico* $t+1$ *el objeto A arribó a x*". Este mensaje lo llamaremos "*nueva referencia*".

Cuando un localizador cualquiera l_z recibe éste mensaje, el localizador busca, en los caches de salida y de nuevas referencias, una entrada que involucre al objeto A. Si una entrada menos reciente es encontrada, entonces la entrada del correspondiente cache es actualizada. l_z propaga su "*actualiza referencia*" mediante las invocaciones salientes de z.

3.6.4.3. Eliminación de objeto

Cuando un objeto A es eliminado del sistema, su eliminación es anunciada mediante el *piggyback*: "*El objeto A fue eliminado*". A este mensaje lo llamaremos "*Eliminado*" y es propagado a los localizadores mediante *piggybacks* contenidos en las invocaciones.

Cuando un localizador recibe un "*eliminado*", el localizador elimina todas las entradas de sus caches que contengan al objeto y coloca el mensaje "*eliminado*" en el cache de eliminación. Luego el localizador propaga el mensaje el mensaje a otros localizadores.

3.6.5. Prioridades de los *piggybacks*

Los *piggybacks* están prioritizados según el siguiente orden, siendo 1 el más prioritario:

1. "*eliminado*".
2. "*se busca*".
3. "*nueva referencia*"

Dadas estas prioridades, no hace falta almacenar más que un solo *piggyback* por objeto, es decir, un mensaje de más prioridad sustituye al de menor prioridad.

Cuando un mensaje "*eliminado*" arriba a un localizador, cualquier otro mensaje referente al mismo objeto será descartado. Cuando un objeto es declarado eliminado, cualquier información sobre su presunta ubicación es falsa.

Cuando se tiene un mensaje "*se busca*" y llega un mensaje "*nueva referencia*", se comparan las estampillas de tiempo lógico. Si la "*nueva referencia*" es más reciente, el mensaje "*se busca*" es actualizado con las coordenadas de la "*nueva referencia*".

3.7. Control de congestión de *piggybacks*

Puesto que los *piggybacks* consumen recursos de comunicación, es necesario emplear mecanismos que controlen el flujo de los *piggybacks*.

Recordemos que todo *piggyback* debe ser etiquetado con un identificador único. Dado un *piggyback* m, un localizador mantiene información acerca de los sitios que ya conoce (o conocerá) m.

Dado un sitio remoto x, el localizador puede asumir que un *piggyback* es conocido (o será conocido) por el sitio x si:

1. El mensaje ha sido enviado al sitio x, o si

2. El mensaje ha sido recibido desde el sitio x

Esta técnica evita que un *piggyback* sea enviado dos o más veces a un mismo sitio. Empero, ella no evita que un sitio reciba un *piggyback* dos o más veces. Las secciones subsiguientes enuncian algunas heurísticas que pueden enfrentar este problema y cuyo uso y eficiencia son temas abiertos de investigación.

3.7.1. Grafo de frecuencia de invocaciones

Los caches de entrada y salida de los localizadores pueden ser usados para construir un grafo actual de invocaciones en la red. Este grafo estaría construido en base a las últimas invocaciones entrantes y salientes.

Según el grafo, cada localizador mantendría una lista "esperanza" de los sitios del cual va a recibir *piggybacks*. Si la estadística es buena, un sitio puede decidir no enviar *piggybacks* hacia aquellos sitios para los cuales se espera recibir invocaciones.

Otra lista "esperanza" mantendría los sitios al cual el localizador enviará *piggybacks*.

Este método exigiría anotar el punto de partida del *piggyback*.

Este grafo podría ayudar al localizador a decidir a cuáles sitios propagar un *piggyback* recibido.

3.7.2. Contador máximo de uso de *piggybacks*

Cada *piggyback* puede tener un contador de sitios por los cuales ha pasado, o cuantas veces a sido considerado para ser enviado. El contador también es incrementado cada vez que el mensaje llega a un sitio diferente. De esta manera, cada localizador puede manejar un máximo que anularía la propagación.

3.7.3. Tiempo de vida de un *piggyback*

Los *piggybacks* pueden tener un tiempo de vida en el sistema. La duración de este tiempo puede fijarse estáticamente o ser dinámicamente fijada en tiempo de creación por inclusión en el cuerpo del mensaje. El mensaje transportaría la fecha de creación.

Un localizador anularía el mensaje si la suma de la fecha de creación más el tiempo de vida es inferior a la fecha actual.

Por supuesto, este mecanismo sugiere sincronización física entre los demonios; algo posible en las redes de hoy en día; **ntp** , por ejemplo.

3.7.4. Máxima cantidad de *piggybacks* por mensaje explícito

Los *piggybacks* que viajan en un mensaje explicito pueden ser limitados por dos factores: el primero es una cantidad máxima de *piggybacks* por mensaje explicito, impuesta según criterios de performance. El segundo, es aprovechar el tamaño de un mensaje explicito, que no debe pasar el tamaño del MTU (Maximum Transfer Unit). Así, un mensaje cualquiera puede completarse con *piggybacks* hasta no sobrepasar el grano del MTU.

3.8. Difusiones (Broadcasts)

Un objeto puede ser localizado mediante una difusión de solicitudes de búsqueda. En su expresión más simple, el sitio que desea localizar un objeto efectúa una difusión y espera por la respuesta del localizador propietario del objeto (figura 3.16).

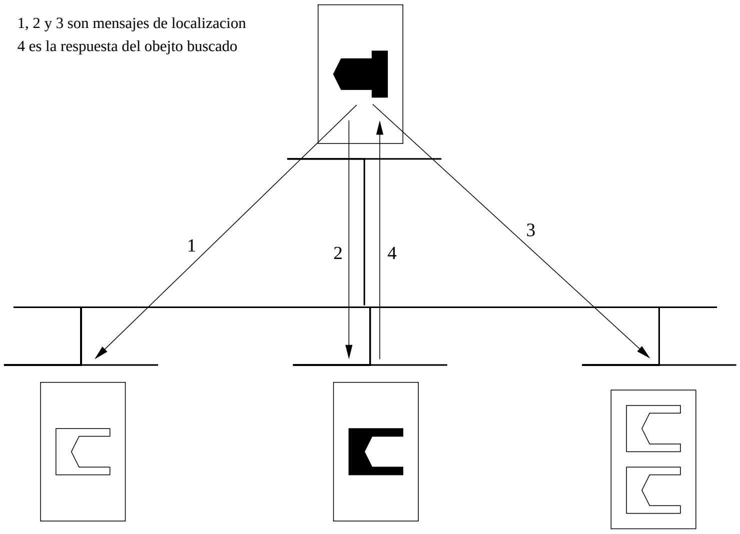

Figura 3.16: Uso de la difusión en la localización de un objeto

En la figura 3.16, los mensajes 1, 2 y 3 son enviados a cada uno de los nodos de la red. El mensaje 4 es la respuesta del sitio que alberga el objeto buscado.

Aunque esta técnica es efectiva en localizar el objeto, ella conlleva costes considerables al tráfico de red que son causados por la difusión. En una proporción menor, el resto de los sitios trabajan en vano.

Para disminuir y amortizar los costes de los protocolos de localización por difusión, se contemplan varios tipos de difusiones:

Difusión Débil: Una difusión es débil en el sentido de que no se garantiza que todos los sitios reciban el mensaje.

Difusión Fuerte: En contraposición, una difusión fuerte asegura que todos los sitios reciban el mensaje. La difusión fuerte tiene asociado un costo de comunicación más alto que la difusión débil, pues la certeza de recepción implica un mensaje de reconocimiento por sitio.

Se puede considerar un protocolo de localización con las difusiones débiles y otro con las fuertes, como sigue:

3.8.1. Protocolo de difusiones débiles

1. Efectuar una difusión débil solicitando respuesta del localizador propietario.
2. Si la difusión anterior no logra encontrar el objeto, entonces efectuar una difusión débil ordenando a cada localizador recabar toda la información que cualquier localizador tenga del objeto solicitado. La información es devuelta con mensajes fiables.

3.8.2. Protocolo de difusiones fuertes

1. Efectuar una difusión fuerte exigiendo respuesta del localizador propietario.
2. Si el paso anterior fracasa, entonces se difunde un mensaje fiable ordenando a todos los servidores reconstruir su tabla propietaria.
3. Si el paso precedente no logra localizar el objeto, entonces el objeto es declarado "inexistente".

Este protocolo es utilizado como último recurso; es decir, cuando otros métodos menos costosos no logren localizar al objeto.

Ambos protocolos cuentan con timeouts para la espera de respuestas, considerando que alguno de los localizadores no se encontrara operativo.

Una manera de utilizar las difusiones es haciendo primero una difusión débil y luego una fuerte. Esto ayuda a bajar los costos, pues existe la posibilidad de encontrar al objeto con una difusión débil. Empero, puede incrementar el costo, pues no es posible concluir que un objeto no existe después de finalizada una difusión débil. Es por ello que la forma de utilizar las difusiones se dejará a la libre decisión del cliente del sistema, quien pudiera considerar la confiabilidad de la red donde se encuentre el sistema.

3.9. Resumen

La técnica subyacente a nuestro sistema de localización fue el fracaso en la invocación. Esta técnica requiere de un servicio de localización al que se reportan las nuevas direcciones de los objetos al migrar.

El servicio de localización se apoya en el caching para mantener las referencias actualizadas. El caching consiste en guardar búsquedas previas con la finalidad de utilizarlas en un futuro sin necesidad de repetir el proceso de búsqueda.

Para mantener frescos los caches, se utilizan las técnicas de prefetching y *piggybacking*. El prefetching consiste en la utilización de los caches para mantener lo más actualizadas posible las referencias. El cache de entrada sirve para verificar las referencias de un objeto en particular cuando ha migrado. Con el cache de salida se puede averiguar periódicamente la validez de una referencia, y así actualizarla cuando sea inválida.

Los *piggybacks* son mensajes que se transportan en las invocaciones. Estos se encargan de llevar información actualizada sobre la localización de objetos. El número de *piggybacks* es proporcional al número de invocaciones en el sistema.

El último recurso a utilizar para la localización es la difusión. Para disminuir los costes, se considera un protocolo de difusiones en cascada, desde las difusiones débiles hacia las fuertes.

Capítulo 4

Escritura Genérica de Servicios Distribuidos

Esta sección nos deja una plantilla para el desarrollo de sistemas orientados a servicios distribuidos. A través de los servicios distribuidos, se puede aprovechar un conjunto de computadores conectados en red, en donde se estarán ejecutado procesos que prestan servicios. Mediante la ejecución coordinada de estos servicios se pueden lograr la consecución de tareas de forma cooperativa.

4.1. Arquitectura de Servicios Distribuidos

Se le llama servicios distribuidos, a un grupo de servicios que son administrados en un entorno de red por un conjunto de procesos llamados demonios. Generalmente, un demonio es asociado a una máquina, donde también se encuentran otros procesos que son sus clientes. Los demonios administran solicitudes que se refieren a servicios que para poder ejecutarlos, podrían requerir de la cooperación de otros demonios.

Mediante el modelo propuesto, se organiza jerárquicamente al conjunto de máquinas, demonios y clientes que intervienen. Los servicios distribuidos son necesarios para la resolución cooperativa de alguna tarea, en donde la información necesaria para resolverla se encuentra distribuida.

Tal como se muestra en la figura 4.1, en el conjunto de máquinas conectadas por una red, un demonio atiende varios procesos clientes. Estos procesos, pueden o no estar en la misma máquina donde está el demonio. La comunicación entre cliente y demonio ó entre demonios se hace a través de mensajes donde se indica el servicio que se requiere ejecutar.

Para solucionar el problema de comunicación fueron desarrollados dos esquemas: comunicación síncrona y comunicación asíncrona. En la comunicación síncrona, un cliente se queda bloqueado a la espera de la respuesta por parte del proceso que presta servicio, mientras que la comunicación asíncrona un cliente no bloquea a la espera de la respuesta, más aun, puede solicitar la respuesta en cualquier instante de tiempo. Una aplicación de la comunicación síncrona es la comunicación local entre clientes y demonios, y para la comunicación asíncrona pudiera ser la comunicación remota entre demonios.

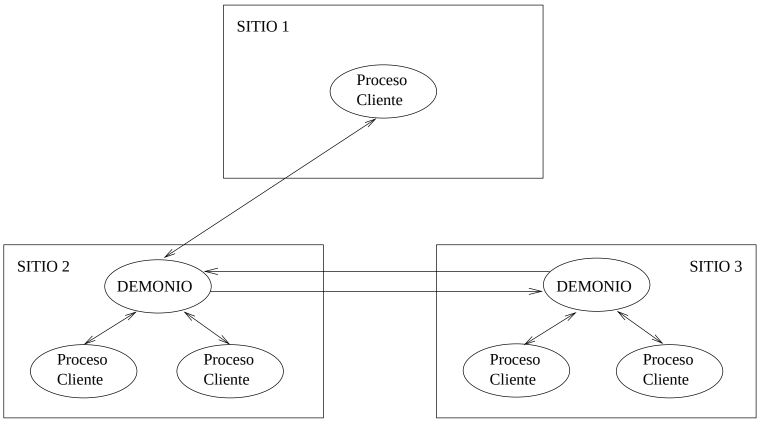

Figura 4.1: Esquema general de los servicios distribuidos

4.1.1. Arquitectura cliente/servidor

Un par cliente/servidor se distingue porque el cliente envía mensajes, posiblemente con solicitudes, a un servidor que los procesa y eventualmente retorna una respuesta.

En la figura 4.2 se aprecia un bosquejo general del modelo propuesto, en donde un cliente cualquiera solicita la ejecución de un servicio. El cliente, al igual que el servidor, consta principalmente de un mecanismo para el envío y recepción de mensajes.

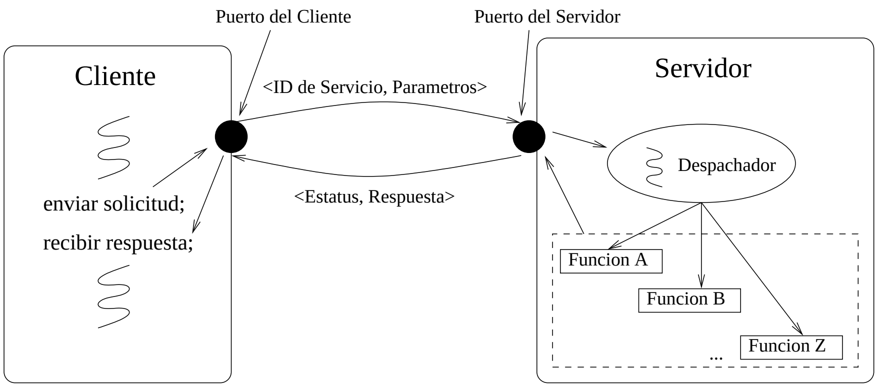

Figura 4.2: Modelo general de la interacción cliente/servidor

El cliente previamente conoce el puerto por donde debe escribirle mensajes al servidor. Una vez que un cliente envía una solicitud, el servidor conoce el puerto por donde el cliente estará esperando la respuesta.

Un mensaje cualquiera de un cliente a un servidor, debe tener un identificador del servicio a ejecutar y los parámetros que utilizará el servicio. Una vez que llega el mensaje con la solicitud, se despacha al servicio correspondiente; éste es ejecutado y luego el servidor puede, atómica y serialmente, atender otras solicitudes. Luego, el servidor retorna una respuesta al cliente, indicando mediante una variable de estatus cualquier evento ocurrido durante la ejecución del servicio.

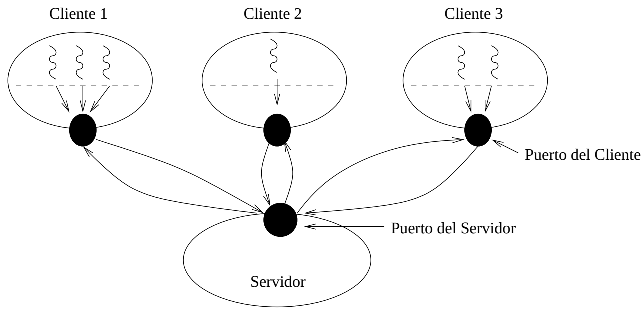

Figura 4.3: Acceso de un servicio por múltiples clientes

Se contempló la posibilidad de tener varios clientes accediendo al mismo servidor simultáneamente, tal como se muestra en la figura 4.3. También se contempló la posibilidad de tener varias *threads* que envían mensajes por el mismo puerto de un cliente. Para ello, se garantiza el acceso exclusivo al puerto mediante estructuras de exclusión mutua. Si se quiere evitar la espera obligatoria para la utilización del puerto, entonces deben instanciarse tantos puertos como se requieran en un mismo cliente.

Del lado del servidor, existe un mecanismo que despacha las solicitudes al servicio correspondiente. Este mecanismo opera de manera síncrona y serializada. Es decir, dos clientes c_1 y c_2, que simultáneamente soliciten la ejecución de un mismo servicio s_1, en el servidor serán atendidos según algún orden; bien sea $[c_1,\ c_2]$ ó $[c_2,\ c_1]$. También se sugiere la ejecución asíncrona creando una *thread* por servicio, lo cual quedaría a la entera responsabilidad del programador.

4.1.2. Protocolo de Registro de Servicios

En un servidor, un servicio debe ser registrado debido a que necesita ser conocido por el despachador. Para poderlo registrar hacen falta: una firma de una función, un código numérico y una cadena de caracteres.

Un cliente registra los servicios como una medida de seguridad. En el cliente son necesarios el mismo código numérico y la misma cadena de caracteres que fueron utilizados en el servidor.

En la comunicación síncrona, primero un servidor registra los servicios y luego lo harán sus clientes. Los clientes al registrar un servicio, envían un mensaje al servidor que les autorice usarlo. Una vez, autorizado el cliente, las subsiguientes solicitudes del servicio sean procesadas.

Esto último es válido en el contexto de la comunicación síncrona. En la comunicación asíncrona es imposible realizar la verificación de los servicios por parte del cliente, pues un cliente no depende de un solo servidor. En este caso solo existe el registro de los servicios por parte de cada servidor, pues no existe la figura de cliente puro.

4.2. Interfaz

A continuación se explica la interfaz que debe utilizarse para hacer un sistema de servicios distribuidos. Primero se presentarán un par de clases bases que servirán para la especialización en clases para los pares cliente/servidor síncronos local y remoto. Luego, se presentará la interfaz para el servicio de comunicación asíncrono remoto.

4.2.1. Comunicación Síncrona

La comunicación síncrona se caracteriza porque el cliente que ejecuta una solicitud, se bloquea hasta la llegada de la respuesta. Luego de llegar la respuesta, el cliente es desbloqueado, y nuevas solicitudes pueden realizarse.

4.2.1.1. Sistema de Mensajes

Existen dos tipos de mensajes en la comunicación síncrona: los mensajes cliente-servidor y los mensajes servidor-cliente. Los mensajes cliente-servidor deben heredar de una clase base mensaje llamada `Msg_Entry_Header`, mientras que los mensajes servidor-cliente lo hacen de `Msg_Exit_Header`. Todo el sistema de mensajes se encuentra implantado en el archivo `base_message_header.H`.

El constructor de la clase `Msg_Entry_Header` tiene la forma siguiente:

```
Msg_Entry_Header(const long srvc_code, const size_t msg_size);
```

Los parámetros de la clase `Msg_Entry_Header` son el servicio a ejecutar `srvc_code` y el tamaño del mensaje `msg_size`.

El siguiente es un ejemplo de un mensaje cliente- servidor, donde se tiene un conjunto de servicios que se especifican a través de un tipo enumerado (`My_Services`) y una clase mensaje (`Message_For_Service_1`) que hereda de `Msg_Entry_Header`:

```
enum My_Services
{
 MY_SERVICE_1,
 MY_SERVICE_2,
 // ...
 MY_SERVICE_N
};

class Message_For_Service_1 : public Msg_Entry_Header
{
 // Whatever data goes here.

 public:
 Message_For_Service_1() :
   Message_Entry_Header(MY_SERVICE_1, sizeof(*this))
 {
   // empty
```

```
 }
};
```

Los mensajes servidor-cliente ser heredados de la clase `Msg_Exit_Header`, cuyo constructor tiene la siguiente forma:

```
Msg_Exit_Header(const long resp_code)
```

El único parámetro de `Msg_Exit_Header` consiste en el estatus de la respuesta (`resp_code`). Mediante esta variable de estatus, un servidor puede comunicarle a un cliente cualquier evento ocurrido al prestar un servicio. En el siguiente ejemplo, se muestra un mensaje servidor-cliente, donde el estatus aparece codificado en un tipo enumerado (`My_Status_Code`), y el mensaje de respuesta `Response_From_Service_1` hereda de `Msg_Exit_Header`.

```
enum My_Status_Code
{
 STATUS_CODE_1,
 STATUS_CODE_2,
 // ...
 STATUS_CODE_n
};

class Response_From_Service_1 : public Msg_Exit_Header
{
 // Whatever data goes here.

 public:
 Response_From_Service_1(const long status_code) :
   Msg_Exit_Header(status_code)
 {
   // empty
  }
};
```

4.2.1.2. Identificación de Clientes y Servidores

Para identificar clientes y servidores se dispone de sendas clases llamadas `Binding_Point` y `Deamon_Locator` respectivamente.

La clase base `Binding_Point` almacena la dirección de retorno de un cliente que ha solicitado un servicio. La clase exporta los métodos siguientes:

```
virtual ssize_t respond (Msg_Exit_Header * return_msg,
                         const size_t msg_size) = 0;

virtual ssize_t respond (Msg_Exit_Header * return_msg,
                         const size_t msg_size,
```

```
                              const void * buffer,
                              const size_t buffer_size) = 0;
```

Las funciones `respond` sirven para devolver la respuesta de un servicio. Los dos primeros parámetros comunes a ambos son: la dirección del mensaje de respuesta `return_msg` y su tamaño `msg_size`. Con la segunda versión de `respond` se puede enviar además un `buffer` de tamaño `buffer_size`. El `buffer` apunta a un bloque de bytes, que pueden ser utilizados para múltiples propósitos, entre ellos, enviar la respuesta en un sistema de invocación.

