5

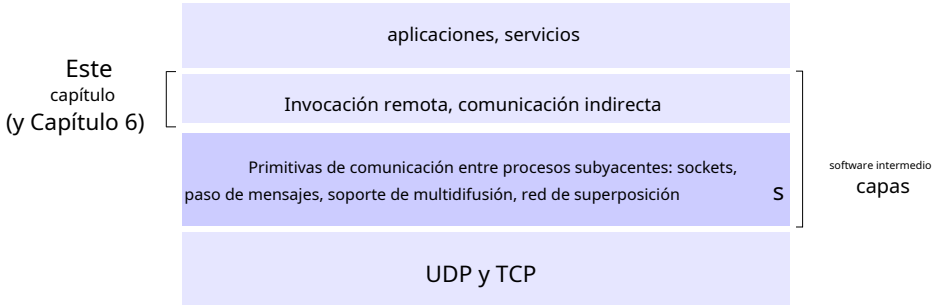
INVOCACIÓN A DISTANCIA

Este capítulo avanza a través de los paradigmas de invocación remota presentados en el Capítulo 2 (las técnicas de comunicación indirecta se abordan en el Capítulo 6). El capítulo comienza examinando el servicio más primitivo, la comunicación de solicitud-respuesta, que representa mejoras relativamente menores a las primitivas de comunicación entre procesos subyacentes discutidas en el Capítulo 4. Luego, el capítulo continúa examinando las dos técnicas de invocación remota más destacadas para la comunicación en sistemas distribuidos:

- El enfoque de llamada a procedimiento remoto (RPC) extiende la abstracción de programación común de la llamada a procedimiento a entornos distribuidos, lo que permite que un proceso de llamada llame a un procedimiento en un nodo remoto como si fuera local.
- La invocación de método remoto (RMI) es similar a RPC pero para objetos distribuidos, con beneficios adicionales en términos de usar conceptos de programación orientada a objetos en sistemas distribuidos y también extender el concepto de una referencia de objeto a los entornos distribuidos globales y permitir el uso de referencias a objetos como parámetros en invocaciones remotas.

El capítulo también presenta Java RMI como un caso de estudio del enfoque de invocación de método remoto (también se puede obtener más información en el Capítulo 8, donde analizamos CORBA).

Figura 5.1 Capas de software intermedio



5.1 Introducción

Este capítulo se ocupa de cómo los procesos (o entidades en un nivel superior de abstracción, como objetos o servicios) se comunican en un sistema distribuido, examinando, en particular, los paradigmas de invocación remota definidos en el Capítulo 2:

- *Protocolos de solicitud-respuesta*representan un patrón sobre el paso de mensajes y admiten el intercambio bidireccional de mensajes como se encuentra en la informática cliente-servidor. En particular, dichos protocolos brindan soporte de nivel relativamente bajo para solicitar la ejecución de una operación remota y también brindan soporte directo para RPC y RMI, que se analizan a continuación.

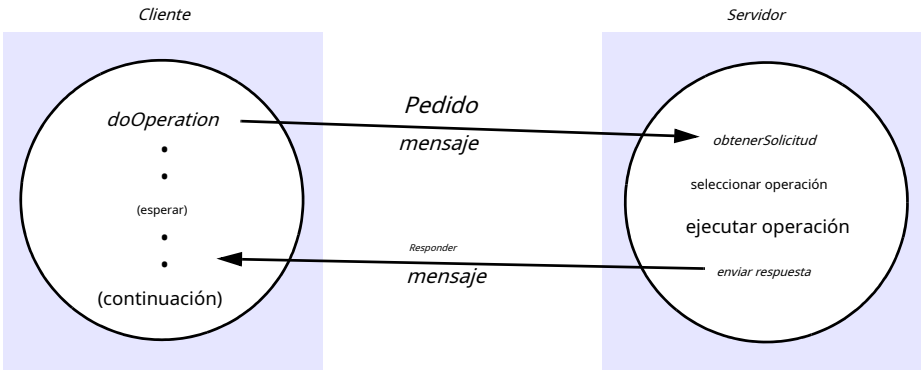
- El ejemplo más antiguo y quizás el más conocido de un modelo más amigable para el programador fue la extensión del modelo de llamada a procedimiento convencional a los sistemas distribuidos (el*llamada a procedimiento remoto*, o RPC, modelo), que permite que los programas cliente llamen a los procedimientos de forma transparente en los programas del servidor que se ejecutan en procesos separados y, en general, en computadoras diferentes a las del cliente.

- En la década de 1990, el modelo de programación basado en objetos se amplió para permitir que los objetos en diferentes procesos se comunicaran entre sí por medio de *invocación de método remoto*(RMI). RMI es una extensión de la invocación de métodos locales que permite que un objeto que vive en un proceso invoque los métodos de un objeto que vive en otro proceso.

Tenga en cuenta que usamos el término 'RMI' para referirnos a la invocación de métodos remotos de manera genérica – esto no debe confundirse con ejemplos particulares de invocación de métodos remotos como Java RMI.

Volviendo al diagrama presentado por primera vez en el Capítulo 4 (y reproducido en la Figura 5.1), este capítulo, junto con el Capítulo 6, continúa nuestro estudio de los conceptos de middleware centrándose en la capa por encima de la comunicación entre procesos. En particular, las Secciones 5.2 a 5.4 se enfocan en los estilos de comunicación enumerados anteriormente, y la Sección 5.5 proporciona un estudio de caso más complejo, Java RMI.

Figura 5.2 Comunicación de solicitud-respuesta



5.2 Protocolos de solicitud-respuesta

Esta forma de comunicación está diseñada para respaldar los roles y los intercambios de mensajes en las interacciones típicas entre cliente y servidor. En el caso normal, la comunicación de solicitud-respuesta es síncrona porque el proceso del cliente se bloquea hasta que llega la respuesta del servidor. También puede ser confiable porque la respuesta del servidor es efectivamente un reconocimiento para el cliente. La comunicación asincrónica de solicitud-respuesta es una alternativa que puede ser útil en situaciones en las que los clientes pueden recuperar las respuestas más tarde; consulte la Sección 7.5.2.

Los intercambios cliente-servidor se describen en los párrafos siguientes en términos de *enviar* y *recibir* operaciones en la API de Java para datagramas UDP, aunque muchas implementaciones actuales usan flujos TCP. Un protocolo creado sobre datagramas evita los gastos generales innecesarios asociados con el protocolo de flujo TCP. En particular:

- Los acuses de recibo son redundantes, ya que las solicitudes van seguidas de respuestas.
- Establecer una conexión implica dos pares de mensajes adicionales además del par requerido para una solicitud y una respuesta.
- El control de flujo es redundante para la mayoría de las invocaciones, que pasan solo pequeños argumentos y resultados.

El protocolo de solicitud-respuesta •El protocolo que describimos aquí se basa en un trío de primitivas de comunicación, *doOperation*, *obtenerSolicitud* y *enviar respuesta*, como se muestra en la Figura 5.2. Este protocolo de solicitud-respuesta hace coincidir las solicitudes con las respuestas. Puede estar diseñado para proporcionar ciertas garantías de entrega. Si se utilizan datagramas UDP, las garantías de entrega deben ser proporcionadas por el protocolo de solicitud-respuesta, que puede utilizar el mensaje de respuesta del servidor como reconocimiento del mensaje de solicitud del cliente. La figura 5.3 describe las tres primitivas de comunicación.

El *doOperation* Los clientes utilizan el método para invocar operaciones remotas. Sus argumentos especifican el servidor remoto y qué operación invocar, junto con información adicional (argumentos) requerida por la operación. Su resultado es una matriz de bytes que contiene la respuesta. Se supone que el cliente que llama *doOperation* ordena el

Figura 5.3 Operaciones del protocolo de solicitud-respuesta

byte público[] doOperation (RemoteRef s, int operationId, byte[] argumentos)
Envía un mensaje de solicitud al servidor remoto y devuelve la respuesta. Los argumentos especifican el servidor remoto, la operación a invocar y los argumentos de esa operación.

byte público[] getRequest ();
Adquiere una solicitud de cliente a través del puerto del servidor.

public void sendReply (byte[] respuesta, InetAddress clientHost, int clientPort);
Envía el mensaje de respuesta *responder* al cliente en su dirección y puerto de Internet.

argumentos en una matriz de bytes y ordena los resultados de la matriz de bytes que se devuelve. El primer argumento de *doOperation* es una instancia de la clase *RemoteRef*, que representa referencias para servidores remotos. Esta clase proporciona métodos para obtener la dirección de Internet y el puerto del servidor asociado. El *doOperation* El método envía un mensaje de solicitud al servidor cuya dirección de Internet y puerto se especifican en la referencia remota proporcionada como argumento. Después de enviar el mensaje de solicitud, *doOperation* invoca *recibir* para obtener un mensaje de respuesta, del cual extrae el resultado y lo devuelve a la persona que llama. la persona que llama *doOperation* se bloquea hasta que el servidor realiza la operación solicitada y transmite un mensaje de respuesta al proceso del cliente.

obtenerSolicitud es utilizado por un proceso de servidor para adquirir solicitudes de servicio, como se muestra en la Figura 5.3. Cuando el servidor ha invocado la operación especificada, utiliza *enviar respuesta* para enviar el mensaje de respuesta al cliente. Cuando el cliente recibe el mensaje de respuesta, el original *doOperation* se desbloquea y continúa la ejecución del programa cliente.

La información a transmitir en un mensaje de solicitud o en un mensaje de respuesta se muestra en la Figura 5.4. El primer campo indica si el mensaje es un *Pedido* o un *Responder* mensaje. El segundo campo, *ID de solicitud,* contiene un identificador de mensaje. A *doOperation* en el cliente genera un *ID de solicitud* para cada mensaje de solicitud, y el servidor copia estos ID en los mensajes de respuesta correspondientes. Esto permite *doOperation* para verificar que un mensaje de respuesta sea el resultado de la solicitud actual, no una llamada anterior retrasada. El tercer campo es una referencia remota. El cuarto campo es un identificador de la operación a invocar. Por ejemplo, las operaciones en una interfaz se pueden numerar 1, 2, 3, ..., si el cliente y el servidor usan un lenguaje común que admita la reflexión, se puede colocar una representación de la operación en este campo.

Figura 5.4 Estructura del mensaje de solicitud-respuesta

Tipo de mensaje	*En t (0=Solicitud, 1=Responder)*
ID de solicitud	*En t*
referencia remota	*RemoteRef*
operaciónId	*int u operación*
argumentos	*// matriz de bytes*

Identificadores de mensajes •Cualquier esquema que involucre la gestión de mensajes para proporcionar propiedades adicionales, como la entrega confiable de mensajes o la comunicación de solicitud-respuesta, requiere que cada mensaje tenga un identificador de mensaje único mediante el cual se pueda hacer referencia a él. Un identificador de mensaje consta de dos partes:

1. un*ID de solicitud*, que el proceso de envío toma de una secuencia creciente de enteros;

2. un identificador para el proceso emisor, por ejemplo, su puerto y dirección de Internet.

La primera parte hace que el identificador sea único para el remitente y la segunda parte lo hace único en el sistema distribuido. (La segunda parte se puede obtener de forma independiente; por ejemplo, si se utiliza UDP, a partir del mensaje recibido).

Cuando el valor de la*ID de solicitud*alcanza el valor máximo para un entero sin signo (por ejemplo, $2^{32} - 1$) se pone a cero. La única restricción aquí es que la vida útil de un identificador de mensaje debe ser mucho menor que el tiempo necesario para agotar los valores en la secuencia de enteros.

Modelo de falla del protocolo de solicitud-respuesta •Si las tres primitivas*doOperation*, *obtenerSolicitud*y*enviar respuesta*se implementan sobre datagramas UDP, luego sufren las mismas fallas de comunicación. Eso es:

- Sufren fallas por omisión.

- No se garantiza que los mensajes se entreguen en el orden del remitente.

Además, el protocolo puede sufrir fallas en los procesos (consulte la Sección 2.4.2). Suponemos que los procesos tienen fallas de bloqueo. Es decir, cuando se detienen, permanecen detenidos, no producen un comportamiento bizantino.

Para tener en cuenta las ocasiones en que un servidor ha fallado o se cae un mensaje de solicitud o respuesta,*doOperation*usa un tiempo de espera cuando está esperando recibir el mensaje de respuesta del servidor. La acción tomada cuando se produce un tiempo de espera depende de las garantías de entrega que se ofrecen.

Tiempos de espera •Hay varias opciones en cuanto a lo que*doOperation*puede hacer después de un tiempo de espera. La opción más sencilla es regresar inmediatamente de*doOperation*con una indicación al cliente de que el*doOperation*ha fallado. Este no es el enfoque habitual: el tiempo de espera puede deberse a la pérdida del mensaje de solicitud o respuesta y, en este último caso, la operación se habrá realizado. Para compensar la posibilidad de mensajes perdidos, *doOperation*envía el mensaje de solicitud repetidamente hasta que obtiene una respuesta o está razonablemente seguro de que la demora se debe a la falta de respuesta del servidor y no a la pérdida de mensajes. Eventualmente, cuando*doOperation* devoluciones, le indicará al cliente mediante una excepción que no se recibió ningún resultado.

Descartar mensajes de solicitud duplicados •En los casos en que el mensaje de solicitud es retransmitido, el servidor puede recibirlo más de una vez. Por ejemplo, el servidor puede recibir el primer mensaje de solicitud pero tardar más que el tiempo de espera del cliente en ejecutar el comando y devolver la respuesta. Esto puede hacer que el servidor ejecute una operación más de una vez para la misma solicitud. Para evitar esto, el protocolo está diseñado para reconocer mensajes sucesivos (del mismo cliente) con el mismo identificador de solicitud y filtrar duplicados. Si el servidor aún no ha enviado la respuesta, no necesita realizar ninguna acción especial: transmitirá la respuesta cuando haya terminado de ejecutar la operación.

Mensajes de respuesta perdidos •Si el servidor ya envió la respuesta cuando recibe una solicitud duplicada, deberá ejecutar la operación nuevamente para obtener el resultado, a menos que haya almacenado el resultado de la ejecución original. Algunos servidores pueden ejecutar sus operaciones más de una vez y obtener los mismos resultados cada vez. Un*operación idempotente* Es una operación que se puede realizar repetidamente con el mismo efecto que si se hubiera realizado exactamente una vez. Por ejemplo, una operación para agregar un elemento a un conjunto es una operación idempotente porque siempre tendrá el mismo efecto en el conjunto cada vez que se realiza, mientras que una operación para agregar un elemento a una secuencia no es una operación idempotente porque extiende la secuencia cada vez que se realiza. Un servidor cuyas operaciones son todas idempotentes no necesita tomar medidas especiales para evitar ejecutar sus operaciones más de una vez.

Historia •Para servidores que requieren retransmisión de respuestas sin reejecución de operaciones, se puede utilizar un historial. El término 'historial' se utiliza para referirse a una estructura que contiene un registro de los mensajes (de respuesta) que se han transmitido. Una entrada en un historial contiene un identificador de solicitud, un mensaje y un identificador del cliente al que se envió. Su propósito es permitir que el servidor retransmita mensajes de respuesta cuando los procesos del cliente los soliciten. Un problema asociado con el uso de un historial es su costo de memoria. Un historial se volverá muy grande a menos que el servidor pueda decir cuándo los mensajes ya no serán necesarios para la retransmisión.

Como los clientes solo pueden realizar una solicitud a la vez, el servidor puede interpretar cada solicitud como un reconocimiento de su respuesta anterior. Por lo tanto, el historial debe contener solo el último mensaje de respuesta enviado a cada cliente. Sin embargo, el volumen de mensajes de respuesta en el historial de un servidor puede seguir siendo un problema cuando tiene una gran cantidad de clientes. Esto se ve agravado por el hecho de que, cuando finaliza un proceso de cliente, no reconoce la última respuesta que recibió; por lo tanto, los mensajes en el historial normalmente se descartan después de un período de tiempo limitado.

Estilos de protocolos de intercambio •Se utilizan tres protocolos, que producen comportamientos diferentes en presencia de fallas de comunicación, para implementar varios tipos de comportamiento de solicitud. Originalmente fueron identificados por Spector [1982]:

- el*solicitud (R)*protocolo;
- el*solicitud-respuesta (RR)*protocolo;
- el*solicitud-respuesta-reconocimiento de respuesta (RRA)*protocolo.

Los mensajes pasados en estos protocolos se resumen en la Figura 5.5. En el protocolo R, un solo *Pedido*El mensaje es enviado por el cliente al servidor. El protocolo R se puede utilizar cuando no hay ningún valor que devolver desde la operación remota y el cliente no requiere confirmación de que la operación se ha ejecutado. El cliente puede proceder inmediatamente después de enviar el mensaje de solicitud, ya que no es necesario esperar un mensaje de respuesta. Este protocolo se implementa sobre datagramas UDP y, por lo tanto, sufre las mismas fallas de comunicación.

El protocolo RR es útil para la mayoría de los intercambios cliente-servidor porque se basa en el protocolo de solicitud-respuesta. No se requieren mensajes de reconocimiento especiales, porque el mensaje de respuesta de un servidor se considera un reconocimiento del mensaje de solicitud del cliente. De manera similar, una llamada posterior de un cliente puede considerarse como un acuse de recibo del mensaje de respuesta de un servidor. Como hemos visto, la comunicación

Figura 5.5 Protocolos de intercambio RPC

Nombre	*Mensajes enviados por*		
	Cliente	*Servidor*	*Cliente*
R	*Pedido*		
RR	*Pedido*	*Responder*	
RRA	*Pedido*	*Responder*	*Confirmar respuesta*

las fallas debidas a la pérdida de datagramas UDP pueden enmascararse mediante la retransmisión de solicitudes con filtrado duplicado y el almacenamiento de respuestas en un historial para su retransmisión.

El protocolo RRA se basa en el intercambio de tres mensajes: solicitud-respuestareconocimiento de respuesta. El*Confirmar respuesta*mensaje contiene el*ID de solicitud* del mensaje de respuesta que se reconoce. Esto permitirá que el servidor descarte entradas de su historial. la llegada de un*ID de solicitud*en un mensaje de acuse de recibo se interpretará como un acuse de recibo de todos los mensajes de respuesta con menor*ID de solicitud*s, por lo que la pérdida de un mensaje de confirmación es inofensiva. Aunque el intercambio implica un mensaje adicional, no es necesario que bloquee al cliente, ya que el acuse de recibo puede transmitirse después de que se haya dado la respuesta al cliente. Sin embargo, utiliza recursos de procesamiento y de red. El ejercicio 5.10 sugiere una optimización del protocolo RRA.

Uso de flujos TCP para implementar el protocolo de solicitud-respuesta •Sección 4.2.3 mencionada que a menudo es difícil decidir sobre un tamaño apropiado para el búfer en el que recibir datagramas. En el protocolo de solicitud-respuesta, esto se aplica a los búfer utilizados por el servidor para recibir mensajes de solicitud y por el cliente para recibir respuestas. La longitud limitada de los datagramas (generalmente 8 kilobytes) puede no considerarse adecuada para su uso en sistemas RMI o RPC transparentes, ya que los argumentos o los resultados de los procedimientos pueden ser de cualquier tamaño.

El deseo de evitar la implementación de protocolos multipaquete es una de las razones para elegir implementar protocolos de solicitud-respuesta sobre flujos TCP, lo que permite que se transmitan argumentos y resultados de cualquier tamaño. En particular, la serialización de objetos de Java es un protocolo de transmisión que permite enviar argumentos y resultados a través de transmisiones entre el cliente y el servidor, lo que hace posible que las colecciones de objetos de cualquier tamaño se transmitan de manera confiable. Si se utiliza el protocolo TCP, garantiza que los mensajes de solicitud y respuesta se entreguen de manera confiable, por lo que no es necesario que el protocolo de solicitud y respuesta se ocupe de la retransmisión de mensajes y el filtrado de duplicados o de historiales. Además, el mecanismo de control de flujo permite pasar grandes argumentos y resultados sin tomar medidas especiales para evitar abrumar al destinatario. Por lo tanto, se elige el protocolo TCP para los protocolos de solicitud y respuesta porque puede simplificar su implementación. Si se envían solicitudes y respuestas sucesivas entre el mismo par cliente-servidor a través del mismo flujo, no es necesario que la sobrecarga de la conexión se aplique a todas las invocaciones remotas. Además, la sobrecarga debida a los mensajes de reconocimiento de TCP se reduce cuando un mensaje de respuesta sigue poco después de un mensaje de solicitud.

Sin embargo, si la aplicación no requiere todas las facilidades que ofrece TCP, se puede implementar un protocolo especialmente diseñado y más eficiente sobre UDP. Por ejemplo, Sun NFS no requiere soporte para mensajes de tamaño ilimitado, ya que

transmite bloques de archivos de tamaño fijo entre el cliente y el servidor. Además, sus operaciones están diseñadas para ser idempotentes, por lo que no importa si las operaciones se ejecutan más de una vez para retransmitir mensajes de respuesta perdidos, por lo que no es necesario mantener un historial.

HTTP: un ejemplo de un protocolo de solicitud-respuesta •El capítulo 1 introdujo el hipertexto Protocolo de transferencia (HTTP) utilizado por los clientes del navegador web para realizar solicitudes a los servidores web y recibir respuestas de ellos. En resumen, los servidores web administran los recursos implementados de diferentes maneras:

- como datos, por ejemplo, el texto de una página HTML, una imagen o la clase de un applet;
- como un programa, por ejemplo, servlets [java.sun.com III], o programas PHP o Python que se ejecutan en el servidor web.