La clase base `Deamon_Locator` se utiliza para la identificación de un servidor. Esta clase tiene una único método cuyo trabajo es el de convertir un identificador de servidor en una cadena de caracteres:

```
virtual const char * stringficate() const = 0;
```

Este método es muy útil cuando hay que pasar por parámetro un identificador de algún servidor por linea de comandos.

4.2.1.3. Interfaz para el Servidor

La interfaz del servidor esta compuesta de funciones para recepción de mensajes, registro de servicios y eliminación de servicios. Un servidor tiene como función tomar los mensajes que envía un cliente y despachar la solicitud al respectivo servicio, luego desde allí responder al cliente.

La clase base utilizada del lado servidor es llamada `Reception_Point` y debe ser invocada con los siguientes parámetros:

```
Reception_Point(unsigned int no_of_services,
                Deamon_Engine_Fct_t _engine_fct)
```

El parámetro `no_of_services` especifica el número de servicios que ofrecerá el servidor, y la función `_engine_fct` permite definir los detalles del procesamiento de la solicitud. Aquí se desembala el mensaje enviado por un cliente conforme al tipo de comunicación utilizada (local ó remota). Luego, esta rutina envía un mensaje al despachador de servicios.

Los métodos que ofrece esta clase son los siguientes:

```
pthread_t * start_deamon()
throw (UnexpectedException);
```

Esta rutina inicia la *thread* de recepción. Una vez que se instancia un servidor de tipo `Reception_Point` la primera acción debería ser la ejecución de la rutina `start_deamon()`. Así, el demonio estará en condiciones de registrar servicios, escuchar y responder a las solicitudes hechas. La función `start_deamon()` devuelve la dirección de la *thread* donde se ejecuta en caso de que se requiera averiguar sobre su estado. La excepción `UnexpectedException` será arrojada si la *thread* no pudo ser creada.

Registro y Borrado de Servicios

El registro de servicios consiste en hacer conocer al despachador, cuales servicios se estarán prestando. Cada servicio es una función, entonces el despachador tendrá una tabla con las direcciones de las funciones que serán oportunamente despachadas:

```
void add_service(long service_code, char * name, Service_Fct sf)
     throw (Duplicated);
```

El método anterior registra un servicio específico. La excepción `Duplicated` sera arrojada en el caso de que se quiera registrar el mismo servicio dos veces. El primer parámetros es el código del servicio (`service_code`), que por lo general se utiliza un tipo enumerado (`enum`), el segundo es una cadena de caracteres que lo identifica (`char * name`), y el tercero (`sf`) es la dirección de la función que presta el servicio:

```
int (*Service_Fct) (Msg_Entry_Header * input_msg,
                    Binding_Point * return_point);
```

Una función con este tipo procesa un mensaje de tipo `Msg_Entry_Header *` y puede responder a través del identificador del cliente `return_point` (ver § 4.2.1.2).

En el siguiente ejemplo, se muestra la forma de un servicio (`service_one`), donde los posibles eventos luego de la ejecución están definidos en `My_Status_Code`.

```
enum My_Status_Code
{
 STATUS_CODE_1,
 STATUS_CODE_2,
 // ...
 STATUS_CODE_n
};

int service_one(Msg_Entry_Header * input_msg, Binding_Point * return_point)
{
 // cast input_msg. (1)
 Input_Type * itp = static_cast<Input_Type *>(input_msg);

 // do whatever work has to be done. (2)

 // prepare response to client. (3)
 Output_Type ot(STATUS_CODE_1);

 // respond to client. (4)
 return_point->respond(&ot, sizeof(Output_Type));

 return 0;
}
```

Hay cuatro partes en un servicio: la transformación (*casting*) del mensaje de entrada (1), el conjunto de instrucciones que procesan el mensaje y dan la forma al servicio (2), la preparación de la respuesta (3) y el retorno de la respuesta (4). Aunque no es un esquema rígido a seguir, es el más directo.

Para eliminar un servicio registrado por `add_service` se utiliza la función `delete_service`:

```
void delete_service(const long service_code)
     throw (NotFound);
```

Aquí solo es necesario especificar el código del servicio (`service_code`). En caso de no encontrar el servicio especificado, entonces la excepción `NotFound` será arrojada.

Luego, es posible reemplazar dinámicamente un servicio si, primero se borra el servicio existente con el método `delete_service` y luego se registra el nuevo servicio con `add_service`.

4.2.1.4. Interfaz para el Cliente

La interfaz para el cliente está compuesta de funciones de envío y recepción de mensajes, y funciones de registro y borrado de servicios. Una vez que se ha enviado una solicitud, es necesario ejecutar la función que permite recibir la respuesta. De no ser así, otros clientes que compartan el puerto, quedarán bloqueados al tratar de enviar un mensaje.

Constructor de la Clase Access_Point

La clase que se utiliza del lado del cliente se llama `Access_Point`. Cada instancia de `Access_Point` corresponde a un canal de comunicación que permanece abierto hasta la destrucción del objeto.

Para obtener una instancia es necesario especificar los siguientes parámetros en la llamada al constructor:

```
Access_Point(unsigned int _no_of_services,
             Read_Fct _read_fct,
             Write_Fct _write_fct);
```

El parámetro `_no_of_services` indica el número de servicios que se quieren acceder desde el cliente. Las funciones de lectura y escritura de mensajes, tienen una interfaz ya definida. Mediante las instancias de estas funciones, se especifica de que manera se leen y escriben los mensajes de acuerdo al mecanismo de comunicación utilizado.

El tipo de la función de lectura de mensajes tiene tres parámetros a saber:

```
typedef ssize_t (*Read_Fct)(Deamon_Locator * dl,
                            void * buffer,
                            const size_t buffer_size);
```

El parámetro `dl` permite fijar unívocamente las coordenadas del servidor desde donde se esperan mensajes de respuesta. Mientras que `buffer` y `buffer_size` son el buffer donde se depositarán los bytes leídos del mensaje y el tamaño esperado respectivamente.

Los parámetros para la función de escritura, son prácticamente iguales, solamente difiere en el tipo del parámetro `buffer`. El parámetro `const void * buffer` indica que al escribir los bytes del mensaje, no serán modificados:

```
typedef ssize_t (*Write_Fct)(Deamon_Locator * dl,
                             const void * buffer,
                             const size_t buffer_size);
```

Registro y Borrado de Servicios

Para gestionar los servicios del lado del cliente se dispone de dos funciones:

```
void add_service(const long service_code, char * name)
     throw (Duplicated);

void delete_service(const long)
     throw (NotFound);
```

La función `add_service` se utiliza para verificar desde el cliente la existencia de un servicio. Cuando esta función es utilizada, se ejecuta una consulta al servidor que chequea que exista el servicio especificado. Los parámetros `service_code` y `name` son el código de servicio y una cadena de caracteres que lo identifica respectivamente.

Este tipo de protocolo ayuda al programador a ser más consistente en la especificación de un servicio. La excepción `Duplicated` es arrojada en caso de que se intente registrar dos veces un mismo servicio.

Funciones de envío y recepción de mensajes

El siguiente par de métodos permite el envío de una solicitud al servidor. Un apuntador con la dirección del mensaje enviado es pasada como primer parámetro (ver § 4.2.1.1). La excepción `NotFound` será arrojada en caso de que el servicio solicitado no haya sido registrado del lado del cliente.

```
void send(const Msg_Entry_Header * request) const
     throw (NotFound);

void send(Msg_Entry_Header * request,
          const void *       buffer,
          const size_t       buffer_size) const
     throw (NotFound);
```

La última de las versiones de `send`, envía además, un bloque de bytes especificado en `buffer` cuyo tamaño es `buffer_size`. Esta opción puede ser utilizada para trasladar invocaciones o mensajes extras que son tratados como un conjunto continuo de bytes.

Para recibir mensajes de respuesta del servidor se utiliza el siguiente par de funciones:

```
void receive(Msg_Exit_Header * reply, const size_t reply_size)
    throw (SizeFault, NotFound);
```

```
void receive(Msg_Exit_Header * message,
             const size_t      message_size,
             void *            buffer,
             const size_t      buffer_size)
     throw (SizeFault, NotFound);
```

Son necesarios como parámetros: `reply`, que es la dirección donde la respuesta será copiada y `reply_size`, que corresponde al tamaño esperado de la respuesta.

La llamada `receive` se bloquea hasta recibir la respuesta de un servicio. La excepción `SizeFault` es arrojada en caso de que el tamaño esperado de la respuesta no coincida con el tamaño que ha sido recibido. La excepción `NotFound` será arrojada cuando se detecte que el servidor ha dejado de funcionar.

El último método de recepción es semánticamente idéntico a la primero. La diferencia está en que además del mensaje recibido, es posible recibir un bloque extra de bytes. Estos bytes recibidos son colocados en `buffer`, y el tamaño máximo esperado de éste es `buffer_size`. El `buffer` puede ser utilizado para la recepción de invocaciones. La excepción `SizeFault` será arrojada cuando no coincida el tamaño esperado de mensaje ó cuando el bloque de bytes exceda el máximo (`buffer_size`) permitido.

4.2.1.5. Clases especializadas para comunicación local y remota

Para desarrollar una aplicación de servicios distribuidos, hay dos tipos de `Access_Point` y `Reception_Point` especializados:

- El par `Local_Access_Point/Local_Reception_Point`, que sirven para gestionar la comunicación local entre dos procesos cualquiera. En el modelo de servicios distribuido propuesto se utiliza para gestionar la comunicación entre cliente y demonio.
- El par `Remote_Access_Point/Remote_Reception_Point`, que gestionan la comunicación síncrona entre dos procesos situados en dos máquinas distintas.

4.2.1.6. Las clases de comunicación local

Este par utiliza **Unix Domain Sockets** para la transmisión de los mensajes. Los sockets utilizados son del tipo **stream** , esto garantiza la comunicación confiable entre dos procesos. Estas clases están disponibles en `local_access_point.H` y `local_reception_point.H`.

Para poder utilizar el par de comunicación local fue necesario tener una especialización de la clase `Deamon_Locator`. Esta especialización sirve para identificar la dirección donde se encuentra el servidor y consiste de un camino absoluto en el sistema de archivos. Un ejemplo de como utilizar `Local_Locator` es el siguiente:

```
Local_Locator server_locator("\dir1\dir2\unique_file");
```

También fue necesario una especialización de la clase `Binding_Point`. Esta fue llamada `Local_Binding` y tiene como función, determinar la dirección de retorno de un cliente. Por medio de una instancia de esta clase es como se responde al cliente que espera por la ejecución de un servicio. La propia clase `Local_Reception_Point` se encarga de manejar las instancias de `Local_Binding`. El constructor de la clase del servidor local tiene la forma siguiente:

```
Local_Reception_Point(const Local_Locator & _local_loc,
                      const unsigned int _no_of_services);
```

Para poner en funcionamiento al servidor, debe instanciarse la clase `Local_Reception_Point` con dos parámetros: `_local_loc`, que indica cual será el punto de recepción para que los clientes puedan conectarse y `_no_of_services` que indica la cantidad de servicios que estarán registrados en el servidor.

Luego de instanciar la clase, debe ponerse en funcionamiento el servidor con el método `start_deamon()` tal como se muestra a continuación:

```
const short NO_OF_SERVICES = 15;

Local_Locator locator("/dir1/dir2/unique_file");

Local_Reception_Point server(locator, NO_OF_SERVICE);

server.start_deamon();
```

Seguidamente, deben registrarse los servicios que se estarán prestando. Para ello, hace falta tener definidas las funciones que se encargarán de llevar a cabo los servicios. En el siguiente ejemplo se observa la definición de los códigos de los servicios, tal como en § 4.2.1.1. Luego, el tipo de los servicios (`service_1`, `service_2`, etc.) es como se explicó en § 4.2.1.2.

```
// enumerative type for indexing services.
enum My_Services
{
 SERVICE_1,
 SERVICE_2,
 // ...
 SERVICE_n
};

// service functions.
extern int service_1(Binding_Point *, Msg_Entry_Header *);
extern int service_2(Binding_Point *, Msg_Entry_Header *);
// ...
extern int service_n(Binding_Point *, Msg_Entry_Header *);

// create server
const short NO_OF_SERVICES = 15;
Local_Locator locator("/dir1/dir2/unique_file");
Local_Reception_Point server(locator, NO_OF_SERVICES);
server.start_deamon();
```

```
// registering of services.
server.add_service(SERVICE_1, "id string for service 1", sevice_1);
server.add_service(SERVICE_2, "id string for service 2", sevice_2);
// ...
server.add_service(SERVICE_n, "id string for service n", sevice_n);
```

Una vez que se han registrado los servicios el servidor puede empezar a procesar solicitudes de los clientes.

Para comunicar un cliente con el servidor anterior, debe instanciarse del lado del cliente un `Local_Access_Point`. Luego, se deben registrar todos los servicios a utilizar antes de empezar a hacer solicitudes.

El tipo enumerado para los servicios del cliente es exactamente el mismo del servidor. Es decir, `My_Services` es el tipo que se estará utilizando en el cliente. De la misma manera las cadenas de caracteres que identifican los servicios en el cliente deben ser las mismas que se utilicen en el servidor.

Un cliente, debe incluir el siguiente código:

```
// enumerative type for indexing services, same as server.
enum My_Services
{
 SERVICE_1,
 SERVICE_2,
 // ...
 SERVICE_n
};

Local_Locator locator("/dir1/dir2/unique_file");

Local_Access_Point client(locator, NO_OF_SERVICES);

client.add_service(SERVICE_1, "id string for service 1");
client.add_service(SERVICE_2, "id string for service 2");
// ...
client.add_service(SERVICE_n, "id string for service n");
```

Una vez que se hayan procesado las lineas del código anterior, un cliente podrá empezar ha realizar solicitudes al servidor. Para tal fin, debe emplear el mecanismo de mensajes descrito en **Funciones de Envío y Recepción de Mensajes** (§ 4.2.1.4).

4.2.1.7. Las clases de comunicación remota

Este par se utiliza para comunicar un cliente y un servidor que se encuentren en dos máquinas distintas. Para la comunicación remota, confiable y de semántica exactamente una vez es utilizado el sistema IPC [12]. El sistema IPC utiliza puertos garantizados únicos para la identificación de un servidor.

Las clases de comunicación remota se encuentran en `remote_access_point.H` y en `remote_reception_point.H`

La única diferencia respecto al funcionamiento del par `Local_Access_Point` y `Local_Reception_Point` es la definición de la clase especializada `Deamon_Locator`. Un `Deamon_Locator` en el sistema de comunicación remoto se construirá con un puerto válido de servidor, tal como se muestra en el siguiente código:

```
char * port_string = /* port string definition */
Port server_port(port_string);
Remote_Locator locator(server_port);
```

Entonces, las únicas lineas de código que deben cambiarse son las encargadas de crear el `Daemon_Locator`, que para este caso se llama `Remote_Locator`. Del resto tanto el servidor como el cliente operan de forma idéntica a sus equivalentes locales.

4.2.2. Comunicación Asíncrona

En la comunicación asíncrona la *thread* que envía un mensaje en el proceso cliente no se bloquea a la espera de la respuesta. Como base de la comunicación asíncrona se utilizó el sistema IPC [12].

Para lograr este propósito, son necesarias al menos dos *threads*. Una para enviar los mensajes y otra para recibir. Además de esto, el IPC ofrece un sistema de identificación de mensajes de solicitud y respuesta, de tal forma que se puede llevar un control para seguir la pista de un RPC. Un par solicitud/respuesta tendrán entonces el mismo identificador de mensaje.

La interfaz de las clases de comunicación asíncrona es muy parecidas a la de la comunicación síncrona. De hecho, se cuenta con prácticamente con los mismos métodos de las clases, con la excepción de que se añaden algunas funcionalidades.

La diferencia con la comunicación síncrona es que la clase de comunicación asíncrona `Remote_Multiserver_Point`, tiene la propiedad de ser cliente y servidor a la vez. Así, la comunicación de un conjunto de procesos que poseen instancias de esta clase, puede hacerse entre todos los posibles pares del conjunto.

Un `Remote_Multiserver_Point` no realiza el protocolo de registro de servicios del lado del cliente. Esto se debe a que se pierde la noción de un procesos cliente y servidor, dedicados. Sin embargo, los servicios deben ser registrados tal como se hizo en el servidor síncrono para que puedan ser despachados.

En la clase de comunicación asíncrona existe la alternativa de registrar los mismos servicios en todos los servidores. Luego, las solicitudes de ejecución de cualquiera de los servicios comunes, es implementada por un subconjunto de métodos y, otro subconjunto esta destinado para la ejecución de servicios que no entran en el común acuerdo.

Debido a que no existe una separación entre el cliente y el servidor, los mensajes de solicitud de servicio y de respuesta serán ambos derivados de la clase `Msg_Entry_Header` (4.2.1.1).

4.2.2.1. La clase template <class Service_Class> Remote_Multiserver_Binding

Esta clase determina las coordenadas del `Remote_Multiserver_Point` que necesita una respuesta de un servicio. En este sentido un `Remote_Multiserver_Binding` ve al `Remote_Multiserver_Point` como un cliente. La clase plantilla `Service_Class` que se pasa como parámetro inicial, indica la clase que contiene los servicios que se prestarán.

Los métodos que exporta esta clase son:

```
ssize_t rpc_reply (const Msg_Entry_Header * data,
                   const size_t data_size)
```

El método `rpc_reply` hace la contestación a un cliente que ha solicitado la ejecución de un servicio. El mensaje de respuesta debe ser derivado de `Msg_Entry_Header` y su tamaño total se especifica en `data_size`. Normalmente, este método es utilizado cuando sea necesario al final de un servicio.

```
ssize_t rpc_reply (const Msg_Entry_Header * data,
                   const size_t data_size,
                   const void * buffer,
                   const size_t buffer_size)
```

Esta variante del `rpc_reply`, permite además retornar un bloque de bytes extra apuntados por `buffer` y de tamaño `buffer_size`. El bloque de bytes (`buffer`) normalmente se puede utilizar para enviar la respuesta en un sistema de invocación.

4.2.2.2. La clase template <class Service_Class> Remote_Multiserver_Point

La clase `Remote_Multiserver_Point` tiene como función servir de cliente y servidor al mismo tiempo. Para ello dispone de una *thread* que se dedica exclusivamente a recibir, mientras que desde cualquier otra *thread*, pueden ser invocados los métodos para enviar mensajes. Esta clase tiene como parámetro inicial la clase que contendrá los servicios.

Así, los servicios prestados por esta clase son de dos tipos: servicios con respuesta y servicios sin respuesta:

```
typedef int (Service_Class::*Sync_Service_Fct)
            (Msg_Entry_Header * entry_msg,
             Remote_Multiserver_Binding<Service_Class> * return_point);

typedef int (Service_Class::*Async_Service_Fct)
            (Msg_Entry_Header * entry_msg);
```

El primer tipo de función corresponde a los servicios desde donde se puede responder al cliente. Entonces, el tipo de mensaje que se recibe es `entry_msg` y la dirección de retorno del cliente es `return_point`.

El segundo tipo de función es para los servicios que no retornan ninguna respuesta al cliente. Solo reciben un mensaje de entrada (`entry_msg`).

Para instanciar esta clase, el constructor necesita los siguientes parámetros:

```
Remote_Multiserver_Point<Service_Class> (const long _no_of_services,
                                         Service_Class * const _ptr_serv_class)
```

El primer parámetro indica el número de servicios a prestar. El segundo es la dirección del objeto que posee los servicios.

Para añadir servicios se procede de igual manera que con las clases de comunicación síncrona, solo que ahora se puede escoger entre dos tipos de servicios: servicios sin respuesta (`Async_Service_Fct`) y servicios con respuesta (`Sync_Service_Fct`):

```
void add_service(const long servc_code,
                 const char * serv_name,
                 Async_Service_Fct asf)
throw (NotFound);

void add_service(const long servc_code,
                 const char * serv_name,
                 Sync_Service_Fct ssf)
throw (NotFound);
```

Funciones de envío y recepción de mensajes

Los métodos de envío pueden ser clasificadas de dos tipos: métodos de envío para servicios comunes, y métodos de envío para cualquier servicio.

Los métodos de envío para servicios comunes están registrados en un conjunto de instancias de `Remote_Multiserver_Point<Service_Class>` y en cada una de ellas se tienen los mismos servicios con los mismos códigos. Sin embargo, operan sobre conjuntos de datos distintos:

```
void async_send(const Port & target_port,
                Msg_Entry_Header * request,
                size_t request_size)
throw (NotFound);

void rpc_send(const Port & target_port,
      Msg_Entry_Header * request,
      size_t request_size)
throw (NotFound);

void rpc_send(const Port & target_port,
      Msg_Entry_Header * request,
      size_t request_size,
      void * buffer,
      size_t buffer_size)
throw (NotFound);
```

Estos métodos tienen en común los tres primeros parámetros, que indican el puerto destino, un apuntador con la dirección del mensaje (`request`) y el tamaño del mensaje (`request_size`). Además, la excepción `NotFound` será arrojada si no se consigue registrado el servicio especificado.

La diferencia está en que el servicio que procesará un mensaje enviado con `async_send` no responderá. Mientras que los servicios que procesen mensajes enviados con `rpc_send` eventualmente retornarán un mensaje de respuesta.

El último método (`rpc_send`) de envío de mensajes tiene dos parámetros extra, que son `buffer` y `buffer_size`. Estos parámetros son la dirección de un bloque de bytes y su tamaño respectivamente.