Las solicitudes de los clientes especifican una URL que incluye el nombre de host DNS de un servidor web y un número de puerto opcional en el servidor web, así como el identificador de un recurso en ese servidor.

HTTP es un protocolo que especifica los mensajes involucrados en un intercambio de solicitud-respuesta, los métodos, argumentos y resultados, y las reglas para representarlos (ordenarlos) en los mensajes. Admite un conjunto fijo de métodos (*CONSEGUIR*,*PONER*, *CORREO,* etc) que son aplicables a todos los recursos del servidor. Es diferente a los protocolos descritos anteriormente, donde cada servicio tiene su propio conjunto de operaciones. Además de invocar métodos en recursos web, el protocolo permite la negociación de contenido y la autenticación con contraseña:

Negociación de contenido: las solicitudes de los clientes pueden incluir información sobre qué representaciones de datos pueden aceptar (por ejemplo, idioma o tipo de medio), lo que permite al servidor elegir la representación más adecuada para el usuario.

Autenticación: Las credenciales y los desafíos se utilizan para admitir la autenticación con contraseña. En el primer intento de acceder a un área protegida por contraseña, la respuesta del servidor contiene un desafío aplicable al recurso. El capítulo 11 explica los desafíos. Cuando un cliente recibe un desafío, hace que el usuario escriba un nombre y una contraseña y envía las credenciales asociadas con las solicitudes posteriores.

HTTP se implementa sobre TCP. En la versión original del protocolo, cada interacción cliente-servidor constaba de los siguientes pasos:

- El cliente solicita y el servidor acepta una conexión en el puerto del servidor predeterminado o en un puerto especificado en la URL.
- El cliente envía un mensaje de solicitud al servidor.
- El servidor envía un mensaje de respuesta al cliente.
- La conexión está cerrada.

Sin embargo, establecer y cerrar una conexión para cada intercambio de solicitud-respuesta es costoso, sobrecarga el servidor y hace que se envíen demasiados mensajes a través de la red. Teniendo en cuenta que los navegadores generalmente realizan múltiples solicitudes al mismo servidor, por ejemplo, para obtener las imágenes en una página recién suministrada, una versión posterior del protocolo (HTTP 1.1, consulte RFC 2616 [Fielding*et al.*1999]) usos*conexiones persistentes*– conexiones que permanecen abiertas a lo largo de una serie de intercambios de solicitud-respuesta entre el cliente

Figura 5.6 HTTP *Pedido*mensaje

método	*URL o nombre de ruta*	*Cuerpo del mensaje de los encabezados de la versión HTTP*		
CONSEGUIR	http://www.dcs.qmul.ac.uk/index.html	HTTP/1.1		

y servidor El cliente o el servidor pueden cerrar una conexión persistente en cualquier momento enviando una indicación al otro participante. Los servidores cerrarán una conexión persistente cuando haya estado inactiva durante un período de tiempo. Es posible que un cliente reciba un mensaje del servidor que indique que la conexión está cerrada mientras está enviando otra solicitud o solicitudes. En tales casos, el navegador reenviará las solicitudes sin la participación del usuario, siempre que las operaciones involucradas sean idempotentes. Por ejemplo, el método *CONSEGUIR*descrito a continuación es idempotente. Cuando se trata de operaciones no idempotentes, el navegador debe consultar al usuario qué hacer a continuación.

Las solicitudes y las respuestas se ordenan en mensajes como cadenas de texto ASCII, pero los recursos se pueden representar como secuencias de bytes y se pueden comprimir. El uso de texto en la representación de datos externos ha simplificado el uso de HTTP para los programadores de aplicaciones que trabajan directamente con el protocolo. En este contexto, una representación textual no agrega mucho a la longitud de los mensajes.

Los recursos de datos se proporcionan como estructuras similares a MIME en argumentos y resultados. Las Extensiones de correo de Internet multipropósito (MIME), especificadas en RFC 2045 [Freed y Borenstein 1996], son un estándar para enviar datos de varias partes que contienen, por ejemplo, texto, imágenes y sonido en mensajes de correo electrónico. Los datos tienen el prefijo de su tipo MIME para que el destinatario sepa cómo manejarlos. Un tipo MIME especifica un tipo y un subtipo, por ejemplo, *Texto sin formato*, *texto/html*, *imagen/gif* o *imagen/jpeg*. Los clientes también pueden especificar los tipos MIME que están dispuestos a aceptar.

Métodos HTTP •Cada solicitud de cliente especifica el nombre de un método que se aplicará a un recurso en el servidor y la URL de ese recurso. La respuesta informa sobre el estado de la solicitud. Las solicitudes y respuestas también pueden contener datos de recursos, el contenido de un formulario o la salida de un recurso de programa que se ejecuta en el servidor web. Los métodos incluyen lo siguiente:

CONSEGUIR: Solicita el recurso cuya URL se proporciona como argumento. Si la URL hace referencia a datos, el servidor web responde devolviendo los datos identificados por esa URL. Si la URL se refiere a un programa, el servidor web ejecuta el programa y devuelve su salida al cliente. Se pueden agregar argumentos a la URL; Por ejemplo, *CONSEGUIR*se puede utilizar para enviar el contenido de un formulario a un programa como argumento. El*CONSEGUIR*la operación se puede condicionar a la fecha en que se modificó por última vez un recurso.*CONSEGUIR*también se puede configurar para obtener partes de los datos.

Con*CONSEGUIR*, toda la información para la solicitud se proporciona en la URL (ver, por ejemplo, la cadena de consulta en la Sección 1.6).

CABEZA: Esta solicitud es idéntica a*CONSEGUIR*, pero no devuelve ningún dato. Sin embargo, sí devuelve toda la información sobre los datos, como la hora de la última modificación, su tipo o su tamaño.

CORREO: especifica la URL de un recurso (por ejemplo, un programa) que puede tratar los datos proporcionados en el cuerpo de la solicitud. El procesamiento realizado sobre los datos depende de la función del programa especificado en la URL. Este método se usa cuando la acción puede cambiar los datos en el servidor. Está diseñado para hacer frente a:

- proporcionar un bloque de datos a un proceso de manejo de datos, como un servlet; por ejemplo, enviar un formulario web para comprar algo en un sitio web;
- publicar un mensaje en una lista de correo o actualizar los detalles de los miembros de la lista;
- extender una base de datos con una operación de agregar.

PONER: Solicita que los datos proporcionados en la solicitud se almacenen con la URL dada como su identificador, ya sea como una modificación de un recurso existente o como un nuevo recurso.

BORRAR: El servidor elimina el recurso identificado por la URL dada. Es posible que los servidores no siempre permitan esta operación, en cuyo caso la respuesta indica un error.

OPCIONES: El servidor proporciona al cliente una lista de métodos que permite aplicar a la URL dada (por ejemplo*CONSEGUIR*,*CABEZA*,*PONER*) y sus requisitos especiales.

RASTRO: El servidor devuelve el mensaje de solicitud. Se utiliza con fines de diagnóstico.

las operaciones*PONER*y*BORRAR*son idempotentes, pero*CORREO*no es necesariamente así porque puede cambiar el estado de un recurso. los otros son*seguro*operaciones en que no cambian nada.

Las solicitudes descritas anteriormente pueden ser interceptadas por un servidor proxy (consulte la Sección 2.3.1). las respuestas a*CONSEGUIR*y*CABEZA*puede ser almacenado en caché por servidores proxy.

Contenido del mensaje •El*Pedido*El mensaje especifica el nombre de un método, la URL de un recurso, la versión del protocolo, algunos encabezados y un cuerpo de mensaje opcional. La Figura 5.6 muestra el contenido de un HTTP*Pedido*mensaje cuyo método es*CONSEGUIR*. Cuando la URL especifica un recurso de datos, el*CONSEGUIR*El método no tiene un cuerpo de mensaje.

Las solicitudes a los proxies necesitan la URL absoluta, como se muestra en la Figura 5.6. Las solicitudes a los servidores de origen (el servidor de origen es donde reside el recurso) especifican un nombre de ruta y brindan el nombre DNS del servidor de origen en un*Anfitrión*campo de encabezado. Por ejemplo,

OBTENER /index.html Servidor
HTTP/1.1: www.dcs.qmul.ac.uk

En general, los campos de encabezado contienen modificadores de solicitud e información del cliente, como condiciones sobre la última fecha de modificación del recurso o tipos de contenido aceptables (por ejemplo, texto HTML, audio o imágenes JPEG). Se puede utilizar un campo de autorización para proporcionar las credenciales del cliente en forma de certificado que especifique sus derechos para acceder a un recurso.

A*Responder*El mensaje especifica la versión del protocolo, un código de estado y un "motivo", algunos encabezados y un cuerpo de mensaje opcional, como se muestra en la Figura 5.7. El código de estado y el motivo brindan un informe sobre el éxito o no del servidor en la realización de la solicitud: el primero es un número entero de tres dígitos para que lo interprete un programa, y el segundo es una frase textual que puede ser entendida por una persona. Los campos de encabezado se utilizan para pasar información adicional sobre el servidor o el acceso al recurso. Por ejemplo, si la solicitud requiere autenticación, el estado de la respuesta indica esto y un encabezado

Figura 5.7 HTTP*Responder*mensaje

versión HTTP	*código de estado*	*motivo*	*encabezados*	*cuerpo del mensaje*
HTTP/1.1	200	DE ACUERDO		datos de recursos

El campo contiene un desafío. Algunos retornos de estado tienen efectos bastante complejos. En particular, una respuesta de estado 303 le dice al navegador que busque en una URL diferente, que se proporciona en un campo de encabezado en la respuesta. Está diseñado para usarse en una respuesta de un programa activado por un *CORREO*solicitud cuando el programa necesita redirigir el navegador a un recurso seleccionado.

El cuerpo del mensaje en los mensajes de solicitud o respuesta contiene los datos asociados con la URL especificada en la solicitud. El cuerpo del mensaje tiene sus propios encabezados que especifican información sobre los datos, como su longitud, su tipo MIME, su conjunto de caracteres, su codificación de contenido y la última fecha en que se modificó. El campo de tipo MIME especifica el tipo de datos, por ejemplo*imagen/jpeg*o*Texto sin formato*. El campo de codificación de contenido especifica el algoritmo de compresión que se utilizará

5.3 Llamada a procedimiento remoto

Como se mencionó en el Capítulo 2, el concepto de llamada a procedimiento remoto (RPC) representa un gran avance intelectual en la computación distribuida, con el objetivo de hacer que la programación de sistemas distribuidos se vea similar, si no idéntica, a la programación convencional, es decir, lograr un alto nivel de transparencia en la distribución. Esta unificación se consigue de una forma muy sencilla, extendiendo la abstracción de una llamada a procedimiento a entornos distribuidos. En particular, en RPC, los procedimientos en máquinas remotas se pueden llamar como si fueran procedimientos en el espacio de direcciones local. El sistema RPC subyacente luego oculta aspectos importantes de la distribución, incluida la codificación y decodificación de parámetros y resultados, el paso de mensajes y la preservación de la semántica requerida para la llamada al procedimiento.

5.3.1 Problemas de diseño para RPC

Antes de analizar la implementación de los sistemas RPC, analizamos tres cuestiones que son importantes para comprender este concepto:

- el estilo de programación promovido por RPC – programación con interfaces;
- la semántica de llamada asociada con RPC;
- la cuestión clave de la transparencia y cómo se relaciona con las llamadas a procedimientos remotos.

Programación con interfaces •La mayoría de los lenguajes de programación modernos proporcionan un medio para organizar un programa como un conjunto de módulos que pueden comunicarse entre sí. La comunicación entre módulos puede ser por medio de llamadas de procedimiento entre módulos

o por acceso directo a las variables en otro módulo. Con el fin de controlar las posibles interacciones entre módulos, un explícito *interfaz* se define para cada módulo. La interfaz de un módulo especifica los procedimientos y las variables a las que se puede acceder desde otros módulos. Los módulos se implementan para ocultar toda la información sobre ellos excepto la que está disponible a través de su interfaz. Mientras su interfaz permanezca igual, la implementación puede cambiarse sin afectar a los usuarios del módulo.

Interfaces en sistemas distribuidos: En un programa distribuido, los módulos pueden ejecutarse en procesos separados. En el modelo cliente-servidor, en particular, cada servidor proporciona un conjunto de procedimientos que están disponibles para que los utilicen los clientes. Por ejemplo, un servidor de archivos proporcionaría procedimientos para leer y escribir archivos. El término *interfaz de servicio* se utiliza para referirse a la especificación de los procedimientos que ofrece un servidor, definiendo los tipos de los argumentos de cada uno de los procedimientos.

La programación con interfaces en sistemas distribuidos tiene una serie de ventajas, derivadas de la importante separación entre interfaz e implementación:

- Al igual que con cualquier forma de programación modular, los programadores solo se preocupan por la abstracción que ofrece la interfaz de servicio y no necesitan estar al tanto de los detalles de implementación.
- Extrapolando a sistemas distribuidos (potencialmente heterogéneos), los programadores tampoco necesitan conocer el lenguaje de programación o la plataforma subyacente utilizada para implementar el servicio (un paso importante hacia la gestión de la heterogeneidad en los sistemas distribuidos).
- Este enfoque proporciona un soporte natural para la evolución del software en el sentido de que las implementaciones pueden cambiar mientras la interfaz (la vista externa) siga siendo la misma. Más correctamente, la interfaz también puede cambiar siempre que siga siendo compatible con el original.

La definición de las interfaces de servicio está influenciada por la naturaleza distribuida de la infraestructura subyacente:

- No es posible que un módulo de cliente que se ejecuta en un proceso acceda a las variables en un módulo en otro proceso. Por lo tanto, la interfaz de servicio no puede especificar el acceso directo a las variables. Tenga en cuenta que las interfaces CORBA IDL pueden especificar atributos, lo que parece romper esta regla. Sin embargo, no se accede a los atributos directamente sino por medio de algunos procedimientos getter y setter agregados automáticamente a la interfaz.

- Los mecanismos de paso de parámetros que se utilizan en las llamadas a procedimientos locales, por ejemplo, llamada por valor y llamada por referencia, no son adecuados cuando la persona que llama y el procedimiento se encuentran en procesos diferentes. En particular, no se admite la llamada por referencia. Más bien, la especificación de un procedimiento en la interfaz de un módulo en un programa distribuido describe los parámetros como *aporte* o *producción*, o a veces ambos. *Aporte* Los parámetros se pasan al servidor remoto enviando los valores de los argumentos en el mensaje de solicitud y luego proporcionándolos como argumentos a la operación que se ejecutará en el servidor. *Producción* Los parámetros se devuelven en el mensaje de respuesta y se utilizan como resultado de la llamada o para reemplazar los valores de la correspondiente

Figura 5.8 Ejemplo de CORBA IDL

```
// En el archivo
Person.idl struct Person {
        nombre de cadena;
        lugar de la cuerda;
        año largo;
};
interfaz PersonList {
        nombre de lista de cadenas de atributos de solo
        lectura; void addPerson(en Persona p) ;
        void getPerson(en cadena nombre, fuera Persona p);
        numero largo();
};
```

variables en el entorno de llamada. Cuando se utiliza un parámetro tanto para la entrada como para la salida, el valor debe transmitirse tanto en los mensajes de solicitud como de respuesta.

- Otra diferencia entre los módulos locales y remotos es que las direcciones en un proceso no son válidas en otro remoto. Por lo tanto, las direcciones no pueden pasarse como argumentos ni devolverse como resultado de llamadas a módulos remotos.

Estas restricciones tienen un impacto significativo en la especificación de los lenguajes de definición de interfaz, como se analiza a continuación.

Lenguajes de definición de interfaz:Un mecanismo RPC se puede integrar con un lenguaje de programación particular si incluye una notación adecuada para definir interfaces, lo que permite que los parámetros de entrada y salida se asignen al uso normal de parámetros del lenguaje. Este enfoque es útil cuando todas las partes de una aplicación distribuida se pueden escribir en el mismo idioma. También es conveniente porque le permite al programador usar un solo lenguaje, por ejemplo, Java, para la invocación local y remota.

Sin embargo, muchos servicios útiles existentes están escritos en C++ y otros lenguajes. Sería beneficioso permitir que los programas escritos en una variedad de lenguajes, incluido Java, accedan a ellos de forma remota.*Lenguajes de definición de interfaz*(IDL) están diseñados para permitir que los procedimientos implementados en diferentes lenguajes se invoquen entre sí. Un IDL proporciona una notación para definir interfaces en las que cada uno de los parámetros de una operación puede describirse como entrada o salida además de tener su tipo especificado.

La Figura 5.8 muestra un ejemplo simple de CORBA IDL. El*Persona*La estructura es la misma que la utilizada para ilustrar la clasificación en la Sección 4.3.1. La interfaz denominada *lista de personas*especifica los métodos disponibles para RMI en un objeto remoto que implementa esa interfaz. Por ejemplo, el método*añadirPersona*especifica su argumento como*en*, lo que significa que es un*aporte*argumento y el método*obtenerPersona*que recupera una instancia de *Persona*by name especifica su segundo argumento como*afuera*, lo que significa que es un*producción* argumento.

Figura 5.9 Semántica de llamadas

Medidas de tolerancia a fallas			*Semántica de llamadas*
Solicitud de retransmisión mensaje	*Duplicar filtración*	*Vuelva a ejecutar el procedimiento o retransmitir respuesta*	
No	No aplica	No aplica	*Tal vez*
Sí	No	Vuelva a ejecutar el procedimiento	*Al menos una vez*
Sí	Sí	Retransmitir respuesta	*Como máximo una vez*

El concepto de IDL se desarrolló inicialmente para sistemas RPC, pero se aplica igualmente a RMI y también a servicios web. Nuestros casos de estudio incluyen:

- Sun XDR como ejemplo de IDL para RPC (en la Sección 5.3.3);
- CORBA IDL como ejemplo de un IDL para RMI (en el Capítulo 8 y también incluido anteriormente);
- el lenguaje de descripción de servicios web (WSDL), que está diseñado para servicios web compatibles con RPC en toda Internet (consulte la Sección 9.3);
- y búferes de protocolo utilizados en Google para almacenar e intercambiar muchos tipos de información estructurada (consulte la Sección 21.4.1).

Semántica de llamadas RPC •Los protocolos de solicitud y respuesta se discutieron en la Sección 5.2, donde mostramos que*doOperation*se puede implementar de diferentes maneras para proporcionar diferentes garantías de entrega. Las opciones principales son:

Mensaje de solicitud de reintento: controla si se retransmite el mensaje de solicitud hasta que se recibe una respuesta o se asume que el servidor ha fallado.

Filtrado de duplicados: controla cuándo se utilizan las retransmisiones y si filtrar las solicitudes duplicadas en el servidor.

Retransmisión de resultados: controla si se debe mantener un historial de mensajes de resultados para permitir que los resultados perdidos se retransmitan sin volver a ejecutar las operaciones en el servidor.

Las combinaciones de estas opciones conducen a una variedad de semánticas posibles para la confiabilidad de las invocaciones remotas tal como las ve el invocador. La figura 5.9 muestra las opciones de interés, con los nombres correspondientes a la semántica que producen. Tenga en cuenta que para las llamadas a procedimientos locales, la semántica es*Exactamente una vez*, lo que significa que cada procedimiento se ejecuta exactamente una vez (excepto en el caso de falla del proceso). Las opciones de semántica de invocación RPC se definen como sigue.

Tal vez la semántica:Con*tal vez*semántica, la llamada a procedimiento remoto puede ejecutarse una vez o no ejecutarse en absoluto. Tal vez la semántica surge cuando no se aplican medidas de tolerancia a fallas y puede sufrir los siguientes tipos de fallas:

- fallas de omisión si se pierde el mensaje de solicitud o resultado;
- fallos de bloqueo cuando falla el servidor que contiene la operación remota.

Si no se ha recibido el mensaje de resultado después de un tiempo de espera y no hay reintentos, no se sabe si el procedimiento se ha ejecutado. Si el mensaje de solicitud se perdió, entonces el procedimiento no se habrá ejecutado. Por otro lado, es posible que se haya ejecutado el procedimiento y se haya perdido el mensaje de resultado. Se puede producir un error de bloqueo antes o después de que se ejecute el procedimiento. Además, en un sistema asíncrono, el resultado de ejecutar el procedimiento puede llegar después del tiempo de espera.*Tal vez*la semántica es útil solo para aplicaciones en las que se aceptan llamadas fallidas ocasionales.

Semántica de al menos una vez:Con*al menos una vez*semántica, el invocador recibe un resultado, en cuyo caso el invocador sabe que el procedimiento se ejecutó al menos una vez, o una excepción que le informa que no se recibió ningún resultado.*Al menos una vez*la semántica se puede lograr mediante la retransmisión de mensajes de solicitud, lo que enmascara las fallas de omisión de la solicitud o el mensaje de resultado.*Al menos una vez*la semántica puede sufrir los siguientes tipos de fallas:

- fallos de bloqueo cuando falla el servidor que contiene el procedimiento remoto;
- fallas arbitrarias: en los casos en que se retransmite el mensaje de solicitud, el servidor remoto puede recibirlo y ejecutar el procedimiento más de una vez, lo que posiblemente provoque que se almacenen o devuelvan valores incorrectos.

La Sección 5.2 define un*operación idempotente*como uno que se puede realizar repetidamente con el mismo efecto que si se hubiera realizado exactamente una vez. Las operaciones no idempotentes pueden tener un efecto incorrecto si se realizan más de una vez. Por ejemplo, una operación para aumentar un saldo bancario en $10 debe realizarse una sola vez; si se repitiera, ¡la balanza crecería y crecería! Si las operaciones en un servidor pueden diseñarse de modo que todos los procedimientos en sus interfaces de servicio sean operaciones idempotentes, entonces*al menos una vez*la semántica de llamada puede ser aceptable.