Los métodos de envío que se utilizan para solicitar servicios no comunes, no hacen el chequeo del servicio registrado y por ende no arrojarán ninguna excepción:

```
MsgId async_send_to_other_server(const Port & target_port,
                                 Msg_Entry_Header * request)

MsgId async_send_to_other_server(const Port & target_port,
                                 Msg_Entry_Header * request,
                                 size_t request_size,
                                 const void * buffer,
                                 size_t buffer_size)

MsgId rpc_send_to_other_server(const Port & target_port,
                               const Msg_Entry_Header * request,
                               size_t request_size)

MsgId rpc_send_to_other_server(const Port & target_port,
                               const Msg_Entry_Header * request,
                               size_t request_size,
                               const void * buffer,
                               size_t buffer_size)
```

Estos métodos, también retornan un `MsgId` que es un identificador del mensaje saliente. El `MsgId` sirve para que el cliente pueda llevar un control en el envío y recepción de los mensajes.

El siguiente es un ejemplo de como utilizar esta clase. El código de los servicios que se prestarán, esta especificado en `My_Services`:

```
# define NO_OF_SERVICES 10

enum My_Services
{
  MY_SERVICE_1,
  MY_SERVICE_2,
  ...
  MY_SERVICE_n,
};

class Any_Server : public Remote_Multiserver_Point<Any_Server>
```

```
{
  int service_1(Msg_Entry_Header *,
                Remote_Multiserver_Binding<Any_Server> *);
  int service_2(Msg_Entry_Header *);

  /* and many other if you want */

 public:
  Any_Sever(/* parameters for this class */) :
            Remote_Multiserver_Point<Any_Server>(NO_OF_SERVICES, this)
  {
   start_deamon();

   add_service(MY_SERVICE_1, "description service 1", &Any_Server::service_1);
   add_service(MY_SERVICE_2, "description service 2", &Any_Server::service_2);
  }
};
```

Una vez que ha sido instanciado `Any_Server` es posible empezar a enviar mensajes a otras instancias de `Remote_Multiserver_Point`.

La forma que tienen los servicios, se muestra a continuación.

```
int Any_Server::service_1(Msg_Entry_Header * input_msg,
                Remote_Multiserver_Binding<Any_Server> * return_point)
{
 // cast input_msg (1)
 Input_Type * itp = static_cast<Input_Type *>(input_msg);

 // do whatever work has to be done. (2)

 // prepare response to client (3)
 Output_Type ot(STATUS_CODE_1);

 // respond to client (4)
 return_point->rpc_reply(&ot, sizeof(Output_Type));

 return 0;
}
```

Hay cuatro partes en un servicio con respuesta: la transformación (*casting*) del mensaje de entrada (1), el conjunto de instrucciones que procesan el mensaje y dan la forma al servicio (2), la preparación de la respuesta (3) y el retorno de la respuesta (4).

La clase también dispone del método `get_dispached_message_id()`, que retorna dentro de un servicio como `Any_Server::service_1` el identificador del mensaje despachado. Mediante la obtención de este identificador, un cliente de la biblioteca, puede llevar un control de estos mensajes.

4.3. Sistema de Servicios Distribuidos

Un modelo de servicios distribuidos puede realizarse de diversas formas. El modelo propuesto en esta sección no debe tomarse como la única manera de configurar un sistema de esta clase.

El siguiente modelo, basado en la arquitectura del sistema de localización, esta compuesto por procesos demonios y procesos clientes. Un demonio reside en cada máquina y atiende a diversos clientes locales. Existirán tantos clientes como los recursos permitan. Igualmente dentro de los recursos entra las limitaciones heredadas de las bibliotecas que se utilizan para la comunicación local y la remota.

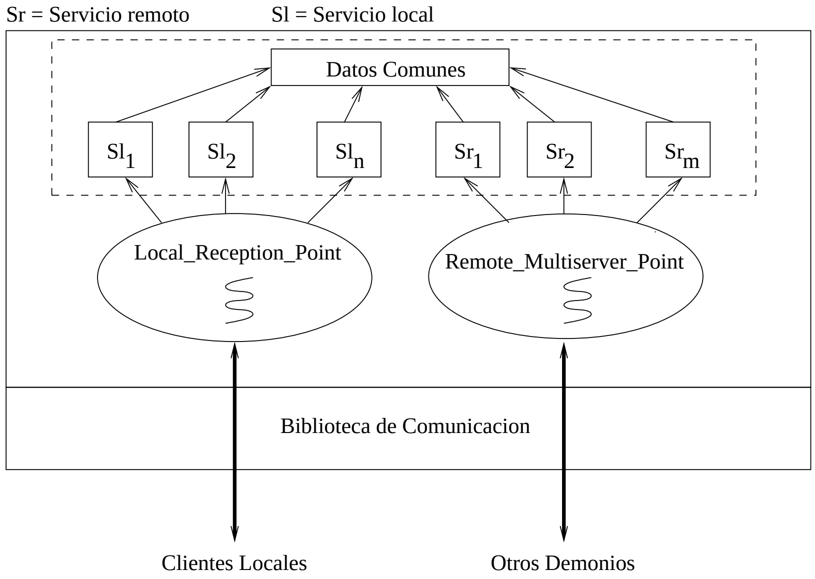

Figura 4.4: Forma de un demonio

Un demonio tiene la forma propuesta por la figura 4.4. Una instancia de servidor de comunicación asíncrona (`Remote_Multiserver_Point`) y otra del servidor comunicación síncrona (`Local_Reception_Point`) deben coexistir dentro del proceso demonio. La instancia del servidor para la comunicación asíncrona se utiliza para la comunicación entre los demonios, mientras que la instancia del servidor de comunicación síncrona se utiliza para comunicar los clientes locales con el demonio.

Bajo la perspectiva del modelo que se propone en esta sección, los clientes utilizan la información procesada por un conjunto de demonios. Para acceder a la información, los demonios exportan una interfaz que se conoce como llamadas al sistema. Una llamada al sistema hecha desde cualquier cliente hace una consulta a un demonio, luego este procesa local o distribuidamente la consulta y retorna una respuesta al cliente.

Los clientes (figura 4.5) por su parte deben tener una instancia de `Local_Access_Point` para que puedan comunicarse de manera confiable con el demonio que les sirve.

Para lograr entonces la consecución de una tarea distribuida, un cliente solicita la ejecución de un servicio a su demonio, y luego el demonio puede requerir de la ayuda de

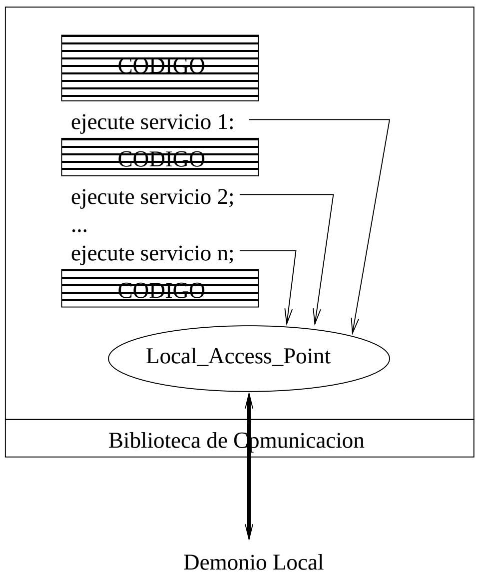

Figura 4.5: Forma de un cliente

otros demonios.

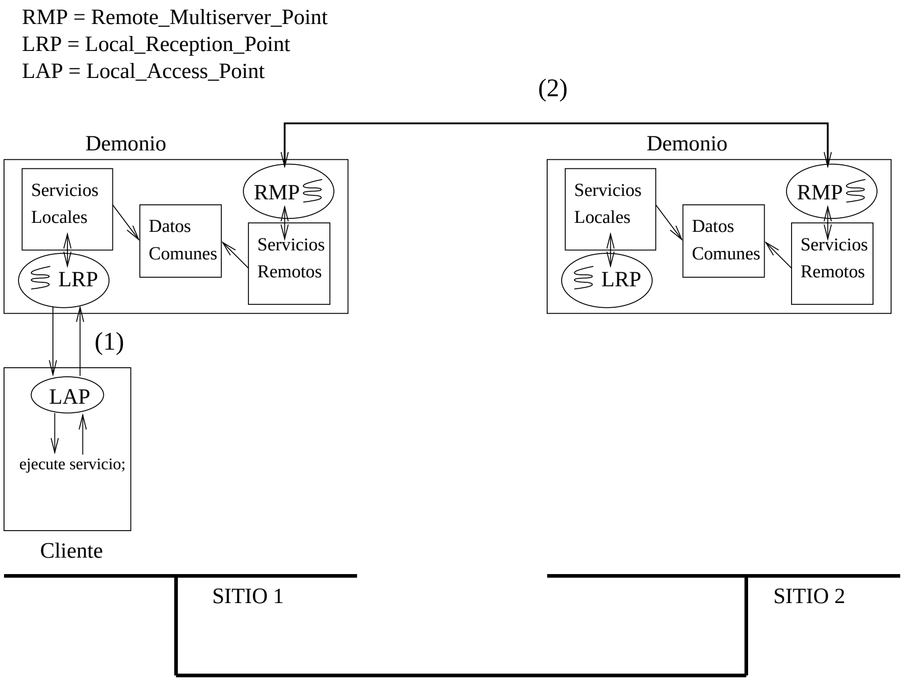

Figura 4.6: Interconexión cliente-demonio

Características	*Máquina 1*	*Máquina 2*
Nombre	brooks.cemisid.ing.ula.ve	dijkstra.cemisid.ing.ula.ve
IP	150.185.131.228	150.185.131.217
Arquitectura y Velocidad del Reloj	AMD 1.2 GHz	AMD 700 Mhz
Memoria RAM	256 Mb	128 Mb
Tarjeta de Red	10 Mb	10Mb

Cuadro 4.1: Datos de las máquinas donde se realizaron las medidas

En la figura 4.6 se observa una configuración más detallada del modelo propuesto. También se observa el flujo de los mensajes cuando un cliente solicita la ejecución de un servicio al demonio. En (1) la consulta local cliente/demonio y en (2) la consulta remota demonio/demonio.

4.4. Desempeño

Para evaluar el desempeño de los de servicios distribuidos, se compararon las clases de comunicación síncrona local y remota, y la clase de comunicación asíncrona remota con los mecanismos de comunicación equivalentes. Para las clases de comunicación remota síncrona y asíncrona, se utilizó el sistema IPC [12] para transmitir los mensajes, y para la clase de comunicación síncrona, se utilizó **Unix Domain Sockets** .

En general, los pares cliente/servidor comparados: `local_access_point` / `local_reception_point`, `remote_access_point` / `remote_reception_point` y, `remote_multiserver_point` / `remote_multiserver_point`, ofrecen el despachado de mensajes a funciones. Esto es una ganancia extra para la organización de sistemas orientados a servicios distribuidos, que además, como se podrá observar en las tablas de medidas, no ofrece un recargo significativo respecto al estilo de comunicación más básico.

En la tabla 4.1 se observan los datos donde se realizaron las medidas que se mostrarán en esta sección. Para el caso de la comunicación local se utilizo solamente la "*Máquina 1*". Mientras que para el caso de la comunicación remota se utilizó la "*Máquina 1*" para la ejecución de los clientes y la "*Máquina 2*" para los servidores.

El experimento consistió en enviar 100000 mensajes de solicitud y respuesta con un tamaño de 16 bytes para ambos. El tiempo en todos los casos fue medido desde el lado del cliente de la siguiente manera:

```
tiempo_inicial = lectura_de_reloj();
envio(msg);
recepcion(buffer);
tiempo_final = lectura_de_reloj();

dif = tiempo_final - tiempo_inicial;
```

Variable	*Sin despachador (seg)*	*Con despachador (seg)*
$\bar{x}$	$13,723 * 10^{-6}$	$16,084 * 10^{-6}$
mín	$12 * 10^{-6}$	$13 * 10^{-6}$
máx	$396 * 10^{-6}$	$341 * 10^{-6}$
σ	$1 * 10^{-6}$	$1 * 10^{-6}$

Cuadro 4.2: Tiempo que toma la acción solicitud/respuesta en el sistema de comunicación síncrona local

Variable	*Sin despachador (seg)*	*Con despachador (seg)*
$\bar{x}$	$0,84859 * 10^{-3}$	$0,85837 * 10^{-3}$
mín	$0,706 * 10^{-3}$	$0,71 * 10^{-3}$
máx	$20,32 * 10^{-3}$	$14,26 * 10^{-3}$
σ	$0,354 * 10^{-3}$	$0,345 * 10^{-3}$

Cuadro 4.3: Tiempo que toma la acción solicitud/respuesta en el sistema de comunicación síncrona remota

En cada caso las funciones de "*envío*" y "*recepción*" fueron reemplazadas por sus correspondientes equivalentes para cada clase.

En la tabla 4.2 se pueden apreciar las medidas obtenidas del experimento para la clase `local_access_point` y `local_reception_point`; cuyas medidas están en la columna "*con despachador*". Nótese que en promedio, solo se hace un recargo del 17 % cuando se utilizan las clases de comunicación. El sobrecargo apreciado en el tiempo mínimo conseguido es del 8,3 %. Este último, sugiere una medida que si bien no se consigue en promedio, es físicamente posible. Es de hacer notar que si la comparación porcentual hecha pareciera elevada, la diferencia es de solo $1\mu seg$ para el mínimo y de $2,361\mu seg$ para el promedio.

En la tabla 4.3 se observan las medidas obtenidas al realizar el experimento con las clases `remote_access_point` y `remote_reception_point`; cuyas medidas están en la columna "*con despachador*". El incremento en promedio para el tiempo de envío y recepción de un mensaje fue del 1,15 %. Este porcentaje es mucho más bajo que el obtenido en el experimento de la tabla 4.2, pues como se puede apreciar, los tiempos de envío y recepción en la comunicación por red son al menos 1 orden de magnitud más grande que en la comunicación local.

Variable	*Sin despachador (seg)*	*Con despachador (seg)*
$\bar{x}$	$0,81717 * 10^{-3}$	$0,83795 * 10^{-3}$
mín	$0,700 * 10^{-3}$	$0,716 * 10^{-3}$
máx	$11,415 * 10^{-3}$	$129,32 * 10^{-3}$
σ	$0,322 * 10^{-3}$	$0,074 * 10^{-3}$

Cuadro 4.4: Tiempo que toma la acción solicitud/respuesta en el sistema de comunicación asíncrona remota

Finalmente, en la tabla 4.4 se tienen los resultados de las medidas efectuadas con la clase `remote_multiserver_point`; cuyas medidas están en la columna "*con despachador*". El incremento del tiempo promedio para el envío y la recepción de un mensaje a través de esta clase es de $2,54\,\%$.

El hecho de que los tiempos reportados para las clases "*con despachador*" son mayores, se debe a la implantación de la función de despachado de mensajes en cada caso consume un tiempo de computo extra. Este es un incremento en tiempo que debe pagarse, sabiendo que para realizar un sistema de servicios distribuidos, un mecanismo para despachar mensajes representa una ganancia en organización y reutilización de código.

Capítulo 5

Interfaz

5.1. Elementos básicos del localizador

El localizador debe utilizar ciertas clases que intervienen en la gestión de localización de los objetos. Estas clases son: `Object_Id`, `Process_Id`, `Site_Id`, `Locator`, `Binding` y `Message_Id`.

Las clases con sufijo "`_Id`", utilizan como clase base a `Uid`. La clase `Uid` fue diseñada para exportar identificadores únicos desde el demonio IPC [12]. Un identificador único esta formado por la concatenación de varios campos que eventualmente le darán unicidad al conjunto. El primero de los campos es la dirección IP de la máquina donde se genera el identificador. El segundo es un entero sin signo de 64 bits de longitud que hace de contador. A través del contador, se puede asignar un número único que en una máquina a 900MHz, en incrementos de una unidad y, en el peor de los casos, tardaría en repetirse 650 años aproximadamente. Por último, se tiene un número aleatorio de 64 bits, que nos pudiera salvar de incurrir en el caso anterior. Esto sirve para afirmar que en un lapso mayor al siglo, podremos confiar en tener identificadores únicos sin colisión.

También existe una clase llamada `Port`, cuya función es la de ofrecer puertos únicos para la comunicación remota. Un `Port` es un identificador único `Uid`, pero añade un identificador de puerto de 32 bits de longitud. A través de los puertos se pueden enviar mensajes de manera local ó remota.

Las estructuras `Object_Id` y `Process_Id` son identificadores únicos del tipo `Uid` que identifican objetos y procesos respectivamente. Un `Site_Id` es un puerto (`Port` [12]) que identifica al sitio.

Un `Locator` es lo que en el capítulo 3 llamamos una referencia. Entonces, esta clase esta formada por el par (`Object_Id`, `Site_Id`) que identifican el sitio donde reside el objeto y una estampilla de tiempo lógico que contará el número de migraciones que el objeto realice.

Finalmente, un `Binding` esta formado por un `Locator`, un `Process_Id` y un `Site_Id`, estos dos últimos, identifican al proceso y al sitio donde se encuentra la referencia.

Todas estas estructuras sientan las bases necesarias para utilizar el sistema de localización de objetos móviles.

5.2. Ejecución del demonio localizador

El demonio localizador es un programa que debe ejecutarse en cada sitio que sea parte del sistema. La ejecución se realiza con el comando `locator_daemon`. Además, pueden modificarse dentro del localizador ciertos valores; por ejemplo los tamaños de las estructuras de datos; así podrá obtenerse un desempeño distinto para cada configuración.

La sintaxis de utilización del demonio localizador es:

```
locator_daemon [OPCIONES ...]
```

Valores de los caches

Las opciones que se presentan a continuación modifican el tamaño de cualquiera de los caches existentes. El tamaño de los caches se refiere al número máximo de registros que un cache pueda almacenar.

- `-i, --cache-in`: Fija la capacidad máxima de registros del cache de entrada.
- `-d, --cache-garbage`: Fija la capacidad máxima de registros del cache de eliminaciones.
- `-m, --cache-mig`: Fija la capacidad máxima de registros del cache de migraciones.
- `-o, --cache-out`: Fija la capacidad máxima de registros del cache de salida.
- `-w, --cache-new-bnds`: Fija la capacidad máxima de registros del cache de nuevas referencias.

Valores para los *piggybacks*

- `-a, --dead-age`: Fija la máxima edad lógica de un *piggyback*. Este es un número entero sin signo mayor que 1.
- `-c, --imp-cnt-size`: Fija un máximo al total de *piggybacks* que un sitio puede almacenar.
- `-p, --imp-per-chunk`: Fija el número máximo permitido de *piggybacks* por mensaje de invocación.
- `-e, --mtu-depend`: Si esta opción es especificada, entonces el total de un mensaje (mensaje del localizador más *piggybacks*)) no excederá el MTU de la red.
- `-t, --mtu`: El tamaño del MTU es por omisión encontrado por el localizador. Sin embargo, con esta opción se puede fijar un tamaño arbitrario del MTU en bytes.

Tamaños de las tablas hash del localizador

Los tamaños de las tablas hash se especifican en cantidad de registros.

- `-b, --obj-table`: Modifica el tamaño de la tabla propietaria de objetos.
- `-n, --cnt-table`: Modifica el tamaño de la tabla de procesos.
- `-r, --resp-table`: Modifica el tamaño de la tabla de las respuestas producto de las invocaciones.

Valores de las difusiones

- `-G, --attach-group`: Si esta opción es especificada, el demonio quedará suscrito al grupo de difusiones de búsqueda.

- `-l, --broadcast-table`: Fija el tamaño de la tabla donde se almacenan los mensajes de respuesta provenientes del servicio de difusión.

- `-D, --daemonize`: Hace que el `locator_daemon` se independice del shell donde fue ejecutado, es decir, se vuelva demonio. El proceso queda en background e independiente de `stdin`, `stdout` y `stderr`.

5.3. Llamadas al localizador

Una llamada al localizador es una funcionalidad que exporta el demonio localizador a través de una llamada a función.

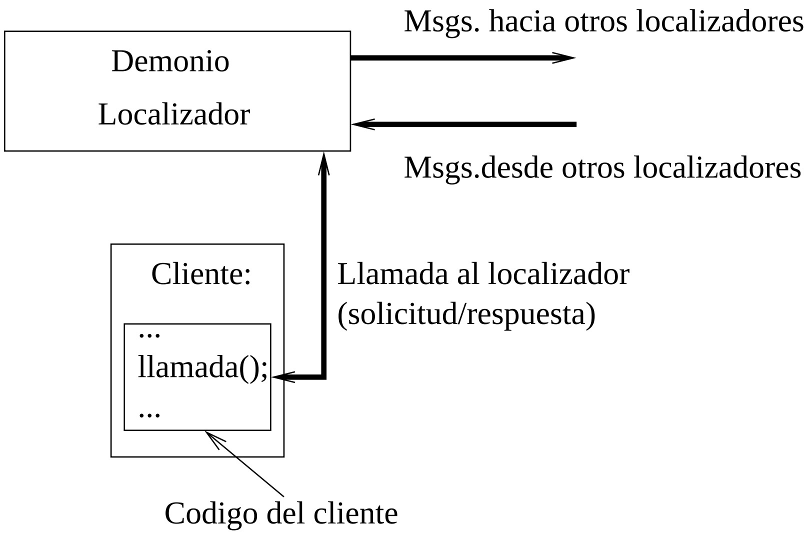

Figura 5.1: Llamada al demonio localizador

Como se puede apreciar en la figura 5.1, un cliente se comunica con el localizador para solicitar uno de sus servicios. Eventualmente, el localizador responderá al cliente indicando alguno de los estados resultantes de la ejecución del servicio.

El localizador exporta un total de 16 llamadas que conciernen a la localización de objetos. Según su funcionalidad, las llamadas se pueden clasificar en llamadas para registro y desregistro de procesos y objetos, migración de procesos y objetos, localización de objetos y actualización de referencias. Todas ellas son explicadas en detalle en lo que resta de esta sección.

5.3.1. Excepciones del localizador

Las llamadas al localizador se clasifican según su funcionalidad. Estas utilizan excepciones C^{++}para informar errores. Las excepciones fueron desarrolladas de manera genérica. Una excepción puede ser utilizada en múltiples situaciones donde su intención sea válida; lo que quiere decir que una excepción puede ser utilizada, sin ambigüedad, para dos semánticas distintas. Por ejemplo, la excepción `NotFound` puede indicar que un servidor no está en el puerto especificado, o también pudiera significar que un objeto solicitado no se encuentra registrado. Cada una de éstas semánticas podrá deducirse según la llamada donde fue generada la excepción.

Pueden ocurrir otras excepciones producto de las funciones de bibliotecas de uso estándar. Las excepciones utilizadas en el localizador son derivadas de la excepción `AlephExc`, quien a su vez deriva de la excepción base `std::exception` de la biblioteca STL.

De este modo, pueden ocurrir otro tipo de excepciones distintas a las especificadas en las firmas de las funciones del localizador. Entonces, mediante la excepción `std::exception`, se sabrá de que excepción se trata.

5.3.2. Registro de objetos y procesos

Mediante el registro de objetos y procesos, se hace saber al localizador del sitio acerca de los objetos y procesos que intervienen en el sistema. Así, el localizador podrá incorporarlos a sus estructuras y a los procesos de invocación, migración y búsqueda.

Se dispone de cuatro primitivas, un par para registro y desregistro de un objeto y otro par para el registro y desregistro de un proceso:

```
Process_Id register_prc();

void unregister_prc(const Process_Id & prc_id) throw(NotFound, RefusedService);
```

Este primer par de llamadas se refiere al registro de procesos. La llamada `register_prc` registra un proceso cuyo identificador, generado por el localizador, es el valor de retorno.

La llamada `unregister_prc` indica al localizador que el proceso especificado en `prc_id` será eliminado del sistema. La excepción `NotFound` indica que el identificador de proceso `prc_id` es inválido. La excepción `RefusedService` indica que el proceso a eliminar se encuentra procesando invocaciones, por lo que su eliminación no es posible; nuevos intentos serán entonces necesarios.

Las llamadas para el registro y desregistro de objetos son:

```
Object_Id register_obj(const Process_Id & prc_id) throw(NotFound);

void unregister_obj(const Object_Id & obj_id) throw(NotFound);
```

Mediante `register_obj` se registra un objeto. El valor de retorno de la función, es un identificador de objeto que devuelve el localizador. El proceso donde el objeto reside debe ser especificado en `prc_id`. Nótese que `prc_id` es un identificador válido de proceso, que ha sido devuelto por la llamada `register_prc`. La excepción `NotFound`, será arrojada cuando el identificador de proceso especificado en `prc_id` sea inválido.

Mediante `unregister_obj` se elimina un objeto del sistema. El identificador del objeto existente es especificado en `obj_id`. La excepción `NotFound` será arrojada cuando este identificador sea inválido o no exista. La excepción `RefusedService` será arrojada cuando el objeto se encuentre procesando alguna invocación, en cuyo caso no será interrumpido. Al obtener esta excepción, varios intentos deben ser hechos para desregistrar al objeto.

5.3.3. Migración de objetos y procesos

El proceso de migración consiste en la movilización de un objeto o un conjunto de objetos (proceso) de un sitio a otro. El sitio desde donde salen los objetos lo denominaremos sitio origen y el sitio donde llegan los objetos lo denominamos sitio destino.

Las llamadas que exporta el localizador no ofrecen ningún tipo de mecanismo de sincronización. Esto es dejado como responsabilidad del cliente del localizador; es decir, cada proceso debería tener un gestor de migración para los objetos. Este gestor se encarga de registrar correctamente el objeto en el sitio destino y de desregistrar al objeto en el sitio origen.

La migración de objetos esta dirigida por las llamadas siguientes:

```
void src_unreg_mig_obj(const Object_Id & obj_id,
                       const Site_Id & tgt_site_id)
   throw(NotFound, ObjectBusy);

void tgt_reg_mig_obj(const Object_Id & obj_id,
                     const Process_Id & prc_id,
                     const Logical_Timestamp timestamp)
   throw(NotFound, Duplicated);
```

La llamada `src_unreg_mig_obj` es invocada del lado origen de la migración y se encarga de eliminar al objeto especificado en `obj_id` de la tabla de objetos y, además, de registrar el evento en el cache de migración, para lo que utiliza la información suministrada acerca del sitio destino `tgt_site_id`.

Si la llamada culmina con éxito, ninguna excepción será arrojada. De otra manera, pueden ser arrojadas las excepciones `NotFound` u `ObjectBusy`. La excepción `NotFound` será arrojada en el caso en que el objeto `obj_id` no se encuentre en la tabla propietaria. La excepción `ObjectBusy` será arrojada cuando el objeto que se desea migrar se encuentra procesando alguna invocación. Nuevos intentos serán necesarios hasta poder migrar el objeto.

La llamada `tgt_reg_mig_obj` es utilizada en el sitio destino de la migración. Sirve para señalar la llegada de un objeto que está migrando.

El registro del objeto en el sitio destino de la migración puede fallar cuando el identificador del objeto `obj_id` suministrado está duplicado. En este caso, la excepción `Duplicated` será arrojada. Otra falla ocurre cuando el identificador de proceso `prc_id` es inválido. En este caso, la excepción `NotFound` es arrojada.

En la migración de objetos pudiera ser útil proveer un mecanismo de sincronización entre procesos. Esto se traduce en sincronizar: desregistrar el objeto del sitio origen después de registrarlo en el sitio destino.

La migración de procesos se hace de forma muy similar a la de objetos. En este sentido, se aprovecha la secuencia de eventos que ocurren en la migración de objetos. De manera simple, la migración de procesos es la migración de un conjunto de objetos. Las llamadas para llevar a cabo este procedimiento son:

```
void src_unreg_mig_prc(const Process_Id & prc_id,
                       const Site_Id & tgt_site_id)
   throw(NotFound);

void tgt_reg_mig_prc(const Process_Id & prc_id)
   throw(Duplicated);
```

Cuando se desea migrar un proceso, debe primero invocarse en el sitio origen de la migración la llamada `src_unreg_mig_prc`. En esta llamada se especifica el identificador del proceso a migrar `prc_id` y el sitio destino de la migración `tgt_site_id`.

Luego, para cada uno de los objetos residentes en el proceso migrante, queda su registro en el cache de migración. Los eventos que suceden en el sitio origen, son los mismos eventos descritos en la migración individual de objetos.

En caso de que el proceso especificado en `prc_id` sea inválido, la excepción `NotFound` será disparada.

La llamada `tgt_reg_mig_prc` crea el proceso en el sitio destino. Luego de tener el proceso creado, los objetos pueden copiarse asíncronamente. La excepción `Duplicated` será arrojada en caso de que el identificador de proceso se encuentre repetido.

5.3.4. Invocación remota

La invocación remota se implanta con tres llamadas: `multi_receive`, `clt_invoke_send` y `srv_invoke_reply`.

```
Message_Id multi_receive(Binding & binding,
                         const Process_Id & receiving_process_id,
                         void * data,
                         size_t & data_size,
                         Reception_Type & reception_type)
   throw (NotFound, ObjectDead, RecentBinding);
```

A través de esta llamada se reciben dos tipos de mensajes: las invocaciones y las respuestas a las invocaciones. Esto implica que la llamada ha de ser utilizada tanto en un proceso cliente como en uno servidor. Esta llamada retorna un identificador único de mensaje `Message_Id` que identifica al par de mensajes de RPC.

Para distinguir un mensaje de invocación de otro de respuesta, se usa el parámetro `reception_type`. Al inspeccionar su valor, encontraremos `RECTYPE_INVOCATION_REPLY` si se trata de la respuesta a una invocación y `RECTYPE_INVOCATION_REQUEST` si se trata de la propia invocación.

La llamada `multi_receive` está diseñada para ser utilizada por un despachador en el proceso servidor y puede ser utilizada desde una *thread* independiente. Este despachador es parte del localizador contenedor. Entre otras funciones, se encarga de registrar todos los objetos residentes en el proceso y guarda las respuestas producto de las invocaciones.

En la variable `binding` puede obtenerse toda la información referente a una invocación entrante o a la respuesta de una invocación. Esto depende del tipo de mensaje procesado.

Si se trata de un `RECTYPE_INVOCATION_REQUEST`, entonces `binding.get_process_id()` y `binding.get_site_id()` identifican al proceso y al sitio desde donde proviene la referencia que será utilizada para devolver la respuesta. En `binding.get_locator()` viene la referencia al objeto invocado. La invocación es copiada en `data` y su tamaño en `data_size`.

Cuando se trata de un `RECTYPE_INVOCATION_REPLY`, la respuesta a la invocación se copiará en `data` y su tamaño en `data_size`.

Una anomalía en la referencia `binding.get_locator()` podrá conocerse a través de las siguientes excepciones:

- `NotFound`: si el objeto que fue invocado no se encontraba en el sitio que decía la referencia.
- `RecentBinding(Binding & binding)`: que informa sobre una referencia más reciente que se conseguirá en `binding.get_locator()`. Si en el proceso de invocación se detectó una referencia más nueva, entonces se debe repetir la invocación.
- `ObjectDead`: si el objeto ha sido eliminado del sistema.

Finalmente, tanto para la recepción de invocaciones como de respuestas, la variable `receiving_process_id` hace las veces de una capacidad que se utiliza para indicar el proceso que ejecuta la llamada. Ante el localizador, esta identificación indica que el proceso está autorizado para recibir mensajes.

Envío de invocaciones y respuestas

```
Message_Id clt_invoke_send(Binding & binding,
                           const void * request,
                           size_t request_size)
   throw (ObjectDead);
```

Con esta llamada se inicia una invocación. Sus parámetros son: el `binding` que especifica la dirección del objeto a invocar, el `request` que es la dirección de memoria de los datos que se van a enviar y su respectivo tamaño especificado en `request_size`.

La excepción `ObjectDead` será arrojada en el caso de que el objeto solicitado haya sido eliminado.

```
void srv_invoke_reply(const Message_Id & msg_id,
                      const Binding & binding,
                      const Process_Id & replying_prc_id,
                      const void * reply,
                      size_t reply_size);
```

Mediante `srv_invoke_reply` se puede responder a una invocación. Para esto debe especificarse el identificador del mensaje `msg_id` suministrado por la llamada `multi_receive`, el `binding` de la invocación, el identificador del proceso `replying_prc_id` que aloja al objeto que responde y la respuesta `reply` con su respectivo tamaño `reply_size`.

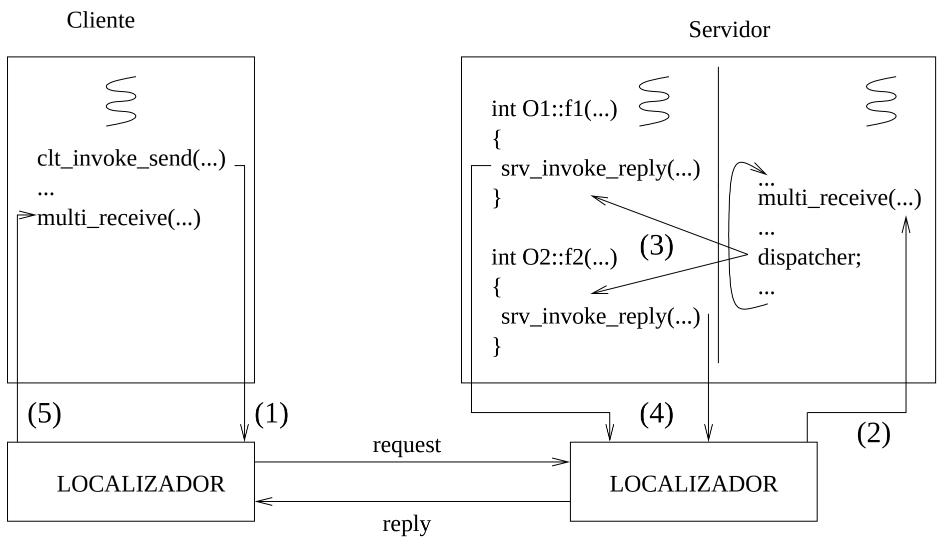

Figura 5.2: Uso de las llamadas de invocación

En la figura 5.2 se puede apreciar como utilizar las llamadas de invocación.

Del lado del cliente se utiliza la llamada `clt_invoke_send` para realizar una invocación (1) a objeto. Tanto la llamada `clt_invoke_send` como la llamada `multi_receive`, pueden estar en una sola *thread* si en el proceso no hay objetos que puedan ser invocados.

En el lado del servidor, la llamada `multi_receive` es utilizada en una *thread* separada, pues esta llamada es bloqueante. `multi_receive` se bloquea a la espera de invocaciones (2) las cuales puedan ser despachadas (3) hacia cualquiera de los métodos de los objetos registrados.

Una vez que es procesada la solicitud, se responde desde el método invocado (4). El cliente se encuentra a la espera de la respuesta con la llamada `multi_receive` (5).

5.3.5. Búsqueda de objetos

Dado un identificador de objetos, una función de búsqueda será capaz de encontrar una referencia `Locator` válida. El siguiente par de llamadas cumple esta función. Su diferencia estriba en el costo que implica hacer cada una.

```
Locator strong_locate(const Object_Id & obj_id) throw(ObjectDead);
```

La llamada `strong_locate` envía una difusión para localizar el objeto solicitado. Esta difusión consiste de un mensaje por sitio con su respectivo reconocimiento. El mensaje de difusión será recibido por todos o por ninguno de los localizadores.

Si la excepción `ObjectDead` es arrojada, se puede concluir que el objeto ha sido eliminado. De no haberse disparado ninguna excepción, el objeto se encontrará en el `Locator` que devuelve la llamada.

Para una difusión de menor costo pero de envío no confiable de mensajes se tiene:

```
Locator weak_locate(const Object_Id & obj_id) throw(NotFound, ObjectDead);
```

Esta llamada, similar a la función `strong_locate`, difunde en dos etapas, mediante el envío no confiable de mensajes, dos tipos de solicitudes de búsqueda del objeto.

La primera etapa consiste en difundir un mensaje preguntando si el objeto está en la tabla propietaria. La segunda consiste en difundir un mensaje preguntando por información que se encuentre en los caches.

Cada etapa tiene un tiempo de expiración para aguardar por la respuesta. Si la segunda etapa no tiene éxito, entonces se concluye que el objeto no fue encontrado; en este caso se dispara la excepción `NotFound`. En este caso no es posible concluir que el objeto fue eliminado, pues no se asegura que todos los mensajes hayan llegado a su destino.

Existe, sin embargo, la posibilidad de que se detecte la inexistencia del objeto a través de alguno de los caches de eliminación, en este caso, la excepción `ObjectDead` será arrojada.

5.3.6. Búsqueda de objetos mediante *piggybacking*

Para utilizar la actualización por *piggybacking*, los mensajes para la solicitud de objetos, pueden se incluidos según la política que el cliente decida. Esto se hace a través de la siguiente llamada:

```
void piggyback_locate(const Binding & old_binding);
```

La llamada `piggyback_locate` instruye al localizador para que en las próximas invocaciones, se les incluya un *piggyback* del tipo "*se busca*", para un objeto cuyo último binding conocido es `old_binding`.

Luego, el cliente deberá revisar el estado del `locator` a través de la llamada:

```
void test_location(Locator & locator) throw (NotFound, ObjectDead);
```

La llamada `test_location` se encarga de actualizar y verificar, localmente, el `locator` especificado. Luego de ejecutar la llamada, es posible que se arrojen las excepciones `NotFound` si el objeto no fue encontrado en ninguno de los caches, u `ObjectDead` si se encontró en el cache de eliminados.

5.3.7. Prefetching

La única forma en que el cliente puede hacer prefetching es con el cache de salida. Así, el cliente puede implantar su propio criterio y adaptarlo a sus necesidades:

```
void ping(unsigned int number_of_entries, const Cache_Update_Policy policy)
   throw (NotFound);
```

Mediante `ping` se comprueba la validez de un número de entradas (`number_of_entries`) en el cache de salida. La comprobación consiste en enviar un mensaje de verificación en la dirección que apunta cada referencia. Luego, una respuesta será devuelta con información acerca del estado del objeto.

Esta comprobación es realizada a las entradas del cache de salida que se encuentran ubicadas en una de las tres zonas especificada en `policy`: las entradas del principio (`LOCCACHE_MRU`), o mayormente conocida como las mas recientemente utilizadas, las entradas del medio (`LOCCACHE_MIDDLE`), o las entradas del final (`LOCCACHE_LRU`) conocidas también como las menos recientemente utilizadas.

La excepción `NotFound` es arrojada en el caso de que el cache de salida se encuentre vacío.

Los otros tipos de prefetching explicados en la sección 3.5 son manejados interna y transparentemente por los localizadores.

Capítulo 6

Implementación

En este capítulo se describe brevemente la implementación del sistema: los caches, las tablas y la gestión de *piggybacks*. También se exponen los detalles del diseño y la interconexión de los distintos módulos que componen al servicio de localización.

6.1. Arquitectura del servicio de localización

Existen dos alternativas para la implementación del localizador: como un modulo más del sistema IPC, o como un servicio independiente que utiliza un sistema IPC para el envío de mensajes. Este último puede ser visto como una capa superior a la de comunicación.

El localizador de objetos dentro del sistema IPC resulta en un sistema menos cohesionado, pues se tendría un servicio IPC que también se encargaría de la localización de objetos. El servicio IPC tendría un menor rendimiento, pues la coexistencia con el localizador implica procesamiento extra.

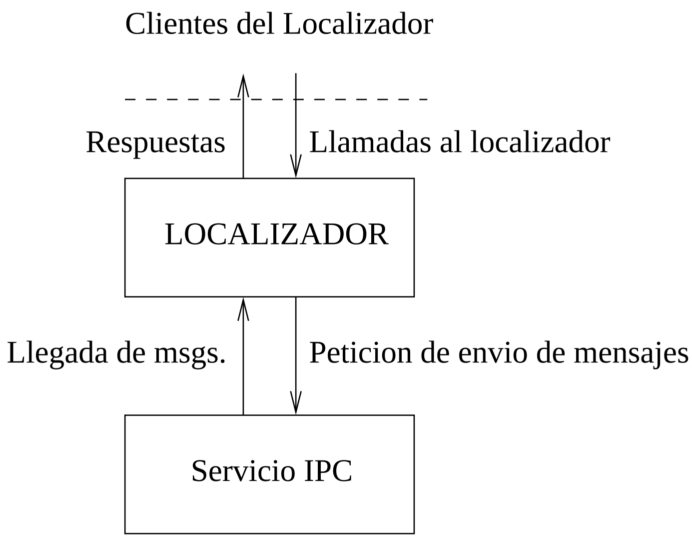

Figura 6.1: Interacción IPC - Localizador

El abordar por separado el IPC y la localización e invocación de objetos, nos permite modificar cada programa independientemente. Resulta también en un código más mantenible y, más importante, un código con un solo propósito.

El localizador llevará cuenta de los procesos y objetos residentes en el sitio. Los localizadores podrán interactuar entre sí a través de la transmisión confiable de mensajes hecha por el IPC.

En la figura 6.1 se muestra la relación entre el sistema IPC y el localizador. Los clientes del localizador se comunican con éste a través de llamadas que serán procesadas y pasadas al IPC para el envío confiable de mensajes. Visto desde el lado receptor, el servicio IPC pasa los mensajes al localizador y éste, eventualmente, pasará la respuesta al cliente.

6.2. Estructuras de datos utilizadas

6.2.1. Caches

Un cache es una estructura finita de recuperación de datos. Puesto que un cache es finito, éste se llena. Cuando esto ocurre, es necesario reemplazar un elemento para dar entrada al nuevo. En nuestra implementación se utilizó la política de reemplazo del menos recientemente utilizado.

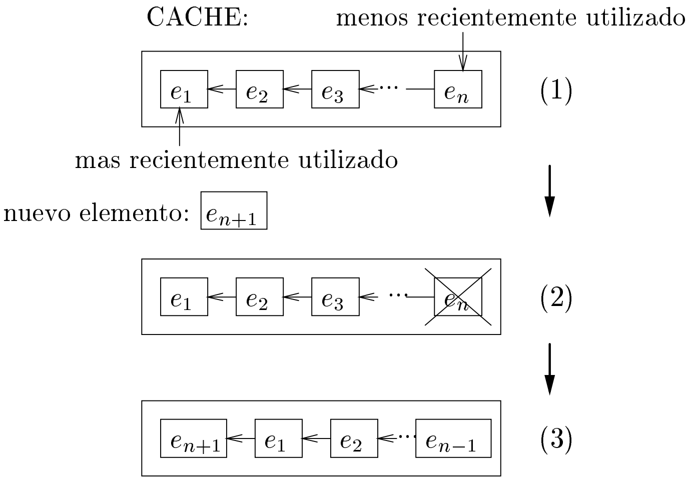

Figura 6.2: Esquema del funcionamiento del cache

En la figura 6.2 se esboza de forma general el funcionamiento de un cache. Cada vez que un elemento es accedido, pasa a ser el más recientemente utilizado. Si se trata de una eliminación, el elemento menos recientemente utilizado (1) será eliminado a la llegada de un nuevo elemento e_{n+1} (2); luego este elemento pasará a ser el más recientemente utilizado (3).