Semántica a lo sumo una vez:Con*como máximo una vez*semántica, la persona que llama recibe un resultado, en cuyo caso la persona que llama sabe que el procedimiento se ejecutó exactamente una vez, o una excepción que le informa que no se recibió ningún resultado, en cuyo caso el procedimiento se habrá ejecutado una vez o no se ejecutará en absoluto.*Como máximo una vez*la semántica se puede lograr usando todas las medidas de tolerancia a fallas descritas en la Figura 5.9. Al igual que en el caso anterior, el uso de reintentos enmascara los fallos de omisión de los mensajes de solicitud o resultado. Este conjunto de medidas de tolerancia a fallas evita fallas arbitrarias al garantizar que para cada RPC un procedimiento nunca se ejecute más de una vez. Sun RPC (discutido en la Sección 5.3.3) proporciona una semántica de llamada al menos una vez.

Transparencia •Los creadores de RPC, Birrell y Nelson [1984], intentaron hacer llamadas a procedimientos remotos lo más parecidas posible a llamadas a procedimientos locales, sin distinción en la sintaxis entre una llamada a procedimiento local y remota. Todas las llamadas necesarias a los procedimientos de clasificación y paso de mensajes se ocultaron al programador que realizaba la llamada. Aunque los mensajes de solicitud se retransmiten después de un tiempo de espera, es transparente para la persona que llama realizar la semántica de las llamadas a procedimientos remotos como la de las llamadas a procedimientos locales.

Más precisamente, volviendo a la terminología del Capítulo 1, RPC se esfuerza por ofrecer al menos transparencia de ubicación y acceso, ocultando la ubicación física del procedimiento (potencialmente remoto) y accediendo también a procedimientos locales y remotos de la misma manera. El middleware también puede ofrecer niveles adicionales de transparencia a RPC.

Sin embargo, las llamadas a procedimientos remotos son más vulnerables a fallas que las locales, ya que involucran una red, otra computadora y otro proceso. Cualquiera que sea la semántica anterior que se elija, siempre existe la posibilidad de que no se reciba ningún resultado y, en caso de falla, es imposible distinguir entre la falla de la red y la del proceso del servidor remoto. Esto requiere que los clientes que realizan llamadas remotas puedan recuperarse de tales situaciones.

La latencia de una llamada a procedimiento remoto es varios órdenes de magnitud mayor que la de uno local. Esto sugiere que los programas que hacen uso de llamadas remotas deben poder tener en cuenta este factor, tal vez minimizando las interacciones remotas. Los diseñadores de Argus [Liskov y Scheifler 1982] sugirieron que una persona que llama debería poder abortar una llamada a un procedimiento remoto que está tardando demasiado de tal manera que no tenga ningún efecto en el servidor. Para permitir esto, el servidor necesitaría poder restaurar las cosas a como estaban antes de llamar al procedimiento. Estos temas se discuten en el Capítulo 16.

Las llamadas a procedimientos remotos también requieren un estilo diferente de paso de parámetros, como se explicó anteriormente. En particular, RPC no ofrece llamada por referencia.

Waldo*et al.*[1994] dicen que la diferencia entre operaciones locales y remotas debe expresarse en la interfaz de servicio, para permitir que los participantes reaccionen de manera consistente ante posibles fallas parciales. Otros sistemas fueron más allá al argumentar que la sintaxis de una llamada remota debería ser diferente a la de una llamada local: en el caso de Argus, el lenguaje se amplió para hacer explícitas las operaciones remotas para el programador.

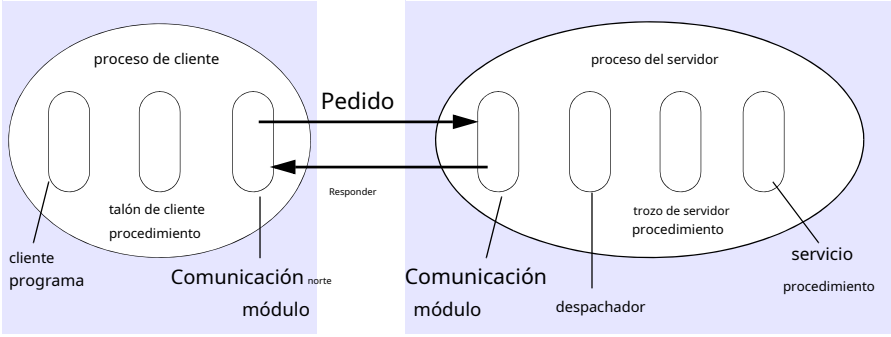
La elección de si RPC debe ser transparente también está disponible para los diseñadores de IDL. Por ejemplo, en algunos IDL, una invocación remota puede generar una excepción cuando el cliente no puede comunicarse con un procedimiento remoto. Esto requiere que el programa cliente maneje tales excepciones, permitiéndole lidiar con tales fallas. Un IDL también puede proporcionar una facilidad para especificar la semántica de llamada de un procedimiento. Esto puede ayudar al diseñador del servicio; por ejemplo, si*al menos una vez*La semántica de llamadas se elige para evitar los gastos generales de*como máximo una vez*, las operaciones deben estar diseñadas para ser idempotentes.

El consenso actual es que las llamadas remotas deben hacerse transparentes en el sentido de que la sintaxis de una llamada remota es la misma que la de una invocación local, pero que la diferencia entre llamadas locales y remotas debe expresarse en sus interfaces.

5.3.2 Implementación de RPC

Los componentes de software necesarios para implementar RPC se muestran en la Figura 5.10. El cliente que accede a un servicio incluye una*procedimiento de trozo*para cada procedimiento en la interfaz de servicio. El procedimiento stub se comporta como un procedimiento local para el cliente, pero en lugar de ejecutar la llamada, ordena el identificador del procedimiento y los argumentos en un mensaje de solicitud, que envía a través de su módulo de comunicación al servidor. Cuando llega el mensaje de respuesta, desarma los resultados. El proceso del servidor contiene un despachador junto con un procedimiento de código auxiliar del servidor y un procedimiento de servicio para cada procedimiento en la interfaz de servicio. El despachador selecciona uno de los procedimientos auxiliares del servidor de acuerdo con el identificador de procedimiento en el mensaje de solicitud. El procedimiento de resguardo del servidor

Figura 5.10 Función de los procedimientos de código auxiliar de cliente y servidor en RPC



luego desarma los argumentos en el mensaje de solicitud, llama al procedimiento de servicio correspondiente y ordena los valores devueltos para el mensaje de respuesta. Los procedimientos de servicio implementan los procedimientos en la interfaz de servicio. Los procedimientos de resguardo del cliente y del servidor y el despachador pueden ser generados automáticamente por un compilador de interfaz a partir de la definición de interfaz del servicio.

RPC generalmente se implementa sobre un protocolo de solicitud-respuesta como los discutidos en la Sección 5.2. Los contenidos de los mensajes de solicitud y respuesta son los mismos que los ilustrados para los protocolos de solicitud y respuesta en la figura 5.4. RPC puede implementarse para tener una de las opciones de semántica de invocación discutidas en la Sección 5.3.1 –*al menos una vez*o *como máximo una vez*generalmente se elige. Para lograr esto, el módulo de comunicación implementará las opciones de diseño deseadas en términos de retransmisión de solicitudes, manejo de duplicados y retransmisión de resultados, como se muestra en la Figura 5.9.

5.3.3 Estudio de caso: Sun RPC

RFC 1831 [Srinivasan 1995a] describe Sun RPC, que fue diseñado para la comunicación cliente-servidor en Sun Network File System (NFS). Sun RPC a veces se llama ONC (Open Network Computing) RPC. Se suministra como parte de varios sistemas operativos Sun y otros UNIX y también está disponible con instalaciones NFS. Los implementadores tienen la opción de usar llamadas a procedimientos remotos sobre UDP o TCP. Cuando Sun RPC se utiliza con UDP, los mensajes de solicitud y respuesta tienen una longitud restringida: en teoría, a 64 kilobytes, pero en la práctica, con más frecuencia, a 8 o 9 kilobytes. Usa*al menos una vez*llamar semántica. Difundir RPC es una opción.

El sistema Sun RPC proporciona un lenguaje de interfaz llamado XDR y un compilador de interfaz llamado*rpcgen*, que está diseñado para usarse con el lenguaje de programación C.

Lenguaje de definición de interfaz •El lenguaje Sun XDR, que se diseñó originalmente para especificar representaciones de datos externos, se amplió para convertirse en un lenguaje de definición de interfaz. Se puede utilizar para definir una interfaz de servicio para Sun RPC especificando un conjunto de definiciones de procedimientos junto con definiciones de tipos compatibles. La notación es bastante primitiva en comparación con la utilizada por CORBA IDL o Java. En particular:

Figura 5.11 Interfaz de archivos en Sun XDR

```
const MAX = 1000; typedef
int FileIdentifier; typedef
int FilePointer; typedef int
Longitud;
datos de estructura {
    longitud int;
    búfer de caracteres [MAX];
};
struct writeargs {
    Identificador de archivo f;
    Posición de FilePointer;
    datos de datos;
};
lecturas de estructura {
    Identificador de archivo f;
    Posición de FilePointer;
    Longitud longitud;
};

programa ARCHIVO LEER ESCRIBIR {
    versión VERSIÓN {
        void ESCRIBIR(escribirargs)=1;          1
        Lectura de datos (readargs) = 2;        2
    }=2;
} = 9999;
```

- La mayoría de los idiomas permiten que se especifiquen los nombres de las interfaces, pero Sun RPC no lo hace; en su lugar, se proporciona un número de programa y un número de versión. Los números de programa se pueden obtener de una autoridad central para permitir que cada programa tenga su propio número único. El número de versión está destinado a cambiarse cuando cambia la firma de un procedimiento. Tanto el programa como el número de versión se pasan en el mensaje de solicitud, por lo que el cliente y el servidor pueden comprobar que están utilizando la misma versión.

- Una definición de procedimiento especifica una firma de procedimiento y un número de procedimiento. El número de procedimiento se utiliza como identificador de procedimiento en los mensajes de solicitud.
- Solo se permite un único parámetro de entrada. Por lo tanto, los procedimientos que requieren múltiples parámetros deben incluirlos como componentes de una sola estructura.
- Los parámetros de salida de un procedimiento se devuelven a través de un único resultado.
- La firma del procedimiento consta del tipo de resultado, el nombre del procedimiento y el tipo del parámetro de entrada. El tipo tanto del resultado como del parámetro de entrada puede especificar un solo valor o una estructura que contiene varios valores.