La indexación de los elementos del cache se hace a través de una tabla hash. De esta manera el orden de búsqueda y eliminación de un elemento específico es $O(1)$. Las entradas en la tabla son encadenadas en una cola, según el acceso, que simula la política LRU.

Caches en el localizador.

Los caches, son aquéllos advertidos en el capitulo 3:

`cache_out`: Guarda las referencias de las invocaciones salientes. Cada registro está formado por un `Locator`.

`cache_in`: Guarda la dirección de los sitios desde donde se invoca a los objetos locales. Cada registro está formado por un `Site_Id`, un `Object_Id` y un `Logical_Timestamp`. El `Site_Id`, es el identificador del sitio desde donde ha ocurrido una invocación. El `Object_Id` es el identificador del objeto invocado y el `Logical_Timestamp` es la última estampilla de tiempo lógico conocida.

`cache_mig`: Guarda las nuevas referencias de objetos que han migrado. Cada registro está formado por un `Locator`.

`cache_new_bnds`: Las referencias almacenadas en este cache, son aquéllas que han sido conocidas a través de los *piggybacks*. Un registro en este cache esta formado por un `Locator`.

`cache_garb`: Guarda los identificadores de objetos que han sido eliminados del sistema. Un registro de este cache está formado por un `Object_Id`.

6.2.2. Tablas hash

Las tablas hash del localizador hacen la resolución de colisiones a través del método de encadenamiento directo.

Las tablas utilizadas en el localizador son las siguientes:

`objects_table` : Esta tabla almacena la información de los objetos registrados. Esta es la llamada *"tabla propietaria"* en el capítulo 3. La clase del registro almacenado en esta tabla tiene los siguientes campos:

```
Object_Id         object_id;
Dlink             link_to_process;
Process_Id        associated_process_id;
Logical_Timestamp logical_timestamp;
State_Of_Object   current_state;
```

El identificador único del objeto está en `object_id`. Cada proceso tiene una lista de objetos, de esta forma podrá llevar cuenta de los objetos que le pertenecen. Así, la variable `link_to_process` es un lazo a la lista de objetos del proceso especificado en `associated_process_id`. La variable `logical_timestamp` es una estampilla de tiempo lógico que guarda el número de veces que un objeto ha migrado. Por último, tenemos la variable `current_state`, que indica el estado actual de un objeto: inactivo (`OBJ_IDLE`), procesando una invocación (`OBJ_PROCESSING_INVOCATION`) ó en estado de migración (`OBJ_MIGRATING`).

`process_table` : La tabla de procesos guarda la información de los procesos registrados. Su principal función es llevar cuenta de los objetos que agrupa. La clase del registro almacenado en esta tabla tiene los siguientes campos:

```
Process_Id            process_id;
pid_t                 p_pid;
Dlink                 associated_objects_list;
Binding_Point *       invocation_return_point;
Dnode<Reception_Type> waiting_invocation_list;
unsigned long long    pending_replys;
State_Of_Process      current_state;
```

El identificador único del proceso a lo largo del todo el sistema está en `process_id`, mientras que en `p_pid` se guarda el identificador de proceso suministrado por el sistema operativo a través de la llamada a sistema `getpid()`. La variable `associated_objects_list` sirve de lazo a la lista de objetos. Por medio de este lazo se tiene acceso a los objetos que el proceso contiene. La utilidad principal de esta lista es durante la migración de procesos, pues se tiene un acceso rápido a cada uno de los objetos.

La variable `invocation_return_point` representa la dirección de retorno del cliente (ver § 4.2.1.2). Por medio de esta dirección se entregan las invocaciones y las respuestas a las invocaciones.

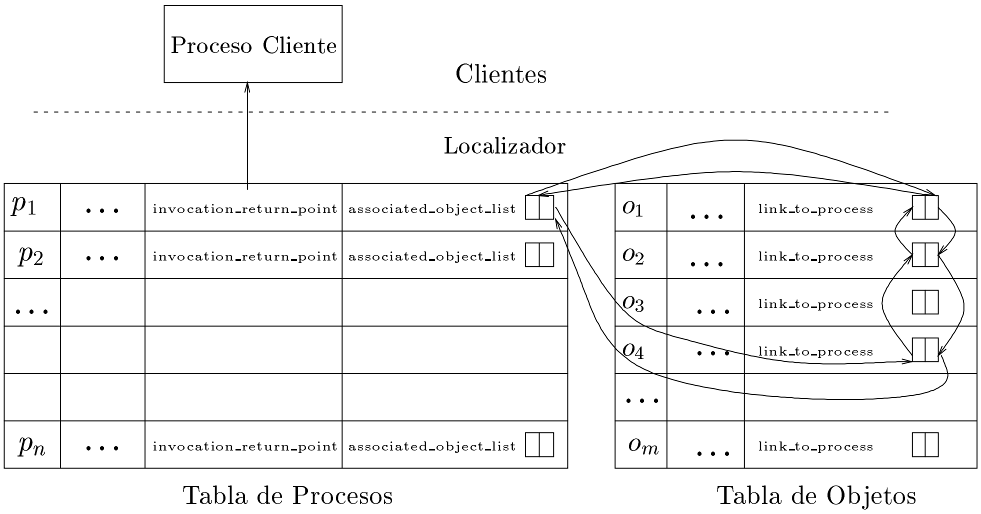

Figura 6.3: Interconexión de las tablas de procesos y objetos

Desde la perspectiva de diseño del localizador, un proceso puede despachar una invocación de objeto a la vez. Esto hace que se tenga la lista `waiting_invocation_list`, que se encarga de retener invocaciones concurrentes. El número de invocaciones pendientes es almacenada en la variable `pending_replys`.

Un proceso puede tener varios estados que son almacenados en `current_state`. Los estados de un proceso son: listo para recepción (`PRC_READY`), invocación siendo procesada (`PRC_PROCESSING_INVOCATION`) y proceso no apto para recibir (`PRC_NOT_RECEIVING`).

En la figura 6.3 se aprecia la manera en que se relacionan la tabla de objetos y de procesos. En el proceso p_1 se encuentran los objetos o_1, o_2 y o_4. Esto se observa siguiendo la conexión de los lazos `link_to_process`. El campo `associated_object_list` sirve de nodo centinela en la lista de objetos, mientras cada uno de los campos `link_to_process` sirve para conectar un objeto a la lista.

`process_answer_table` : Mediante esta tabla se guardan las direcciones de retorno de los procesos remotos que están esperando por la respuesta de una invocación. En la figura 6.4 se aprecia la utilidad de esta tabla durante el proceso de invocación.

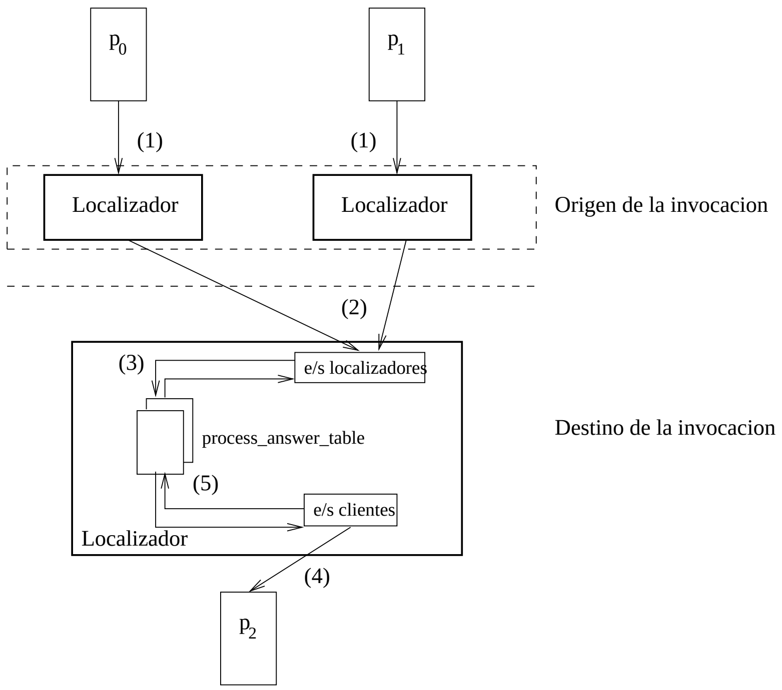

Figura 6.4: Uso de la tabla `process_answer_table`

En una primera etapa los procesos p_0 y p_1 envían invocaciones (1), a través de los respectivos localizadores. Una vez alcanzado el destino de las invocaciones (2), el servicio de invocación en el localizador destino ingresa (3) un par de registros correspondientes a los procesos invocantes (p_0 y p_1). Después, pasa la invocación al proceso que contiene al objeto invocado (4). Luego de que p_2 procesa la invocación, el localizador busca los registros de p_0 y p_1 que fueron almacenados en la tabla `process_answer_table` para obtener la dirección de retorno (5).

La clase del registro almacenado en esta tabla tiene los siguientes campos:

```
Message_Id message_id;
Process_Id process_id;
Remote_Multiserver_Binding<Locator_Remote_Services> * return_point;
```

La variable `message_id` se utiliza para identificar el par solicitud/respuesta en la llamada RPC. La variable `process_id` es el identificador del proceso que realiza la invocación. Este proceso se encuentra remotamente registrado en el localizador, cuya dirección es obtenida a través de `return_point`.

6.3. Implementación del servicio de localización

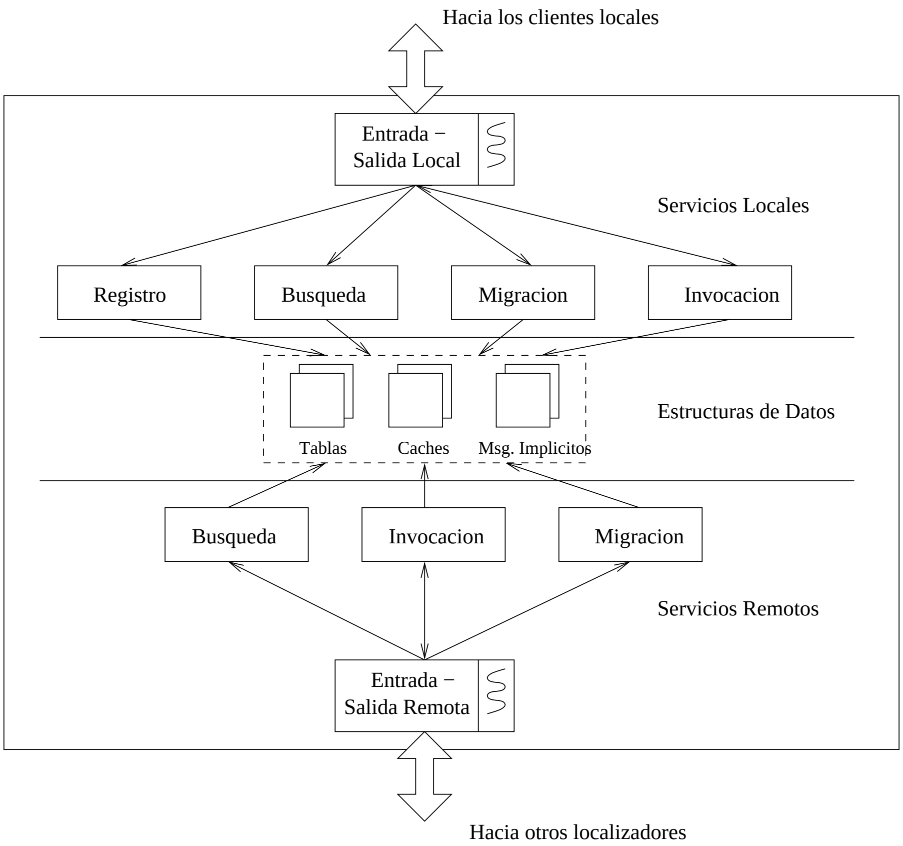

Figura 6.5: Arquitectura del localizador

En la figura 6.5 se esboza la arquitectura general del localizador. Esta arquitectura está basada en la arquitectura de los servicios genéricos distribuidos expuesta en el capítulo 4.

El módulo de entrada y salida local, es un `Local_Reception_Point` (ver § 4.2.1.6) y, su función es la de comunicar los clientes locales con el localizador. Nótese que este módulo posee una *thread* de control independiente. Así, se garantiza la recepción continua de mensajes locales (llamadas al localizador). Este módulo despacha los servicios del localizador accedido a través de las llamadas de registro, búsqueda, migración e invocación que fueron explicadas en el capítulo 5.

El módulo de entrada y salida remota es un `Remote_Multiserver_Point` (§ 4.2.2.2). Su función es comunicar al localizador con otros localizadores a través de mensajes explícitos, los cuales serán descritos en la siguiente sección. Este módulo tiene una *thread* de control independiente, que garantiza la continuidad en la comunicación. También atiende solicitudes de búsqueda, invocación y migración de otros localizadores.

Todo el conjunto de servicios locales y remotos que posee el localizador puede enviar mensajes a través de la salida local y remota. Esto quiere decir que la invocación, que originalmente es producto de una llamada al localizador (e-s local), se convierte en una solicitud de invocación remota. Esta solicitud será despachada por una función de envío remoto invocada desde el servicio local. Esto sucede también de forma inversa cuando una solicitud de invocación recibida por el módulo de entrada y salida remota, es despachada a un cliente local.

Los servicios del localizador tienen acceso completo a las estructuras de datos. De allí que se disponga de estructuras para la exclusión mutua al momento de accesar, modificar o eliminar algún registro de cualquiera de las tablas, de los caches o del contenedor de *piggybacks*.

6.3.1. Mensajes explícitos

Un mensaje explícito tiene como función anunciar un evento de un localizador a otro con la finalidad de realizar una invocación o de ejecutar una acción de localización. Estos mensajes son generados como consecuencia de la petición de alguno de los servicios que el sistema localizador exporta a los clientes. La figura 6.6 esboza la estructura de un mensaje del localizador.

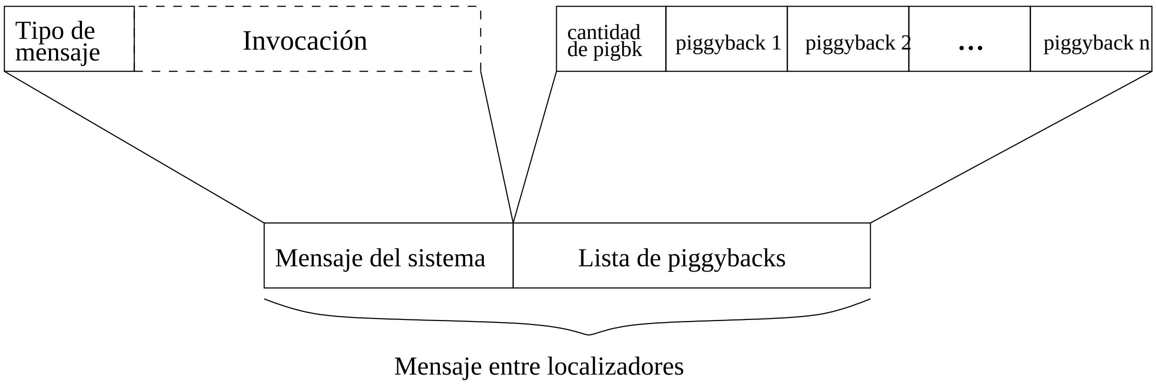

Figura 6.6: Formato de un mensaje del localizador

El tamaño de la sección de los *piggybacks* es determinado en tiempo de ejecución y según criterios de performance (ver § 5.2). Tal y como se muestra en la figura 6.6, un mensaje explícito, está formado por el mensaje propio del sistema de localización, más una parte destinada a los *piggybacks*.

Un mensaje del localizador tiene una cabecera donde se transporta información del sistema de comunicación. Seguidamente, tiene una región donde se transportan los datos pertinentes al tipo de mensaje: invocación, localización, prefetching, etc. Finalmente, viene la región donde se transporta la invocación, solo en el caso de los mensajes de invocación.

La lista de *piggybacks* tiene un primer campo que indica el número de *piggybacks* y, luego concatenados uno detrás del otro, cada uno de los *piggybacks*.

Se tiene un total de 8 mensajes explícitos que se describen a continuación:

1. **Solicitud de invocación:** Mediante este mensaje se hace la invocación a un objeto. Este mensaje está formado por un `Binding` y la invocación.

2. **Respuesta a la invocación:** Este mensaje sirve para transportar la respuesta a una invocación. Esta formado por un Binding, el tipo de respuesta y la respuesta a la invocación.

   El tipo de respuesta puede ser: invocación exitosa, objeto no encontrado, binding más reciente y objeto eliminado. Un tipo de respuesta diferente a invocación exitosa requiere tomar acciones pertinentes de localización, pues la invocación no ha sido llevada a cabo.

3. **Solicitud de búsqueda:** Es el mensaje enviado en una difusión. Esta formado por un `Binding`.

4. **Anuncio de inexistencia de un objeto:** Dice si un objeto ha sido eliminado del sistema. Lleva el identificador del objeto eliminado.

5. **Indagación de existencia:** Permite saber si un objeto todavía esta en el lugar donde dice su "*referencia*". Es el mensaje utilizado para hacer el prefetching con el cache de salida.

6. **Respuesta a la indagación:** Ofrece información acerca del estado del objeto indagado con el mensaje anterior. Es el mismo tipo de respuesta que fue descrito en el mensaje **Respuesta a la invocación**.

7. **Anuncio de referencia:** Este mensaje tiene como función llevar una referencia hacia cualquier sitio. El mensaje es utilizado en acciones de prefetching y difusión. Este mensaje trae un `Locator`.

8. **Registros del cache de entrada:** Lleva al localizador destino de la migración las entradas correspondientes al objeto en el cache de entrada del sitio origen de la migración. Esta formado por una lista de `Site_Id` que se refiere a las direcciones de los localizadores desde donde se hicieron invocaciones al objeto migrante.

En la tabla 6.1 se muestran los tamaños en bytes de los mensajes explícitos. El tamaño de los mensajes de solicitud y respuesta de invocación, dependen de s_i y s_r que son los tamaños de la invocación y de la respuesta a la invocación respectivamente. Otros tamaños están en función de la cantidad de elementos que se comunican en el mensaje (n). Para los mensajes de *indagación de existencia* y *respuesta a la indagación*, n representa el número de objetos a indagar y el número de objetos indagados, respectivamente. Mientras que para el mensaje *anuncio de referencia*, n indica el número de referencias que lleva el mensaje.

Mensaje	*Nombre de la clase*	*Tamaño (bytes)*
Solicitud de invocación	`Invocation_Request_Msg`	$148 + s_i$
Respuesta a la invocación	`Invocation_Reply_Msg`	$152 + s_r$
Solicitud de búsqueda	`Find_Owner_CallMsg`	40
Anuncio de inexistencia	`Dead_Anounce`	40
Indagación de existencia	`Ping_N_Objects_Request`	$24 + 56n$ $(n > 0)$
Respuesta a la indagación	`Ping_Reply_For_N_Objects`	$24 + 56n$ $(n > 0)$
Anuncio de referencia	`Locator_Anounce`	$44 + 32n$ $(n > 0)$

Cuadro 6.1: Tamaños de los mensajes explícitos

6.4. Registro de objetos y procesos

Las funciones de registro de objetos y procesos son la interfaz para el manejo de las tablas `objects_table` y `process_table`. Las funciones de registro o desregistro, ingresarán o eliminarán objetos o procesos de las respectivas tablas.

Al ingresar un proceso, se crea un registro en la tabla de procesos. En este momento, la lista de objetos está vacía. Al ingresar objetos posteriormente, se crean los registros en la tabla de objetos. Los objetos son pertinentemente enlazados en la lista de objetos de cada proceso, tal como se describió en § 6.2.2.

6.5. Invocación

Dentro de la invocación existen varios subprocesos que se encargan de resolver tareas que han podido ser modularizadas. Los dos subprocesos más importantes dentro de la invocación son el manejo de los *piggybacks* y la búsqueda en caches que a continuación se presentan.

6.5.1. Gestión de *piggybacks*

La gestión de *piggybacks* se realiza mediante una clase formada por un árbol binario hilado del tipo AVL y una tabla hash. Estas dos estructuras rigen la organización y el control de los *piggybacks*.

Para la gestión de *piggybacks* se consideran las siguientes premisas:

- Un *piggyback* que ya es conocido por un sitio, es descartado en los sucesivos grupos de mensajes hacia dicho sitio.

- Cada *piggyback* pertenece a un nodo etiquetado con una edad lógica. Luego, cada *piggyback* es promovido hacia otro nodo con mayor edad, cada vez que éste es revisado para ser enviado.

- Existe solo un *piggyback* por objeto, pues los *piggyback* siguen un orden de prioridad donde el más prioritario reemplaza al menos prioritario. Las prioridades vienen dadas según el tipo de *piggyback* si se trata de *piggybacks* distintos, donde, del más al menos prioritario son: *eliminado*, *se busca* y *nueva referencia*. Por otro lado, si se

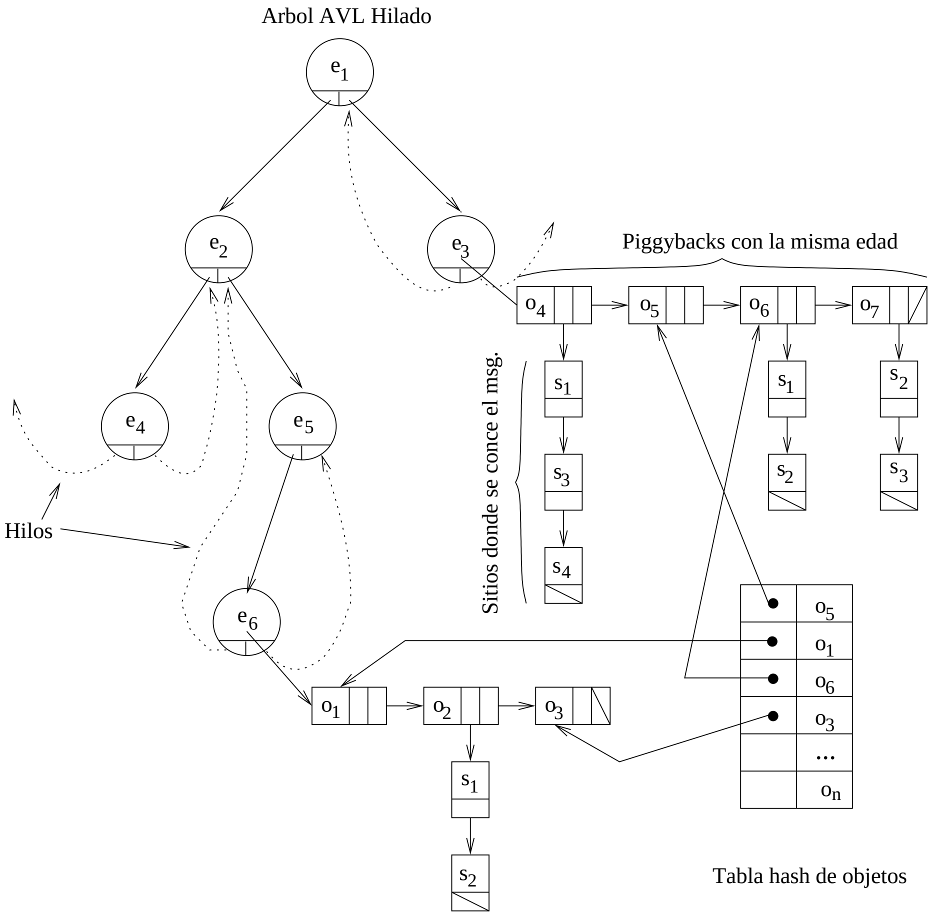

Figura 6.7: Estructura de datos para la gestión de *piggybacks*

Piggyback	*Nombre de la clase*	*Tamaño (bytes)*
eliminado	`Piggyback_Death_Anounce_Msg`	64
se busca	`Piggyback_Locator_Anounce`	96
nueva referencia	`Piggyback_Find_Msg`	144

Cuadro 6.2: Tamaños de los mensajes *piggyback*

tienen *piggybacks* iguales y el nuevo *piggyback* trae información más reciente, éste es reemplazado y se colocado en un nodo con estampilla lógica cero.

- Los *piggybacks* de la misma edad son insertados en una lista doble en un nodo del árbol según un orden FIFO.

- Para escoger k *piggybacks*, se comienza por el nodo de menor edad. Luego, se escogen uno a uno los *piggybacks* encolados según el orden FIFO en dicho nodo. Se consideran, entonces, sólo aquellos que no han sido enviados hacia el sitio destino. Luego, si un *piggyback* es escogido, inmediatamente el identificador de sitio destino es agregado a la lista de sitios conocidos por el *piggyback*. De esta misma forma, se procederá según el orden creciente de la edad con los siguientes nodos del árbol hasta completar los k mensajes.

En la figura 6.7 se aprecia la estructura de datos que gestiona los *piggybacks*. Los nodos del árbol tienen como campo clave la edad del *piggyback*; algunos de ellos son e_1, e_2, etc. En cada uno de estos nodos, existe una lista de *piggybacks* que tienen la misma edad. En la lista de *piggybacks* del nodo e_3, el *piggyback* referente al objeto o_7 fue el primero en llegar y será el primero en salir. Finalmente, el *piggyback* referente al objeto o_7, es conocido por los sitios s_2 y s_3.

Cada una de las estructuras utilizadas fue escogida para una función específica. Con el árbol AVL hilado, se pueden escoger, a través de un recorrido infijo y de espacio $O(1)$, los k *piggybacks* que viajaran en la invocación. El espacio que ocupa el recorrido se debe a que no se necesita utilizar una pila.

Por otro lado con la tabla hash de objetos, un *piggyback* referente a un objeto en particular puede ser actualizado en $O(1)$. De la misma manera, se puede saber en tiempo $O(1)$ si, para un objeto en particular existen mensajes.

Para la gestión de *piggybacks*, se exportan los siguientes métodos:

```
void add_piggyback_msg(const Piggyback_Msg_Header &);
```

Dado un *piggyback* del tipo `Piggyback_Msg_Header` y, según las reglas expuestas en § 3.6.3, este método se encarga de actualizar el *piggyback* del objeto correspondiente en el caso de que ya fuera conocido. De otra manera, si el *piggyback* no se conoce, se inserta.

En la tabla 6.2 se muestran los *piggyback* que se utiliza el sistema. La cantidad de espacio exacta que ocupa un conjunto de *piggybacks* está dada por la siguiente formula:

$$S = 4 + 64x_1 + 144x_2 + 96x_3$$

Donde x_1, x_2 y x_3 representan el espacio ocupado por cada tipo de *piggyback*: *eliminado*, *se busca* y *nueva referencia* que viajan en el conjunto, respectivamente. Hay, además, una carga extra de 4 bytes, que se debe a una variable de tipo `unsigned`, por medio de la cual se indica la cantidad de mensajes para un conjunto determinado.

```
size_t get_piggyback_chunk(const Site_Id & target_site_id,
                           void * out_buffer,
                           size_t out_buffer_size);
```

Mediante este método se obtienen un grupo de *piggybacks* hacia el sitio `target_site_id`. Los *piggybacks* son escritos en la región de memoria `out_buffer` de tamaño `out_buffer_size`. El método retorna el número de *piggybacks* empaquetados según los parámetros de configuración del localizador (ver § **??**) y el tamaño del buffer `out_buffer_size`.

```
void process_chunk(const Site_Id & src_site_id,
                   const void * in_buffer);
```

El método `process_chunk`, procesa un paquete de *piggybacks* almacenado en `in_buffer` generado por el método `get_piggyback_chunk` en el sitio `src_site_id`.

6.5.2. Búsqueda en caches

El conjunto de caches ofrece información acerca del paradero o estado de algún objeto. El siguiente algoritmo es el utilizado en cualquier situación en la que se requiera buscar información acerca de un objeto en los caches.

Algoritmo 6.5.1: BUSCARENCACHES(R_i)

entrada: R_i es una referencia

Busque R_i en el cache de eliminados
Si R_i fue encontrado
  Entonces Retornar (`OBJETO_ELIMINADO`)
Si no
  Llamemos R_n a la búsqueda de R_i en el cache de salida
  Si R_n es valida
    Entonces
      Si $R_n > R_i$
        Entonces
          $R_i \leftarrow R_n$
          Retornar (`REFERENCIA_RECIENTE`)
  Llamemos R_1 a la búsqueda de R_i en el cache de migración
  Llamemos R_2 a la búsqueda de R_i en el cache de nuevas referencias
  $R_3 \leftarrow$ MAX(R_1, R_2)
  Si $R_3 > R_i$
    Entonces Retornar (`REFERENCIA_RECIENTE`)
    Si no Retornar (`OBJETO_NO_ENCONTRADO`)

6.5.3. Procesamiento de la invocación

El procesamiento de una invocación está compuesto de cuatro etapas, dos ocurren en el cliente y las otras dos en el servidor. El cliente procesa el envío de la invocación y luego la recepción de la respuesta, mientras que en el servidor es de manera contraria. Primero ocurre la recepción de la invocación y luego el envío de la respuesta.

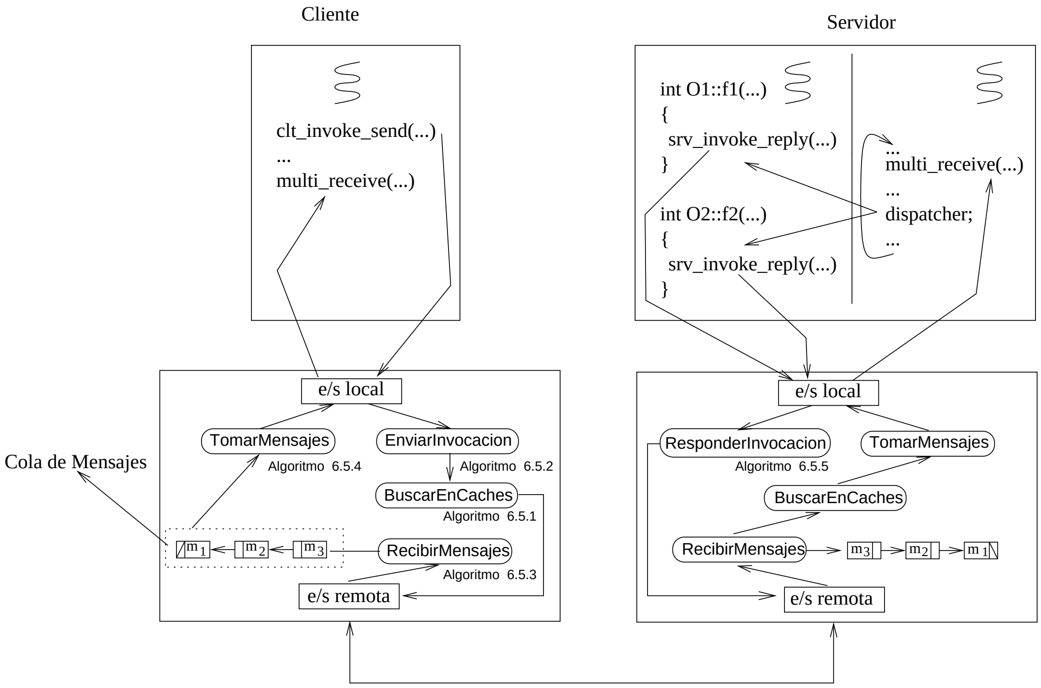

Figura 6.8: Flujo de los algoritmos de la invocación

En la figura 6.8 se aprecia donde tienen lugar los algoritmos descritos en esta sección. Al seguir la secuencia a través de las flechas se aprecia donde son ejecutados cada uno ellos.

En el envío de una invocación, un cliente del localizador utiliza la llamada `clt_invoke_send`. Luego, en el localizador origen, el servicio se ejecuta según el siguiente algoritmo:

Algoritmo 6.5.2: ENVIARINVOCACION(R_i, m_i)

---

entrada: R_i es la referencia del objeto a invocar

entrada: m_i es el mensaje de invocación

$R_n \leftarrow$ BUSCARENCACHES(R_i)
Si $R_n =$ `OBJETO_ELIMINADO`
  Entonces Retornar (`OBJETO_ELIMINADO`)

  De lo contrario si $R_n =$ `REFERENCIA_RECIENTE`
  Entonces $R_i \leftarrow R_n$
Añadir los *piggybacks* al mensaje de invocación m_i
Enviar m_i hacia R_i

Un mensaje de invocación o de respuesta, llegará por el modulo de recepción remota. Mediante algoritmo siguiente se lleva a cabo la recepción de mensajes:

Algoritmo 6.5.3: RECIBIRMENSAJES(O_i, m_r)

---

entrada: O_i es el objeto que ha de ser invocado

entrada: m_r es el mensaje recibido

Si O_i no se encuentra en la tabla propietaria
  Entonces $\begin{cases} R_n \leftarrow \text{BUSCARENCACHES}(O_i) \\ \text{Añadir los } \textit{piggybacks} \text{ al mensaje } m_r \\ \text{Enviar la respuesta } m_r \text{ hacia } R_n \end{cases}$

  Si no $\begin{cases} \text{Sea } e \text{ el estado del objeto } O_i \\ \textbf{Si } \text{e= } \texttt{RECIBIENDO} \\ \quad \textbf{Entonces } \text{Entregar el mensaje a } O_i \\ \\ \quad \textbf{Si no } \text{Encolar el mensaje } m_r \end{cases}$

En el procesamiento de la invocación y de la respuesta, un proceso que contiene objetos utiliza llamada `multi_receive`. Esta llamada indica al localizador la necesidad de recoger los mensajes existentes para cualquiera de los objetos residentes en el proceso. El algoritmo ejecutado en el localizador cuando se utiliza esta llamada es a grandes rasgos el siguiente:

Algoritmo 6.5.4: TOMARMENSAJES(m_r, p_r)

entrada: m_r es el mensaje a recibir: una invocación o una respuesta

entrada: p_r es el proceso que recibe

Si no existen mensajes por procesar
Entonces Bloquearse a la espera de invocaciones o respuestas

Si no $\begin{cases} \text{Sea } m_r \text{ un mensaje extraído de la cola de mensajes} \\ \text{Entregar al proceso } p_r \text{ el mensaje } m_r \end{cases}$

Finalmente, el proceso de invocación culmina con la llamada `srv_invoke_reply`. Desde el lado del servidor, esta llamada responde a una solicitud de invocación. El algoritmo a ejecutar es el siguiente:

Algoritmo 6.5.5: RESPONDERINVOCACIÓN(O, p_o, m_r)

entrada: p_o es la dirección del proceso que invocó

entrada: m_r es el mensaje de respuesta

entrada: O es el objeto invocado

Actualizar el cache de entrada con $< p_o, O >$
Añadir los *piggybacks* al mensaje m_r
Entregar la respuesta m_r a p_o

6.6. Migración

La migración de objetos consta de dos llamadas, una para desregistrar el objeto del sitio origen (`src_unreg_mig_obj`) y la otra para registrar el objeto en el sitio destino (`tgt_reg_mig_obj`).

Estas llamadas delegan al cliente del sistema la responsabilidad de sincronizar el orden entre el localizador o los procesos origen y destino.

Durante el proceso de migración pueden ocurrir varios escenarios:

1. Los procesos origen y destino de la migración utilizan las llamadas de la forma más directa y sin ningún mecanismo de sincronización. En el proceso origen se llama a `src_unreg_mig_obj` y en el proceso destino a `tgt_reg_mig_obj`.

   Las invocaciones serían redirigidas de acuerdo a la información que se halle en los caches. Existe la posibilidad de que una invocación entrante en el sitio origen de la migración sea respondida con un mensaje donde se indique la dirección más reciente.

   Por otro lado, podría suceder que la invocación fracase si el objeto no ha llegado aún al sitio destino. Si esto pasa, puede intentar reinvocarse pasado un período de tiempo escogido por el cliente. Este sería el mecanismo más simple a utilizar.

Si se requiere de mecanismos más elaborados, entonces se tienen las técnicas de *piggybacking*, prefetching con el cache de salida, o las difusiones.

2. Se pueden retener las invocaciones en el sitio origen de la migración y luego:

   - Se sincronizan el sitio origen y destino de la migración, de tal forma que se pueda conocer con certeza en el sitio origen, en qué momento el objeto ha llegado al sitio destino. Así, se puede retornar la nueva dirección del objeto en la respuesta de fracaso a la invocación.
   - Se reenvían las invocaciones hacia el sitio destino de la migración, pues éste es conocido por el sitio origen.

     Al realizar alguna de las operaciones descritas anteriormente, hay que tomar en cuenta el factor carga. Es decir, la migración toma algún tiempo, en el que es posible que una ráfaga de invocaciones llegue hacia el objeto migrante. Redireccionar el conjunto de invocaciones hacia el objeto migrante, puede traer como consecuencia sobrecargar al nuevo sitio destino, pues todos intentarían simultáneamente reinvocar hacia la nueva dirección.

3. Se puede eliminar el puerto del objeto. Esto haría fracasar la invocación con un mensaje de puerto inválido ú objeto inexistente. Esto deja como responsabilidad del cliente activar los métodos de búsqueda según su propio criterio.

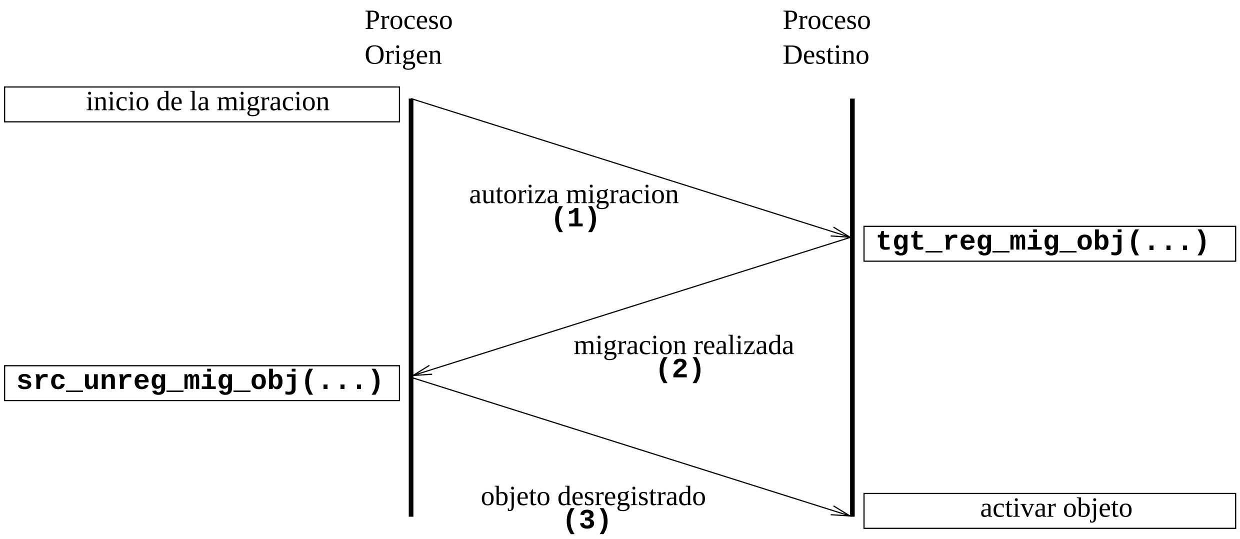

Figura 6.9: Sincronización de la migración por parte del cliente

En la figura 6.9 se muestra como ejemplo el escenario 2 en el que se desea sincronizar los sitios origen y destino de la migración. Una vez iniciada la migración, el proceso origen envía un mensaje al proceso destino donde autoriza la migración (1); así, el proceso destino podrá ejecutar el servicio `tgt_reg_mig_obj` de registro del objeto migrado. Luego, el proceso destino envía un mensaje al proceso origen indicando que la migración ha sido realizada (2). En este momento, el proceso origen ejecuta la llamada de desregistrado del objeto `src_unreg_mig_obj`. Finalmente, el proceso origen envía un mensaje donde indica que el objeto ha sido desregistrado exitosamente (3). Una vez recibido el mensaje, el proceso destino activa al objeto. Después de este último paso, el objeto está listo para recibir invocaciones y volver migrar.

6.6.1. Algoritmos de la migración

La llamada `tgt_reg_mig_obj` en el localizador destino sigue el algoritmo siguiente:

Algoritmo 6.6.1: REGITRAROBJMIGRADO(O_m, t)

entrada: O_m es el objeto que está migrando

entrada: t es la estampilla de tiempo lógico

Registrar el objeto O_m con la estampilla t
Actualizar en el cache de salida las referencias a O_m

La llamada `src_unreg_mig_obj` se corresponde con el siguiente algoritmo:

Algoritmo 6.6.2: DESREGISTRAOBJMIGRANTE(O_m, S_d, t)

entrada: O_m es el objeto que está migrando

entrada: S_d es el sitio destino de la migración

entrada: t es la estampilla de tiempo lógico actual

$t \leftarrow t + 1$
Insertar en el cache de migración la referencia $< O_m,\ S_d,\ t >$
Insertar en el cache de salida la referencia $< O_m,\ S_d,\ t >$
Fijar el estado del objeto O_m a `MIGRANDO`
Hacer prefetching con el cache de entrada según O_m

6.7. Difusión de mensajes

En nuestro prototipo, la difusión de mensajes se simula a través de un servidor centralizado de difusión. Este servidor ofrece llamadas de difusión confiable o no confiable a un conjunto de procesos que se han suscrito al servicio. Dicho conjunto forma un grupo. Cuando un miembro del grupo envía un mensaje de difusión, el servicio de difusión se encarga de propagarlos hacia cada uno de los miembros del grupo.

En la figura § 6.10 se aprecia la arquitectura general del servidor de difusiones. El módulo de entrada y salida es un `Remote_Multiserver_Point` (ver § 4.2.2.2). Desde allí se

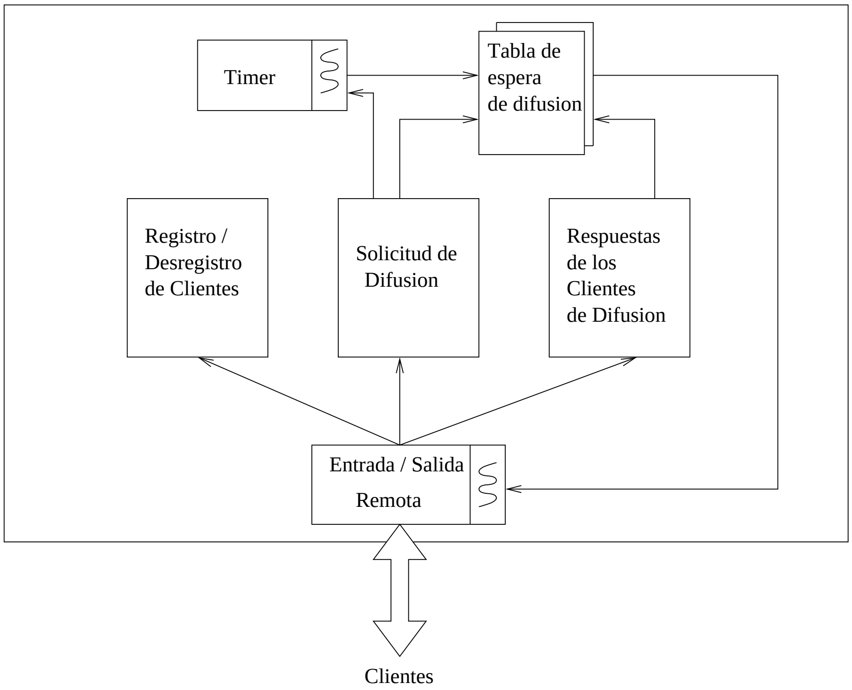

Figura 6.10: Arquitectura general del servidor de difusiones

despachan servicios de registro y desregistro de clientes de la difusión, solicitud de difusiones y respuestas a las difusiones. Todos estos servicios son manejados a través de la clase de entrada y salida remota. El servidor de difusión cuenta con un modulo planificador de eventos en el tiempo (*timer*), que gestiona los *timeouts*.