Por ejemplo, vea la definición XDR en la Figura 5.11 de una interfaz con un par de procedimientos para escribir y leer archivos. El número de programa es 9999 y el número de versión es 2. El*LEER*El procedimiento (línea 2) toma como parámetro de entrada una estructura con tres componentes que especifican un identificador de archivo, una posición en el archivo y la cantidad de bytes requeridos. Su resultado es una estructura que contiene el número de bytes devueltos y los datos del archivo. El*ESCRIBIR*procedimiento (línea 1) no tiene ningún resultado. El*ESCRIBIR*y*LEER*los procedimientos reciben los números 1 y 2. El número 0 está reservado para un procedimiento nulo, que se genera automáticamente y está diseñado para probar si un servidor está disponible.

Este lenguaje de definición de interfaz proporciona una notación para definir constantes, typedefs, estructuras, tipos enumerados, uniones y programas. Typedefs, estructuras y tipos enumerados utilizan la sintaxis del lenguaje C. El compilador de interfaz*rpcgen*se puede utilizar para generar lo siguiente a partir de una definición de interfaz:

- procedimientos de talón de cliente;
- servidor*principal*procedimiento, despachador y procedimientos de stub de servidor;
- Procedimientos de clasificación y desclasificación de XDR para uso del despachador y procedimientos de stub de cliente y servidor.

Vinculante •Sun RPC ejecuta un servicio de enlace local denominado*mapeador de puertos*en un número de puerto conocido en cada computadora. Cada instancia de un asignador de puertos registra el número de programa, el número de versión y el número de puerto en uso por cada servicio que se ejecuta localmente. Cuando un servidor se inicia, registra su número de programa, número de versión y número de puerto con el asignador de puertos local. Cuando un cliente se inicia, descubre el puerto del servidor haciendo una solicitud remota al mapeador de puertos en el host del servidor, especificando el número de programa y el número de versión.

Cuando un servicio tiene múltiples instancias ejecutándose en diferentes computadoras, las instancias pueden usar diferentes números de puerto para recibir solicitudes de clientes. Si un cliente necesita multidifundir una solicitud a todas las instancias de un servicio que utilizan diferentes números de puerto, no puede usar un mensaje de multidifusión de IP directo para este fin. La solución es que los clientes realicen llamadas a procedimientos remotos de multidifusión multidifundiéndolas a todos los mapeadores de puertos, especificando el programa y el número de versión. Cada mapeador de puertos reenvía todas esas llamadas al programa de servicio local correspondiente, si lo hay.

Autenticación. Los mensajes de solicitud y respuesta de RPC de Sun proporcionan campos adicionales que permiten que la información de autenticación se transmita entre el cliente y el servidor. El mensaje de solicitud contiene las credenciales del usuario que ejecuta el programa cliente. Por ejemplo, en el estilo de autenticación UNIX, las credenciales incluyen el*fluido*y*Gid*del usuario Los mecanismos de control de acceso se pueden construir sobre la información de autenticación que se pone a disposición de los procedimientos del servidor a través de un segundo argumento. El programa del servidor es responsable de hacer cumplir el control de acceso al decidir si ejecutar cada llamada de procedimiento de acuerdo con la información de autenticación. Por ejemplo, si el servidor es un servidor de archivos NFS, puede verificar si el usuario tiene suficientes derechos para realizar una operación de archivo solicitada. Se pueden admitir varios protocolos de autenticación diferentes. Éstas incluyen:

- ninguno;
- estilo UNIX, como se describe arriba;
- un estilo en el que se establece una clave compartida para firmar los mensajes RPC;
- Kerberos (ver Capítulo 11).

Un campo en el encabezado RPC indica qué estilo se está utilizando.

Un enfoque más genérico de la seguridad se describe en RFC 2203 [Eisler*y otros*. 1997]. Proporciona el secreto y la integridad de los mensajes RPC, así como la autenticación. Permite que el cliente y el servidor negocien un contexto de seguridad en el que no se aplica seguridad o, en el caso de que se requiera seguridad, se puede aplicar la integridad del mensaje o la privacidad del mensaje, o ambas.

Programas cliente y servidor •Más material sobre Sun RPC está disponible en www.cdk5.net/rmi . Incluye ejemplos de programas cliente y servidor correspondientes a la interfaz definida en la Figura 5.11.

5.4 Invocación de método remoto

Invocación de método remoto(RMI) está estrechamente relacionado con RPC pero se extiende al mundo de los objetos distribuidos. En RMI, un objeto que llama puede invocar un método en un objeto potencialmente remoto. Al igual que con RPC, los detalles subyacentes generalmente están ocultos para el usuario. Los puntos en común entre RMI y RPC son los siguientes:

- Ambos admiten la programación con interfaces, con los beneficios resultantes que se derivan de este enfoque (consulte la Sección 5.3.1).
- Por lo general, ambos se construyen sobre protocolos de solicitud y respuesta y pueden ofrecer una variedad de semánticas de llamadas, como*al menos una vez*y*como máximo una vez*.
- Ambos ofrecen un nivel similar de transparencia, es decir, las llamadas locales y remotas emplean la misma sintaxis, pero las interfaces remotas suelen exponer la naturaleza distribuida de la llamada subyacente, por ejemplo, al admitir excepciones remotas.

Las siguientes diferencias conducen a una mayor expresividad cuando se trata de la programación de aplicaciones y servicios distribuidos complejos.

- El programador puede usar todo el poder expresivo de la programación orientada a objetos en el desarrollo de software de sistemas distribuidos, incluido el uso de objetos, clases y herencia, y también puede emplear metodologías de diseño orientadas a objetos relacionadas y herramientas asociadas.
- Sobre la base del concepto de identidad de objeto en los sistemas orientados a objetos, todos los objetos en un sistema basado en RMI tienen referencias de objetos únicas (ya sean locales o remotas), dichas referencias de objetos también se pueden pasar como parámetros, ofreciendo así parámetros significativamente más ricos. pasar la semántica que en RPC.

El tema del paso de parámetros es particularmente importante en los sistemas distribuidos. RMI permite al programador pasar parámetros no solo por valor, como parámetros de entrada o salida, sino también por referencia de objeto. Pasar referencias es particularmente atractivo si el parámetro subyacente es grande o complejo. El extremo remoto, al recibir una referencia de objeto, puede acceder a este objeto mediante la invocación de un método remoto, en lugar de tener que transmitir el valor del objeto a través de la red.

El resto de esta sección examina el concepto de invocación de métodos remotos con más detalle, analizando inicialmente los problemas clave relacionados con los modelos de objetos distribuidos antes de analizar los problemas de implementación relacionados con RMI, incluida la recolección de elementos no utilizados distribuidos.

5.4.1 Problemas de diseño para RMI

Como se mencionó anteriormente, RMI comparte los mismos problemas de diseño que RPC en términos de programación con interfaces, semántica de llamadas y nivel de transparencia. Se anima al lector a consultar la Sección 5.3.1 para la discusión de estos elementos.

El problema de diseño añadido clave se relaciona con el modelo de objetos y, en particular, lograr la transición de objetos a objetos distribuidos. Primero describimos el modelo de objetos convencional de una sola imagen y luego describimos el modelo de objetos distribuidos.

El modelo de objetos •Un programa orientado a objetos, por ejemplo en Java o C++, consta de una colección de objetos que interactúan, cada uno de los cuales consta de un conjunto de datos y un conjunto de métodos. Un objeto se comunica con otros objetos invocando sus métodos, generalmente pasando argumentos y recibiendo resultados. Los objetos pueden encapsular sus datos y el código de sus métodos. Algunos lenguajes, por ejemplo Java y C++, permiten a los programadores definir objetos a cuyas variables de instancia se puede acceder directamente. Pero para su uso en un sistema de objetos distribuidos, los datos de un objeto deben ser accesibles solo a través de sus métodos.

Referencias de objetos:Se puede acceder a los objetos a través de referencias a objetos. Por ejemplo, en Java, una variable que parece contener un objeto en realidad contiene una referencia a ese objeto. Para invocar un método en un objeto, se proporcionan la referencia del objeto y el nombre del método, junto con los argumentos necesarios. El objeto cuyo método se invoca a veces se denomina*objetivo*y a veces el *receptor*. Las referencias a objetos son valores de primera clase, lo que significa que pueden, por ejemplo, asignarse a variables, pasarse como argumentos y devolverse como resultados de métodos.

Interfaces:Una interfaz proporciona una definición de las firmas de un conjunto de métodos (es decir, los tipos de sus argumentos, valores devueltos y excepciones) sin especificar su implementación. Un objeto proporcionará una interfaz particular si su clase contiene código que implementa los métodos de esa interfaz. En Java, una clase puede implementar varias interfaces y los métodos de una interfaz pueden ser implementados por cualquier clase. Una interfaz también define tipos que se pueden usar para declarar el tipo de variables o de los parámetros y devolver valores de métodos. Tenga en cuenta que las interfaces no tienen constructores.

Acciones :La acción en un programa orientado a objetos es iniciada por un objeto que invoca un método en otro objeto. Una invocación puede incluir información adicional (argumentos) necesarios para llevar a cabo el método. El receptor ejecuta el método apropiado y luego devuelve el control al objeto que lo invoca, a veces proporcionando un resultado. La invocación de un método puede tener tres efectos:

1. El estado del receptor puede cambiar.
2. Se puede crear una instancia de un nuevo objeto, por ejemplo, utilizando un constructor en Java o C++.
3. Pueden tener lugar invocaciones adicionales de métodos en otros objetos.

Como una invocación puede dar lugar a más invocaciones de métodos en otros objetos, una acción es una cadena de invocaciones de métodos relacionados, cada uno de los cuales finalmente regresa.

Excepciones:Los programas pueden encontrar muchos tipos de errores y condiciones inesperadas de diversa gravedad. Durante la ejecución de un método, se pueden descubrir muchos problemas diferentes: por ejemplo, valores inconsistentes en las variables del objeto o fallas en

intenta leer o escribir en archivos o sockets de red. Cuando los programadores necesitan insertar pruebas en su código para tratar todos los posibles casos inusuales o erróneos, esto resta claridad al caso normal. Las excepciones proporcionan una forma limpia de tratar las condiciones de error sin complicar el código. Además, cada encabezado de método enumera explícitamente como excepciones las condiciones de error que podría encontrar, lo que permite a los usuarios del método tratarlas. Un bloque de código se puede definir para*tirar*una excepción siempre que surjan condiciones o errores particulares inesperados. Esto significa que el control pasa a otro bloque de código que*capturas*La excepción. El control no regresa al lugar donde se lanzó la excepción.

Recolección de basura:Es necesario proporcionar un medio para liberar el espacio ocupado por los objetos cuando ya no se necesitan. Un lenguaje como Java, que puede detectar automáticamente cuando ya no se puede acceder a un objeto, recupera el espacio y lo pone a disposición para su asignación a otros objetos. Este proceso se llama*recolección de basura*. Cuando un lenguaje (por ejemplo, C++) no soporta la recolección de basura, el programador tiene que hacer frente a la liberación de espacio asignado a los objetos. Esto puede ser una fuente importante de errores.