Para poder recibir un mensaje de difusión, es necesario suscribirse al servidor de difusión. Así, un cliente de este servicio podrá enviar y recibir mensajes.

Cuando el servidor de difusión recibe una solicitud, éste crea un registro en la tabla de espera de difusión. La solicitud especifica el número de clientes que se espera respondan al mensaje; también se planifica un tiempo de expiración por la espera de las respuestas.

Los mensajes de respuestas son recolectados por el servicio de difusión y, finalmente, al obtener todos los mensajes esperados o, al expirar el *timeout*, se responde al proceso que inició la solicitud.

La simulación de la difusión no está alejada del modelo real del servicio de difusión de mensajes.

6.7.1. Difusiones desde el localizador

Al iniciarse, cada localizador envía un mensaje de suscripción al servicio de difusiones. De esta manera, se pueden enviar y recibir difusiones de búsqueda desde cada localizador. El conjunto de localizadores forma entonces un grupo.

En la figura 6.11 se aprecia el mecanismo de difusión desde el localizador. Un cliente del localizador hace una llamada de difusión (1) para la búsqueda de objetos:

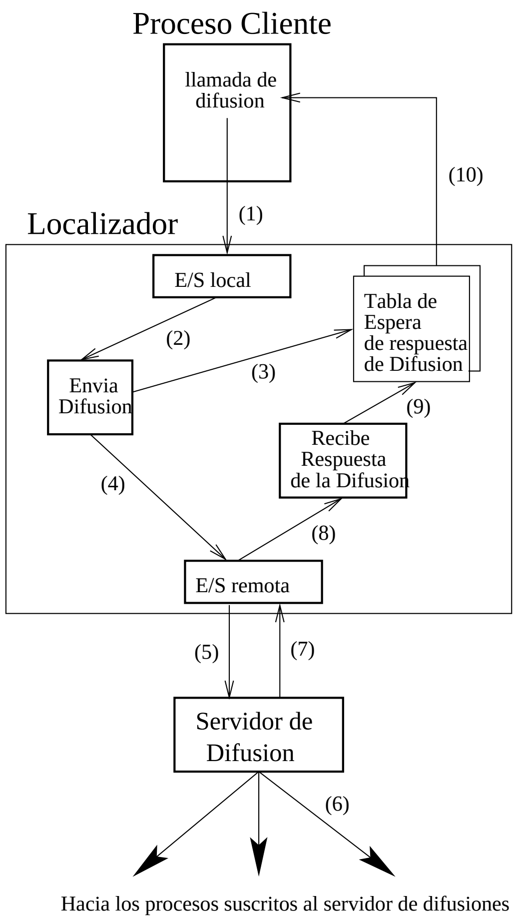

Figura 6.11: Mecanismo de la difusión desde el localizador

`strong_locate` o `weak_locate` (ver § 5.3.5). Una vez que el localizador recibe la llamada, el servicio correspondiente de difusión es ejecutado (2).

El gestor de la difusión dentro del localizador anota un registro con la dirección del cliente en la tabla de espera de respuestas de la difusión(3). Luego, desde el mismo servicio, es enviado un mensaje con la petición al servidor de difusiones (4,5).

El servidor de difusiones se encarga de entregar el mensaje a todos los miembros adscritos al servicio (6). Una vez que servidor recolecte las respuestas a la difusión, éstas son entregadas al localizador (7).

La entrega de las respuestas al localizador es procesada por un servicio especial para la recepción de respuestas (8). Una vez procesada la información obtenida del servicio de difusión, se busca en la tabla de espera la dirección del cliente que realizó la llamada (9). Luego, el localizador entrega al cliente (10) las respuestas de la difusión realizada.

6.7.2. Implementación del servicio real de difusión

El servicio de difusión explicado en la sección 6.7 es solo un prototipo para la difusión. Una discusión sobre el sistema real de difusión de mensajes se presenta en esta sección.

La difusión de mensajes debe hacerse a través de un protocolo de grupos. A este protocolo lo llamaremos Protocolo de Presentación, pues debe ser utilizado cuando un nuevo sitio se incorpora o se presenta al grupo.

El protocolo de presentación tiene como objetivo establecer normas para la comunicación en grupo, donde un conjunto de sitios se conocen unos a otros y se comunican eventualmente para cumplir un propósito común. En la definición de grupo que aquí se plantea, se establece una división por niveles o estratos. Con esta división es posible tener grupos de gran tamaño. Además, se establecerán normas para la división y jerarquización del grupo.

Un grupo está compuesto de varios subgrupos. La manera de comunicarse entre los subgrupos es mediante la elección de un representante de entre sus miembros. El representante servirá entonces para enlazar subgrupos.

A través del protocolo de presentación se puede conformar dinámicamente los grupos. En cualquier instante de tiempo un nuevo sitio podría incorporarse y mediante el protocolo de presentación se garantizaría la consistencia.

Dentro del protocolo se tienen 3 abstracciones: el sitio, el grupo y el representante de grupo. Un grupo esta compuesto de sitios y para establecer la comunicación entre subgrupos es necesario tener un representante.

El protocolo estructura a un grupo como se aprecia en la figura 6.12, donde además se tienen las siguientes derivaciones:

- Existen 3 subgrupos: G, G' y G".
- Todos los sitios de un mismo subgrupo están en un mismo nivel.
- Los subgrupos G y G' están en un mismo nivel.
- El subgrupo G" esta en un nivel superior a G y G'.
- G y G' hubieran sido independientes si G" no existiese.

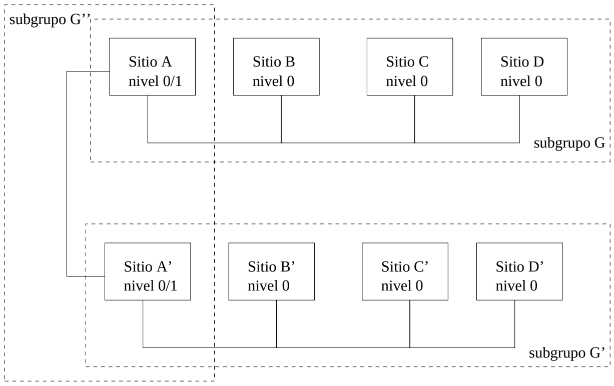

Figura 6.12: Abstracciones del protocolo de presentación

- El sitio A es el representante del grupo G y, el sitio A' es el del grupo G'.

Cada uno de los miembros del grupo posee una tabla de homólogos en la que se guarda una lista de los sitios que pertenecen a al mismo subgrupo.

Refiriéndonos a la figura 6.12, el Sitio B tendría una tabla donde se lista el Sitio A, C y D. Esta tabla de homólogos además tiene asociado un nivel, que para el caso del Sitio B es cero.

Los sitios A y A' pertenecen a dos subgrupos distintos y por lo tanto a niveles distintos. El Sitio A posee dos tablas de homólogos, una con nivel cero donde se encuentran los sitios B, C y D; y otra con nivel uno donde se encuentra el sitio A'. Lo mismo ocurre para el Sitio A'.

Dada la manera en que el protocolo organiza los sitios, se plantea una jerarquización. Esta jerarquización propone un crecimiento de abajo hacia arriba; es decir, siempre que un nuevo sitio entra en el sistema, se registra en un subgrupo con nivel 0; posteriormente, éste pudiera ocupar un nivel mayor si se convirtiera en representante de grupo.

Los eventos que suceden a grandes rasgos del arribo de un nuevo sitio a un subgrupo del sistema son:

1. Se envía un mensaje de difusión dentro de un subgrupo preguntando por la *tabla de homólogos de nivel 0*.

2. Se espera la respuesta de alguno de los sitios interrogados en el paso 1. Esta tabla contiene los sitios registrados en el nivel 0 donde ocurre la "presentación".

3. Al arribo de la tabla, se hace una difusión fuerte indicando la llegada del nuevo sitio. De esta forma ocurrirán las actualizaciones en las tablas de los sitios que conforman este subgrupo.

Fundamentalmente, en el protocolo, existen dos tipos de eventos. El primero es la presentación de un nuevo sitio al grupo. Como se aprecia en la figura 6.13, los sitios conectados entre sí son los de un mismo nivel. Al arribo de un nuevo sitio, éste solo se registra en el entorno del nivel cero.

El segundo es el envío de mensajes de tipo *broadcast*, que utilizará como soporte la estructuración propuesta. Los niveles $n >= 1$ sirven para agrupar sitios y para aliviar la carga del *broadcast*. Así, cuando un mensaje de difusión es enviado desde un nivel 0, éste atraviesa todos los niveles hasta llegar al más alto, donde en cada etapa se propaga a los niveles homólogos. Un mensaje que llega al nivel 1 será propagado al nivel 0; un mensaje que llega al nivel 2 sera propagado a los niveles 1 y 0 y así, sucesivamente.

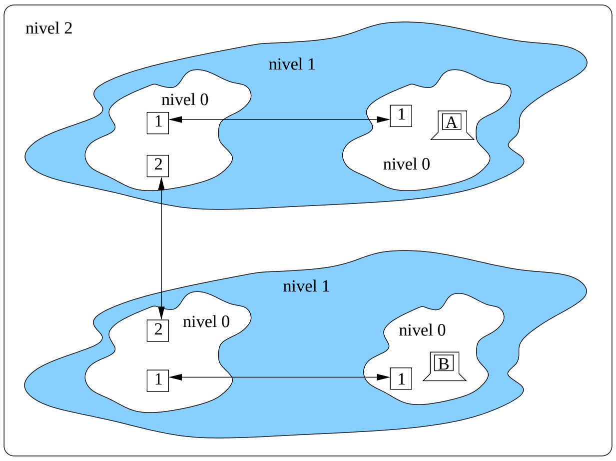

Figura 6.13: Estratificación por niveles

Observando nuevamente la figura 6.13, supongamos que desde el sitio A se envía un mensaje de difusión donde se busca un objeto que se encuentra en el sitio B. Primero, el mensaje se propaga entre todos los sitios locales a A; es decir, en los sitios con nivel 0. Puesto que en el entorno de A se ha definido un sitio con nivel 1, éste se comunicará con su homólogo. Al arribo del mensaje al sitio con nivel 1, éste es distribuido al nivel inferior (nivel 0). Luego, uno de ellos también es un representante de nivel 2 que, eventualmente, se comunicará con sus homólogos. En el ejemplo se observa que, cuando llegue el mensaje al sitio de nivel 2, comienza una propagación en orden inverso, desde el nivel 2 hacia el 1 y por último el 0, donde se encuentra el sitio B que contiene al objeto solicitado.

Capítulo 7

Desempeño

En este capítulo se presenta una breve evaluación experimental del sistema. Para ello se exponen dos tipos de medidas: el performance de la invocación y medidas para el desempeño de las técnicas de localización.

7.1. Desempeño de las invocaciones

Esta sección presenta una evaluación de rendimiento del mecanismo de invocación explicado en los capítulos 3 y 5.

Se comparó la diferencia de costo entre una invocación realizada a través del sistema IPC y una realizada con el servicio de localización. La idea es indagar el costo que impone el localizador a una invocación. Esto se averigua a través de la medición del tiempo de ida y vuelta de una invocación.

7.1.1. Tiempo de invocación en red de área local

Las medidas de esta subsección, fueron tomadas en la red *Ethernet* 100Mb/seg del CEMISID. La tabla 7.1 presenta las características de las máquinas usadas en el experimento.

En la figura 7.1 se observan invocaciones en el rango de 10 a 128 bytes. El tamaño de estos paquetes se considera pequeño. Se observa que los tiempos de ida y vuelta son prácticamente iguales. Al promediar el costo en este rango, tanto para el IPC como para el localizador, se tiene que el localizador añade un 51,6 % de costo al sistema IPC.

Características	*Máquina Cliente*	*Máquina Servidor*
Nombre y dominio	ritchie.cemisid.ing.ula.ve	dijkstra.cemisid.ing.ula.ve
IP	150.185.131.225	150.185.131.217
Arquitectura y Velocidad del Reloj	AMD Athlon 630 MHz	AMD-K7 600 MHz
Memoria RAM	128 MB	128 MB
Tarjeta de Red	100 Mb/seg	100 Mb/seg

Cuadro 7.1: Datos de las máquinas donde se realizaron las medidas locales

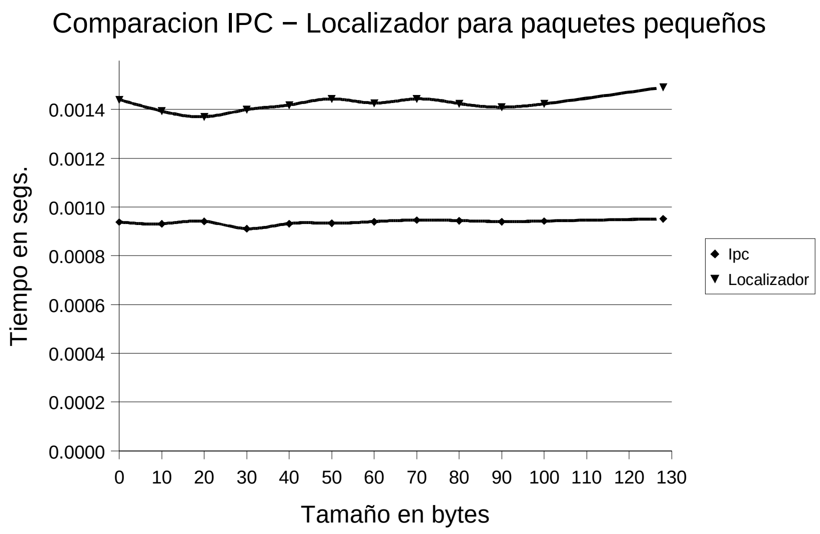

Figura 7.1: Comparación IPC - Localizador para paquetes pequeños en una red de área local

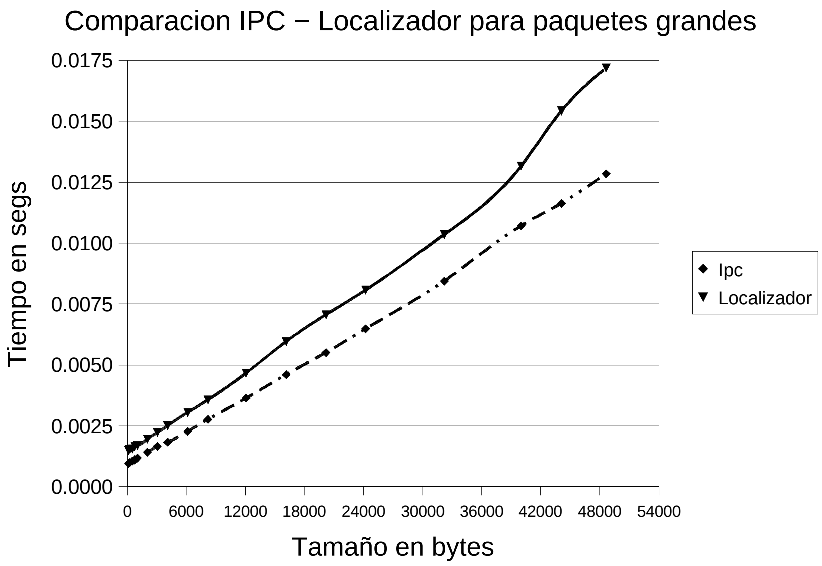

Figura 7.2: Comparación IPC - Localizador para paquetes grandes en una red de área local

En la figura 7.2 muestra el costo, tanto del IPC como del localizador, para tamaños de invocaciones entre 256 y 48656 bytes.

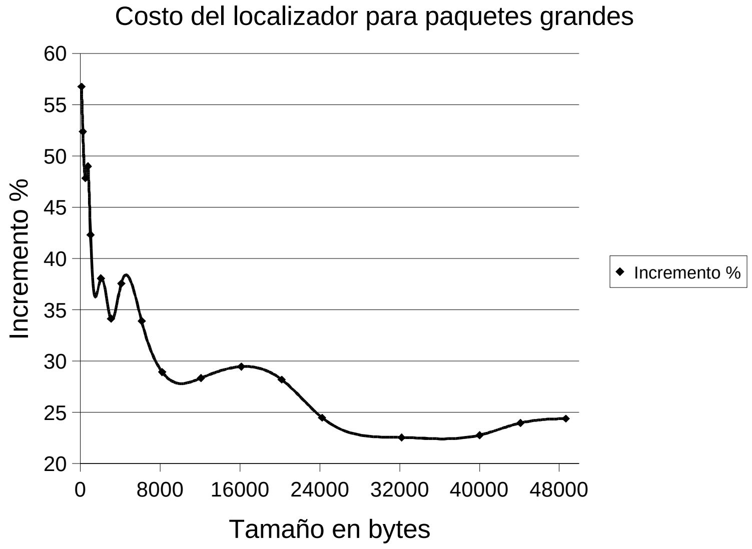

Figura 7.3: Incremento porcentual del localizador respecto al IPC para una red de área local

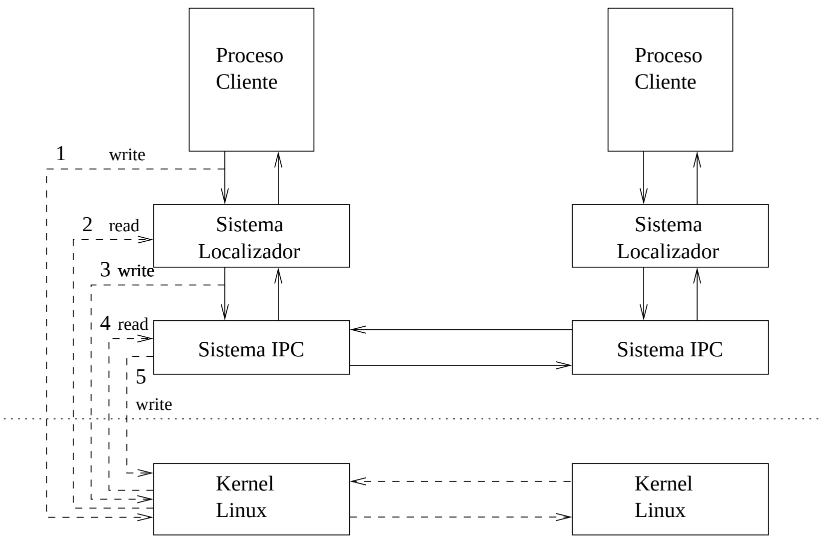

Figura 7.4: Flujo real de la comunicación entre procesos

El costo del localizador disminuye conforme aumenta el tamaño de la invocación. Para evidenciar esto, la figura 7.3 muestra el radio porcentual entre una invocación IPC y una invocación con el localizador. Se observa que a medida que el tamaño del paquete aumenta, el costo que añade el localizador se vuelve menos significativo, hasta llegar a

Características	*Máquina Cliente*	*Máquina Servidor*
Nombre y dominio	brooks.cemisid.ing.ula.ve	atenea.cecalc.ula.ve
IP	150.185.131.228	150.185.138.21
Arquitectura y Velocidad del Reloj	AMD 1.2 GHz	Pentium II 266 MHz
Memoria RAM	256 Mb	192 Mb
Tarjeta de Red	100 Mb/seg	100 Mb/seg

Cuadro 7.2: Datos de las máquinas donde se realizaron las medidas para redes distintas

oscilar entre un 20 % y 25 %.

La figura 7.4 ilustra el costo que tiene el flujo de mensajes entre procesos que utilizan la comunicación local, en este caso son: un proceso cliente, el sistema localizador y el sistema IPC. Todos utilizan *Unix Domain Sockets* del tipo *stream*. Las líneas continuas muestran el flujo normal de una invocación, mientras que las líneas punteadas el flujo real incluyendo al kernel.

Las líneas 1, 3 y 5 muestran el flujo real de comunicación cuando un cliente envía una invocación. El paso por el kernel se debe a la llamada sistema *write*. Por otro lado las lineas punteadas 2 y 4 muestran cuando cada proceso recibe mensajes a través de la llamada sistema *read*. Nótese que el flujo señalado solo describe el envío de un mensaje de invocación desde el proceso cliente hasta el IPC, que corresponde sólo a la mitad de la invocación del lado del cliente.

Dado que el mecanismo de comunicación local tiene la forma descrita, se puede decir que el costo añadido por el localizador a la invocación es aceptable para redes de área local.

La observación anterior sugiere emplear técnicas más eficientes para comunicación local; por ejemplo colas de mensajes [12]. Actualmente, este mecanismo impone límite de 4KB en el tamaño de los mensajes, que representa un orden de magnitud menos comparado con los 48KB que se alcanzaron en las pruebas.

7.1.2. Tiempo de invocación ínter-redes

Las medidas presentadas en esta subsección se refieren al tiempo de invocación utilizando dos redes distintas interconectadas, o lo que se conoce como una red de área metropolitana. En la figura 7.5 se aprecia el esquema de conexión entre el cliente y el servidor.

Como es de esperar, los tiempos de comunicación son mucho mayores a los obtenidos con la red de área local. En la tabla 7.2, se presentan las características de las máquinas utilizadas en el experimento.