Objetos distribuidos •El estado de un objeto consta de los valores de sus variables de instancia. En el paradigma basado en objetos, el estado de un programa se divide en partes separadas, cada una de las cuales está asociada con un objeto. Dado que los programas basados en objetos se dividen lógicamente, la distribución física de objetos en diferentes procesos o computadoras en un sistema distribuido es una extensión natural (el tema de la ubicación se analiza en la Sección 2.3.1).

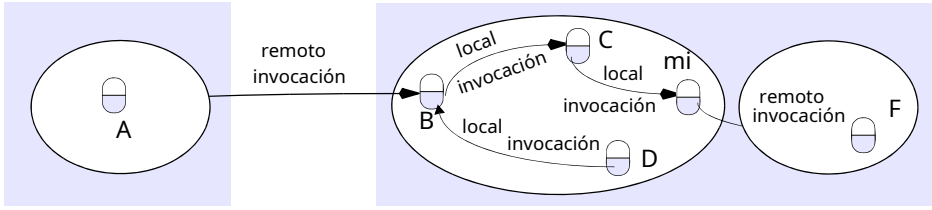
Los sistemas de objetos distribuidos pueden adoptar la arquitectura cliente-servidor. En este caso, los objetos son administrados por servidores y sus clientes invocan sus métodos mediante la invocación de métodos remotos. En RMI, la solicitud del cliente para invocar un método de un objeto se envía en un mensaje al servidor que administra el objeto. La invocación se realiza ejecutando un método del objeto en el servidor y el resultado se devuelve al cliente en otro mensaje. Para permitir cadenas de invocaciones relacionadas, los objetos en los servidores pueden convertirse en clientes de objetos en otros servidores.

Los objetos distribuidos pueden asumir otros modelos arquitectónicos. Por ejemplo, los objetos se pueden replicar para obtener los beneficios habituales de tolerancia a fallas y rendimiento mejorado, y los objetos se pueden migrar con miras a mejorar su rendimiento y disponibilidad.

Tener objetos de cliente y servidor en diferentes procesos impone*encapsulación*. Es decir, solo se puede acceder al estado de un objeto mediante los métodos del objeto, lo que significa que no es posible que los métodos no autorizados actúen sobre el estado. Por ejemplo, la posibilidad de RMI concurrentes de objetos en diferentes computadoras implica que se puede acceder a un objeto concurrentemente. Por lo tanto, surge la posibilidad de accesos conflictivos. Sin embargo, el hecho de que los datos de un objeto sean accedidos únicamente por sus propios métodos permite que los objetos proporcionen métodos para protegerse contra accesos incorrectos. Por ejemplo, pueden usar primitivas de sincronización como variables de condición para proteger el acceso a sus variables de instancia.

Otra ventaja de tratar el estado compartido de un programa distribuido como una colección de objetos es que se puede acceder a un objeto a través de RMI, o se puede copiar en un caché local y acceder directamente, siempre que la implementación de la clase esté disponible localmente.

Figura 5.12 Invocaciones de métodos locales y remotos



El hecho de que se acceda a los objetos solo a través de sus métodos da lugar a otra ventaja de los sistemas heterogéneos, que se pueden usar diferentes formatos de datos en diferentes sitios; estos formatos pasarán desapercibidos para los clientes que usan RMI para acceder a los métodos de los objetos.

El modelo de objetos distribuidos •Esta sección analiza las extensiones del modelo de objetos para que sea aplicable a los objetos distribuidos. Cada proceso contiene una colección de objetos, algunos de los cuales pueden recibir invocaciones locales y remotas, mientras que los otros objetos pueden recibir solo invocaciones locales, como se muestra en la Figura 5.12. Las invocaciones de métodos entre objetos en diferentes procesos, ya sea en la misma computadora o no, se conocen como invocaciones de métodos remotos. Las invocaciones de métodos entre objetos en el mismo proceso son invocaciones de métodos locales.

Nos referimos a objetos que pueden recibir invocaciones remotas como*objetos remotos*. En la Figura 5.12, los objetos B y F son objetos remotos. Todos los objetos pueden recibir invocaciones locales, aunque solo pueden recibirlas de otros objetos que tengan referencias a ellos. Por ejemplo, el objeto C debe tener una referencia al objeto E para que pueda invocar uno de sus métodos. Los siguientes dos conceptos fundamentales están en el corazón del modelo de objetos distribuidos:

Referencias a objetos remotos: Otros objetos pueden invocar los métodos de un objeto remoto si tienen acceso a su*referencia a objetos remotos*. Por ejemplo, una referencia de objeto remoto para B en la Figura 5.12 debe estar disponible para A.

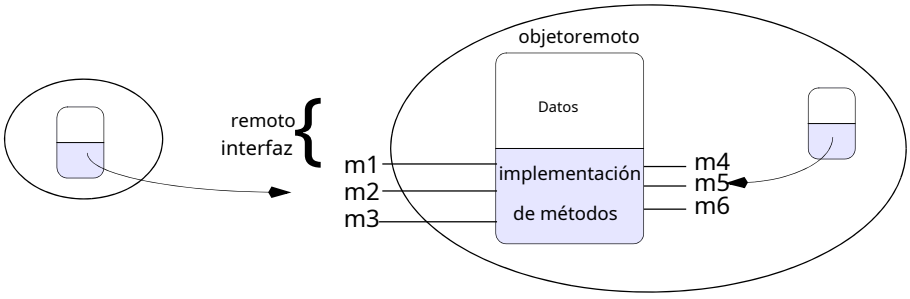
Interfaces remotas: Cada objeto remoto tiene un*interfaz remota*que especifica cuál de sus métodos se puede invocar de forma remota. Por ejemplo, los objetos B y F de la figura 5.12 deben tener interfaces remotas.

A continuación, analizamos las referencias a objetos remotos, las interfaces remotas y otros aspectos del modelo de objetos distribuidos.

Referencias a objetos remotos:La noción de referencia de objeto se amplía para permitir que cualquier objeto que pueda recibir un RMI tenga una referencia de objeto remoto. Una referencia de objeto remoto es un identificador que se puede usar en un sistema distribuido para referirse a un objeto remoto único en particular. Su representación, que generalmente es diferente de la de una referencia de objeto local, se analiza en la Sección 4.3.4. Las referencias a objetos remotos son análogas a las locales en que:

1. El invocador especifica el objeto remoto para recibir una invocación de método remoto como una referencia de objeto remoto.
2. Las referencias a objetos remotos se pueden pasar como argumentos y resultados de invocaciones de métodos remotos.

Figura 5.13 Un objeto remoto y su interfaz remota



Interfaces remotas:La clase de un objeto remoto implementa los métodos de su interfaz remota, por ejemplo, como métodos de instancia pública en Java. Los objetos en otros procesos pueden invocar solo los métodos que pertenecen a su interfaz remota, como se muestra en la Figura 5.13. Los objetos locales pueden invocar los métodos en la interfaz remota así como otros métodos implementados por un objeto remoto. Tenga en cuenta que las interfaces remotas, como todas las interfaces, no tienen constructores.

El sistema CORBA proporciona un lenguaje de definición de interfaz (IDL), que se utiliza para definir interfaces remotas. Consulte la Figura 5.8 para ver un ejemplo de una interfaz remota definida en CORBA IDL. Las clases de objetos remotos y los programas cliente pueden implementarse en cualquier lenguaje para el que esté disponible un compilador IDL, como C++, Java o Python. Los clientes de CORBA no necesitan usar el mismo lenguaje que el objeto remoto para invocar sus métodos de forma remota.

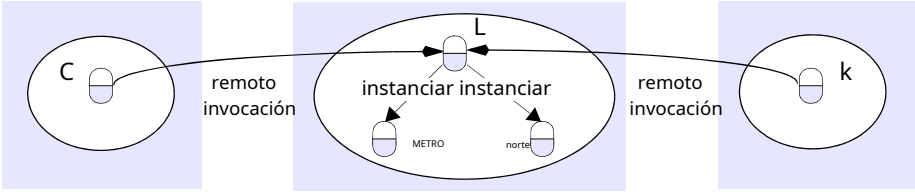
En Java RMI, las interfaces remotas se definen de la misma manera que cualquier otra interfaz de Java. Adquieren su capacidad de ser interfaces remotas extendiendo una interfaz denominada*Remoto*. Tanto CORBA IDL (Capítulo 8) como Java admiten la herencia múltiple de interfaces. Es decir, se permite que una interfaz amplíe una o más interfaces.

Acciones en un sistema de objetos distribuidos •Como en el caso no distribuido, una acción se inicia mediante la invocación de un método, lo que puede dar lugar a más invocaciones de métodos en otros objetos. Pero en el caso distribuido, los objetos involucrados en una cadena de invocaciones relacionadas pueden estar ubicados en diferentes procesos o diferentes computadoras. Cuando una invocación cruza el límite de un proceso o computadora, se usa RMI y la referencia remota del objeto debe estar disponible para el invocador. En la Figura 5.12, el objeto A necesita contener una referencia de objeto remoto al objeto B. Las referencias de objetos remotos se pueden obtener como resultado de invocaciones de métodos remotos. Por ejemplo, el objeto A de la figura 5.12 podría obtener una referencia remota al objeto F desde el objeto B.

Cuando una acción conduce a la creación de instancias de un nuevo objeto, ese objeto normalmente vivirá dentro del proceso donde se solicita la creación de instancias, por ejemplo, donde se usó el constructor. Si el objeto recién instanciado tiene una interfaz remota, será un objeto remoto con una referencia de objeto remoto.

Las aplicaciones distribuidas pueden proporcionar a los objetos remotos métodos para crear instancias de objetos a los que RMI puede acceder, proporcionando así de manera efectiva el efecto de instanciación remota de objetos. Por ejemplo, si el objeto L de la figura 5.14 contiene un método para crear objetos remotos, las invocaciones remotas desde C y K podrían conducir a la instanciación de los objetos M y N, respectivamente.

Figura 5.14 Instanciación de objetos remotos



Recolección de basura en un sistema de objetos distribuidos:Si un lenguaje, por ejemplo Java, admite la recolección de elementos no utilizados, entonces cualquier sistema RMI asociado debería permitir la recolección de elementos no utilizados de objetos remotos. La recolección de basura distribuida generalmente se logra mediante la cooperación entre el recolector de basura local existente y un módulo agregado que lleva a cabo una forma de recolección de basura distribuida, generalmente basada en el recuento de referencias. La Sección 5.4.3 describe dicho esquema en detalle. Si la recolección de elementos no utilizados no está disponible, se deben eliminar los objetos remotos que ya no se necesitan.