Los tiempos de invocación para los dos sistemas de comunicación son graficados en las figuras 7.6 y 7.7. Cada punto en las gráficas, corresponde al mínimo de 5000 mediciones. Al observar las desviaciones estándar en cada punto, se aprecia claramente una variación considerable en la data. En el mejor de los casos, la diferencia entre el mínimo y el máximo de cada conjunto es de dos ordenes de magnitud. A través de la desviación estándar podemos confirmar que en este tipo de redes es normal tener variaciones extremas en los tiempos de ida y vuelta debido al jitter de red.

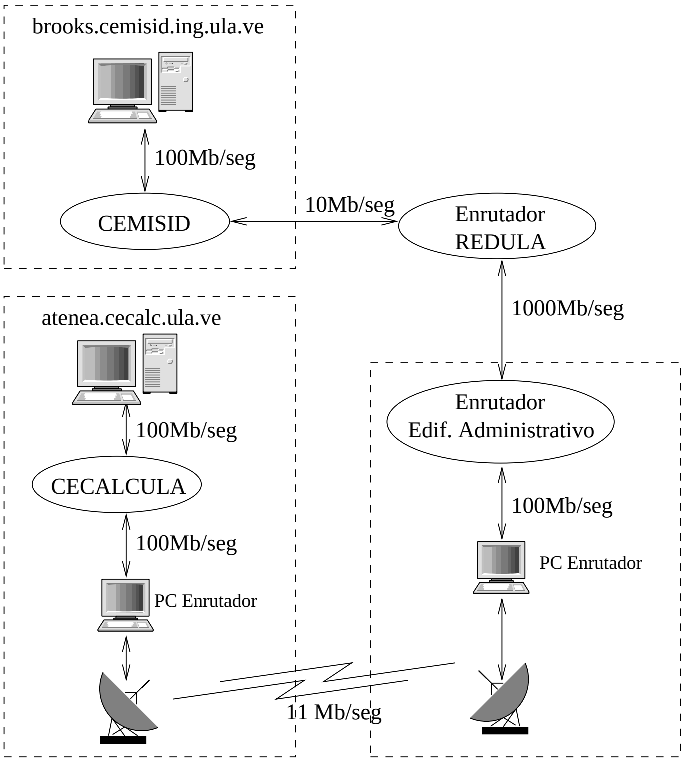

Figura 7.5: Mapa de conexión entre las máquinas cliente - servidor en redes distintas

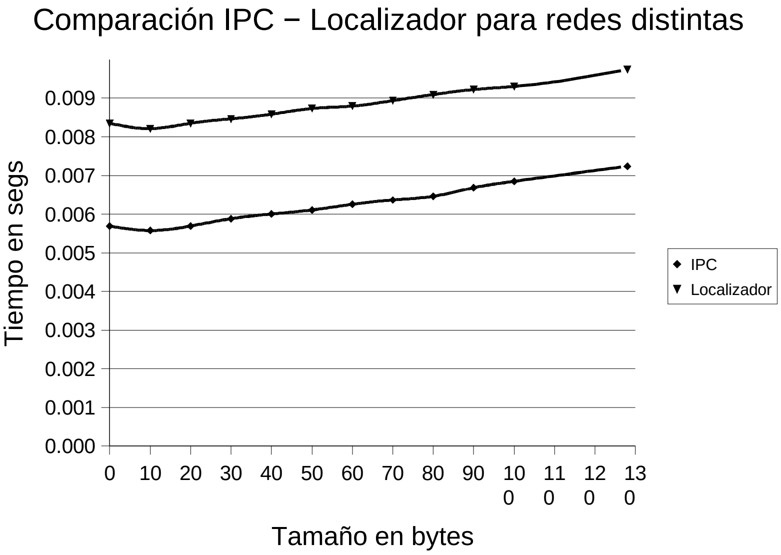

Figura 7.6: Comparación IPC - Localizador para paquetes pequeños

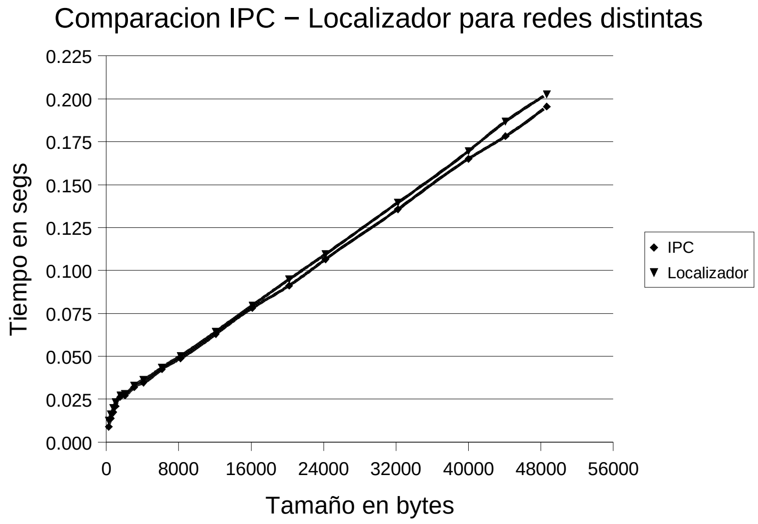

Figura 7.7: Comparación IPC - Localizador para paquetes grandes

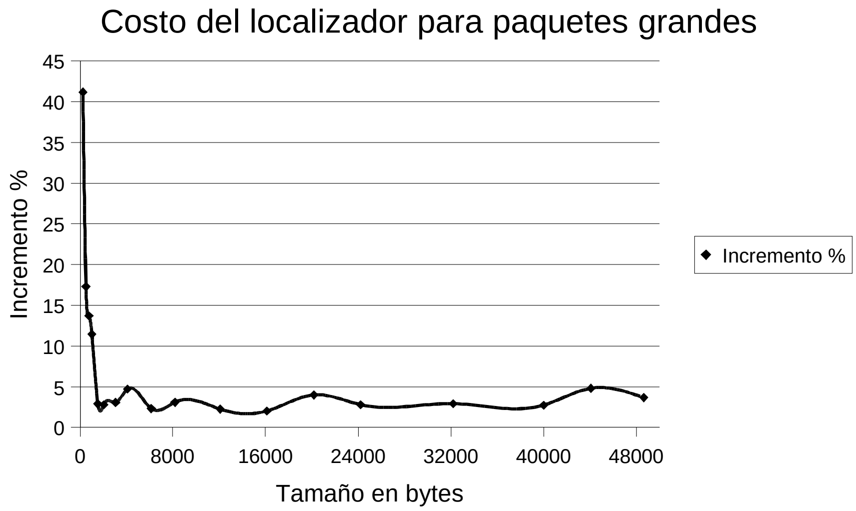

Figura 7.8: Incremento porcentual del localizador respecto al IPC para redes distintas

Al hacer una comparación entre el servicio IPC y el servicio de localización para redes distintas, nos damos cuenta que, según los tiempos mínimos, la invocación a través del servicio IPC es un poco más rápida que mediante el servicio de localización. El incremento porcentual es de 41,27 % para invocaciones pequeñas; este incremento es menor al 51,6 % promedio calculado en la red de aérea local, para los mismos valores. Para valores entre 256 y 48652 la diferencia llega a estar alrededor del 5 %, tal y como se muestra en la figura 7.8.

Es interesante observar el coeficiente de variación ($\sigma/\bar{x}*100\,\%$) para el conjunto de datos del IPC, cual es, en promedio de 785 %. En cambio, para una red de área local, el coeficiente de variación promedio es de 103 %.

Por otro lado, el coeficiente de variación promedio de las medidas reportadas para el localizador es de 671 %. Este mismo coeficiente tomado en una red de área local es de 53,82 %. Nuevamente, la diferencia entre ambos coeficientes de variación es grande (12 veces más grande para redes distintas); esto evidencia el alto jitter de red.

La gran diferencia entre los coeficientes de variación de una red de área metropolitana (MAN) y una red de área local, muestra que la latencia de red en una red MAN es significativa. Así, podemos intuir que para redes distintas el costo añadido por el localizador no es relevante.

Nuestro sistema se fue concebido desde el principio como un sistema para utilizarse a gran escala. En este sentido el sistema de localización exhibe una buena calidad de servicio.

7.2. Desempeño de las técnicas de localización

Para ilustrar la utilidad del caching y de nuestros métodos de actualización, se realizó un experimento con 5 sitios, 2 procesos por sitio, 5 objetos por proceso y 5 referencias por proceso; lo que totaliza 50 objetos y 50 referencias. Los objetos fueron programados para moverse por todo el espacio de sitios según una probabilidad. Las referencias fueron escogidas aleatoriamente y siempre hacia objetos remotos.

La invocación ocurría según una probabilidad p y la migración según una probabilidad q, tal que $p + q = 1$.

En un primer experimento, los localizadores solamente utilizaron las técnicas de caching y prefetching, sin utilizar *piggybacks*. En cada sitio, cada proceso tenía programado un total de 5000 intentos con probabilidad de invocación $p = 0{,}9$ y una probabilidad de migración $q = 0{,}1$. El objeto a migrar y el sitio destino de la migración eran escogidos de forma aleatoria.

En el primer experimento se obtuvo que, en promedio, sólo el 0,8 % de las invocaciones tuvieron que volverse a realizar; es decir, se obtuvo a través de una respuesta de fracaso una indicación de una referencia más nueva. Por otro lado, en el 99,2 % de las veces, las referencias siempre estaban actualizadas. Es importante acotar que nunca hubo fallas en la invocación; es decir, siempre había información en los caches acerca de la localidad del objeto. Nunca hubo necesidad de utilizar difusiones.

En un segundo experimento, se aumentó la probabilidad de migración a $q = 0{,}3$; lo que deja la probabilidad de invocación en $p = 0{,}7$. En esta configuración, hubo que reinvocar el 5,62 % de las veces. Nótese el aumento en el número de reinvocaciones producto del incremento en la velocidad de migración de los objetos.

El tercer experimento se realizó con los mismos parámetros del segundo, pero esta vez se utilizaron los *piggybacks*. Los resultados arrojaron una disminución de 49,8 % en el número de reinvocaciones; es decir, en promedio, en cada proceso se reinvocaba el 2,7 % de las veces. Este último experimento denota la incidencia que promete esta técnica en la localización de objetos móviles.

De igual forma, en los dos últimos experimentos, nunca hubo fallas en la invocación. Así pues, nunca tuvo que utilizarse el mecanismo de la difusión.

7.3. Consumo de memoria del localizador

La memoria ocupada por el demonio localizador en el momento del lanzamiento es de 4992 KB.

Luego de ejecutar un experimento tal como el que se describió en la sección 7.2, el promedio de carga de los localizadores era 8832 KB.

Mediante la siguiente formula, se puede conocer con precisión cuánta memoria se necesita para registrar procesos y objetos durante el tiempo que se ejecute localizador:

$$S = 84n_o + 80n_p$$

Donde, S es la cantidad total en bytes ocupada por el total de objetos n_o y el total de procesos n_p registrados en el localizador.

La formula se obtuvo al calcular la cantidad de espacio que ocupan un objeto y un proceso en las tablas de objetos y procesos respectivamente (§ 6.2.2).

Capítulo 8

Conclusión

La localización de objetos tal y como fue tratada en este trabajo, tiene una connotación completamente distribuida. Esto hace que buscar un objeto sea una tarea cooperativa, donde se puede prescindir de algunos de los localizadores sin que esto afecte la calidad del servicio gracias a la redundancia de información ofrecida por los caches.

El servicio de localización también muestra versatilidad, pues está ampliamente parametrizado. Cambiando unas pocas líneas de código, un localizador puede servir a clientes remotos, conformando así un esquema con un poco de centralización si así lo quisiera el cliente.

El sistema es escalable por varias razones. Primero, la distribución planteada en el sistema permite escalarlo, pues a mayor numero de sitios, mayor será la probabilidad de encontrar información redundante sobre el paradero de los objetos. Segundo, el sobrecoste del envío y recepción de invocaciones a través del localizador sobre redes distintas es despreciable.

El sistema garantiza la transparencia a través de una interfaz que ofrece la ilusión de centralización. El localizador es capaz de intervenir las invocaciones y redirigirlas, si en alguno de sus caches hay información más reciente del objeto invocado. Esto último hace al localizador transparente respecto al cliente.

8.1. Aportes del proyecto

Mediante una interfaz sencilla, es posible llevar a cabo tareas de localización, migración, e invocación de objetos. El sistema está escrito en C^{++} y el código es conforme a POSIX. Lo que hace que sea un sistema portátil entre diferentes versiones de UNIX.

Uno de los legados de este trabajo es el esquema para el desarrollo de servicios distribuidos explicado en el capítulo 4. Mediante un esquema orientado a objetos, se pueden construir procesos que administran servicios de forma distribuida. El esquema propuesto es sencillo, funcional y fácilmente extensible. Muestra de ello se puede apreciar con el desarrollo del servicio de localización y el servicio de difusión en cascada.

El servicio de localización muestra como a través de los servicios distribuidos se puede lograr un esquema cooperativo entre procesos que se encuentren esparcidos en distintas máquinas. Por otro lado, el servicio de difusión muestra que puede implantarse un servicio centralizado para propósitos de distribución utilizando las mismas herramientas que para los servicios distribuidos.

El servicio de localización de objetos sirve como base para el desarrollo de aplicaciones distribuidas orientadas a objetos. Además, los objetos que intervienen en el sistema pueden moverse a voluntad, pues se disponen de mecanismos para la actualización de sus referencias.

Una de las principales premisas en el diseño de nuestro sistema fue la escalabilidad. Las técnicas de caching, prefetching y piggybacks aunadas a la arquitectura distribuida del sistema, hacen posible que el sistema sea escalable. Pues, a medida que aumenta el número de sitios, mayor será la probabilidad de encontrar información redundante acerca del paradero de los objetos.

Finalmente, la arquitectura del sistema es sencilla, el sistema es bastante cohesionado y de mínimo acoplamiento, todo esto se traduce en que en un futuro se pueda mantener y ampliar con más facilidad.

8.2. Lecciones aprendidas

El caching es una técnica efectiva de localización. Permite fijar un límite en el consumo de recursos y recuperar información rápidamente. En este sentido se evidencia que la política de reemplazo LRU es la mejor.

Este trabajo demuestra que se puede prescindir de la técnica de *forwarding* para hacer localización distribuida.

Aunque el localizador realiza transparentemente localizaciones efectivas, delega lo posible al cliente, quien podrá entonar el sistema según el perfil de sus aplicaciones.

8.3. Perspectivas

8.3.1. Mejoras al localizador

Una primera mejora al localizador puede ocurrir en el sistema de servicios distribuidos (§ 4). Más robustez pudiera ser añadida tratando el caso de desconexión abrupta de un cliente. Se necesitaría una especie de recolector de basura que libere los recursos ocupados por un cliente desconectado abruptamente.

También se puede aliviar el costo de manejo de memoria, si se usan arenas de memoria. Así los servicios despachados asíncronamente no requieren de la operación de solicitud de memoria *malloc*.

La comunicación remota para el servicio IPC fue implantada utilizando UDP. Una mejora sustancial podría hacerse si se utilizan *Raw Sockets*.

Utilizando *Raw Sockets* se pudiera descargar al kernel de trabajo. Para el envío de mensajes, se omite por completo la capa UDP, interactuando directamente con la capa IP a través de paquetes propios de la aplicación (sin cabecera UDP). La recepción de paquetes es mejor, pues una vez que un paquete ha llegado a la capa de enlace de datos, éste es pasado a un servicio propio de captura de paquetes, quien los pasa directamente a la aplicación.

El servicio de difusión puede mejorarse significativamente desarrollando el esquema explicado en § 6.7.2. Para implantar la difusión no confiable se puede utilizar el sistema *multicasting* de IP, con el que se envían mensajes no confiables de manera sencilla a un conjunto de máquinas suscritas a una determinada dirección IP. La ventaja del *multicasting* IP es que se pueden enviar mensajes entre redes de área local y entre redes distintas.

8.3.2. Modelos analíticos y de simulación

Modelos analíticos y de simulación del localizador ayudarían notablemente a mejorar la evaluación de nuestras técnicas. Se ahorraría tiempo en las pruebas a gran escala.

Los experimentos realizados no representaron aplicaciones reales y son de baja escala. Es necesario realizar estudios completos para la entonación del sistema.

Para incrementar la escala deben utilizarse, en una primera etapa, un mayor número de máquinas, procesos, objetos y referencias para las pruebas en la redes de área local. Si se quiere representar aplicaciones reales, deben buscarse modelos estadísticos con los que se pueda modelar la frecuencia de invocación y migración en las aplicaciones. Esto permitiría simular y comprender una aplicación distribuida real. En una segunda etapa, se puede agrandar la escala si se incluyen máquinas de otras redes.

8.3.3. *Forward Addresses* en capa de transporte

Una mejora sustancial puede hacerse a nivel de transporte de mensajes. Tal y como esta implantado actualmente, el sistema de invocación debe esperar llevar todo el mensaje de un sitio a otro para percatarse que el objeto servidor no está disponible.

A nivel de transporte, un mensaje es dividido en paquetes de red. Pudiera implantarse un protocolo donde al observarse el arribo del primer paquete, se reenvíe al nuevo sitio destino sin tener que esperar por el resto del mensaje.

También es importante que se pueda responder al cliente desde el sitio final de la migración e, implantar la compresión de la cadena de lazos.

8.3.4. Captura de objetos rápidos

Una de las suposiciones sobre la migración es que los objetos tienen una velocidad de migración menor que la velocidad de invocación. Bajo esta suposición, siempre se puede encontrar un objeto, aun en el peor de los casos (cuando se recurre a la difusión).

Sin embargo, si la migración de un objeto se realiza con más frecuencia que la invocación, no sería factible alcanzar al objeto requerido. Este problema es denominado el de "*captura de objetos rápidos*".

En un primer acercamiento a la solución de este problema, un grupo de localizadores pueden ser instruidos acerca de la existencia de un objeto rápido. Estos podrían, entonces, actuar como "*alcabalas*" de control, que se encargarían de detener el objeto hasta que pueda ser alcanzado.

Bibliografía

[1] Y. Artsy and R. Finkel. Designing a process migration facility - the charlotte experience. *IEEE Computer*, 22(9):47–56, 1989.

[2] R. T. Braden. T/tcp - tcp extentions for transactions, functional specification. *RFC*, (1644):38, jul 1994.

[3] Jeffrey Chasse, G. Amador Franz, Edward Lazowska, Henry Levy, and Richard Litlefield. The amber system: parallel programming on a network of multiprocessors. In *12th ACM Symposium on Operating Systems Principles*, pages 147–158, dec 1989.

[4] Mossière Jacques Chevalier Pierre-Yves, Hagimot Daniel and Xavier Rousset de Pina. Le système réparti à objets Guide. Technical report, 1995.

[5] Frederik Douglis and Jhon K. Ousterhout. Process migration in the sprite operating system. In *7th International Conference on Distributed Computer Systems*, sep 1987.

[6] Paulo Ferreira. *Larchant: ramasse-miettes dans une mémoire partagé répartie avec persistance par atteignabilité.* PhD thesis, Université Paris VI, may 1996.

[7] Robert Fowler. The complexity of using forwarding addresses for decentralized object finding. In *5th ACM SIGOPS Conference on Principles of Distributed Computing*, pages 108–120, aug 1986.

[8] A. Goscinski. *Distributed Operating Systems. The Logical Design.* Addison-Wesley, 1991. ISBN 0-201-41704-9.

[9] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17:64–76, jan 1991.

[10] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transaction on Computer Systems*, 6(1):109–113, feb 1988.

[11] Leandro León. *Une architecture pour les systèmes répartis à objets mobiles.* PhD thesis, Université Paris VI, feb 1998.

[12] Carlos Nava and León Leandro. Un sistema IPC para eldesarrollo de aplicaciones cliente-servidor en Internet. In *XXVI Conferencia Latinoamericana de Informática*, sept 2000.

[13] David Plainfossé. *Distributed Garbage Collection and Referencing Managment in the Soul System*. PhD thesis, Université Paris VI, jun 1994.

[14] M. Powel and B. Miller. Process migration in DEMOS/MP. In *9th Symposium on Operating Systems Principles*, pages 110–119, oct 1983.

[15] Mark Rozier, Vadim Abrossimov, François Armoand, Ivan Boule, Michel Gien, Frederic Herrmann, C. Kaiser, P. Leonard, S. Langlois, and W.Ñeuhauser. The chorus distributed operating system. *Computing Systems*, 1:305–379, oct 1988.

[16] Alexander Schill and Markus Mock. Dc++: Distributed object-oriented system support on top of OSF DCE. *Distributed System Engineering*, 1(2):112–125, dec 1993.

[17] Marc Shapiro, Ivon Gourhent, Sabine Habert, Laurence Mosseri, Michael Ruffin, and Céline Valot. Sos: An object-oriented operating system - assesment and perspectives. *Computing Systems*, 2(4):287–337, 1989.

[18] Mark Shapiro, Peter Dickman, and David Plainfossé. Ssp: Chain robust, distruted references supporting acyclic garbage collection. Technical Report 1997, INRIA, nov 1992.

[19] Mukesh Singhal and Niranjan Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, Inc., 1994. ISBN 0-07-057572-X.

[20] P. Sinha, K. Park, X. Jia, K. Shimuze, and M. Maekawa. Process migration in the galaxy distributed system. In *5th International Parallel Processing Symposium*, pages 611–618, 1991.

[21] Pradeep Sinha. *Distributed Operating Systems*. IEEE Press, 1996. ISBN 0-7803-1119-1.

[22] C. Steketee, W. Zhu, and P. Moseley. Implementation of process migration in amoeba. In *14th International Conference on Distributed Systems*, pages 194–203, 1994.

[23] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemtable remote execution facilities for the v-system. In *10th ACM symposium on Operating Systems Principles*, pages 2–14, dec 1985.