Excepciones:Cualquier invocación remota puede fallar por razones relacionadas con que el objeto invocado se encuentre en un proceso o computadora diferente al que la invocó. Por ejemplo, es posible que el proceso que contiene el objeto remoto se bloquee o que esté demasiado ocupado para responder, o que se pierda la invocación o el mensaje de resultado. Por lo tanto, la invocación de métodos remotos debería poder generar excepciones, como tiempos de espera que se deben a la distribución, así como los generados durante la ejecución del método invocado. Ejemplos de esto último son un intento de leer más allá del final de un archivo o acceder a un archivo sin los permisos correctos.

CORBA IDL proporciona una notación para especificar excepciones a nivel de aplicación, y el sistema subyacente genera excepciones estándar cuando ocurren errores debido a la distribución. Los programas cliente de CORBA necesitan poder manejar excepciones. Por ejemplo, un programa cliente de C++ utilizará los mecanismos de excepción de C++.

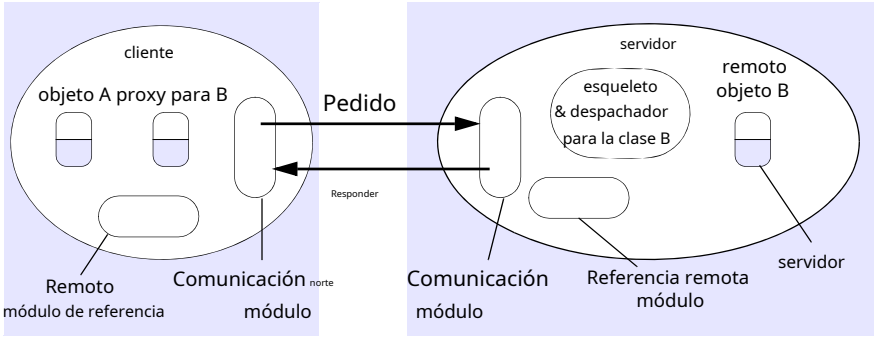
5.4.2 Implementación de RMI

Varios objetos y módulos separados están involucrados en lograr una invocación de método remoto. Estos se muestran en la figura 5.15, en la que un objeto A de nivel de aplicación invoca un método en un objeto B remoto de nivel de aplicación para el que tiene una referencia de objeto remoto. Esta sección analiza las funciones de cada uno de los componentes que se muestran en esa figura, tratando primero con los módulos de comunicación y referencia remota y luego con el software RMI que se ejecuta sobre ellos.

Luego exploramos los siguientes temas relacionados: la generación de proxies, el enlace de nombres a sus referencias a objetos remotos, la activación y pasivación de objetos y la ubicación de objetos desde sus referencias a objetos remotos.

Módulo de comunicación •Los dos módulos de comunicación que cooperan llevan a cabo el protocolo de solicitud-respuesta, que transmite*pedido*y*responder*mensajes entre el cliente y el servidor. Los contenidos de*pedido*y*responder*Los mensajes se muestran en la Figura 5.4. El módulo de comunicación utiliza sólo los tres primeros elementos, que especifican el tipo de mensaje, su*ID de solicitud*y la referencia remota del objeto a invocar. El*operaciónId*y

Figura 5.15 El papel del proxy y el esqueleto en la invocación de métodos remotos



toda la clasificación y desclasificación son responsabilidad del software RMI, que se analiza a continuación. Los módulos de comunicación son juntos responsables de proporcionar una semántica de invocación específica, por ejemplo*como máximo una vez*.

El módulo de comunicación en el servidor selecciona el despachador para la clase del objeto a invocar, pasando su referencia local, que obtiene del módulo de referencia remota a cambio del identificador de objeto remoto en el*pedido*mensaje. El papel del despachador se analiza en la próxima sección sobre el software RMI.

Módulo de referencia remota •Un módulo de referencia remota es responsable de traducir entre referencias de objetos locales y remotas y de crear referencias de objetos remotos. Para apoyar sus responsabilidades, el módulo de referencia remota en cada proceso tiene un*tabla de objetos remotos*que registra la correspondencia entre las referencias a objetos locales en ese proceso y las referencias a objetos remotos (que son de todo el sistema). La tabla incluye:

- Una entrada para todos los objetos remotos retenidos por el proceso. Por ejemplo, en la Figura 5.15 el objeto remoto B se registrará en la tabla en el servidor.
- Una entrada para cada proxy local. Por ejemplo, en la figura 5.15, el proxy para B se registrará en la tabla del cliente.

La función de un proxy se analiza en la subsección sobre el software RMI. Las acciones del módulo de referencia remota son las siguientes:

- Cuando se va a pasar un objeto remoto como argumento o resultado por primera vez, se le pide al módulo de referencia remota que cree una referencia de objeto remoto, que agrega a su tabla.
- Cuando llega una referencia a un objeto remoto en un*pedido*o*responder*mensaje, se solicita al módulo de referencia remota la referencia de objeto local correspondiente, que puede referirse a un proxy o a un objeto remoto. En el caso de que la referencia del objeto remoto no esté en la tabla, el software RMI crea un nuevo proxy y le pide al módulo de referencia remota que lo agregue a la tabla.

Los componentes del software RMI llaman a este módulo cuando están ordenando y ordenando referencias a objetos remotos. Por ejemplo, cuando un*pedido*llega un mensaje, la tabla se utiliza para averiguar qué objeto local se va a invocar.

sirvientes •A*servidor*es una instancia de una clase que proporciona el cuerpo de un objeto remoto. Es el servidor que eventualmente maneja las solicitudes remotas pasadas por el esqueleto correspondiente. Los servidores viven dentro de un proceso de servidor. Se crean cuando se crean instancias de objetos remotos y permanecen en uso hasta que ya no se necesitan, y finalmente se recolectan como elementos no utilizados o se eliminan.

El software RMI •Este consiste en una capa de software entre los objetos de nivel de aplicación y los módulos de comunicación y referencia remota. Las funciones de los objetos de middleware que se muestran en la figura 5.15 son las siguientes:

Apoderado: La función de un proxy es hacer que la invocación de métodos remotos sea transparente para los clientes al comportarse como un objeto local para el invocador; pero en lugar de ejecutar una invocación, la reenvía en un mensaje a un objeto remoto. Oculta los detalles de la referencia del objeto remoto, la ordenación de argumentos, la ordenación de resultados y el envío y recepción de mensajes del cliente. Hay un proxy para cada objeto remoto para el que un proceso tiene una referencia de objeto remoto. La clase de un proxy implementa los métodos en la interfaz remota del objeto remoto que representa, lo que garantiza que las invocaciones de métodos remotos sean adecuadas para el tipo de objeto remoto. Sin embargo, el proxy los implementa de manera bastante diferente. Cada método del proxy calcula una referencia al objeto de destino, su propio*operaciónId*y sus argumentos en un *pedido*mensaje y lo envía al destino. Luego espera la*responder* mensaje, lo desarma y devuelve los resultados al invocador.

Despachador: Un servidor tiene un despachador y un esqueleto para cada clase que representa un objeto remoto. En nuestro ejemplo, el servidor tiene un despachador y un esqueleto para la clase de objeto remoto B. El despachador recibe*pedido*mensajes del módulo de comunicación. utiliza el*operaciónId*para seleccionar el método apropiado en el esqueleto, pasando el*pedido*mensaje. El despachador y el proxy usan la misma asignación de*ID de operación*a los métodos de la interfaz remota.

Esqueleto: La clase de un objeto remoto tiene un*esqueleto*, que implementa los métodos en la interfaz remota. Se implementan de forma bastante diferente a los métodos del sirviente que encarna un objeto remoto. Un método esqueleto desarma los argumentos en el*pedido*mensaje e invoca el método correspondiente en el servidor. Espera a que se complete la invocación y luego ordena el resultado, junto con cualquier excepción, en un *responder*mensaje al método del proxy de envío.

Las referencias a objetos remotos se organizan en la forma que se muestra en la Figura 4.13, que incluye información sobre la interfaz remota del objeto remoto, por ejemplo, el nombre de la interfaz remota o la clase del objeto remoto. Esta información permite determinar la clase de proxy para que se pueda crear un nuevo proxy cuando sea necesario. Por ejemplo, el nombre de la clase de proxy se puede generar agregando_*apoderado*al nombre de la interfaz remota.

Generación de las clases para proxies, dispatchers y skeletons •Las clases para el
El proxy, el despachador y el esqueleto utilizados en RMI son generados automáticamente por un compilador de interfaz. Por ejemplo, en la implementación Orbix de CORBA, las interfaces de objetos remotos se definen en CORBA IDL, y el compilador ~~de interfaz se p~~uede usar para generar las clases para proxies, despachadores y esqueletos en C++ o en Java [www.iona.com]. Para Java RMI, el conjunto de métodos que ofrece un objeto remoto se define como una interfaz Java

que se implementa dentro de la clase del objeto remoto. El compilador Java RMI genera las clases de proxy, despachador y esqueleto a partir de la clase del objeto remoto.

Invocación dinámica: una alternativa a los proxies •El proxy que acabamos de describir es estático, en el sentido que su clase se genera a partir de una definición de interfaz y luego se compila en el código del cliente. Sin embargo, a veces esto no es práctico. Suponga que un programa cliente recibe una referencia remota a un objeto cuya interfaz remota no estaba disponible en tiempo de compilación. En este caso, necesita otra forma de invocar el objeto remoto.*Invocación dinámica*da al cliente acceso a una representación genérica de una invocación remota como la*doOperation* método utilizado en el Ejercicio 5.18, que está disponible como parte de la infraestructura para RMI (consulte la Sección 5.4.1). El cliente proporcionará la referencia del objeto remoto, el nombre del método y los argumentos para*doOperation*y luego esperar a recibir los resultados.

Tenga en cuenta que aunque la referencia del objeto remoto incluye información sobre la interfaz del objeto remoto, como su nombre, esto no es suficiente: los nombres de los métodos y los tipos de argumentos son necesarios para realizar una invocación dinámica. CORBA proporciona esta información a través de un componente llamado Repositorio de interfaz, que se describe en el Capítulo 8.

La interfaz de invocación dinámica no es tan conveniente de usar como proxy, pero es útil en aplicaciones donde algunas de las interfaces de los objetos remotos no se pueden predecir en tiempo de diseño. Un ejemplo de tal aplicación es la pizarra compartida que usamos para ilustrar Java RMI (Sección 5.5), CORBA (Capítulo 8) y servicios web (Sección 9.2.3). Para resumir: la aplicación de pizarra compartida muestra muchos tipos diferentes de formas, como círculos, rectángulos y líneas, pero también debería poder mostrar nuevas formas que no se predijeron cuando se compiló el cliente. Un cliente que utiliza la invocación dinámica puede abordar este desafío. Veremos en la Sección 5.5 que la descarga dinámica de clases a los clientes es una alternativa a la invocación dinámica. Está disponible en Java RMI, un sistema de un solo idioma.

Esqueletos dinámicos:Está claro, a partir del ejemplo anterior, que también puede surgir que un servidor necesite alojar objetos remotos cuyas interfaces no se conocían en el momento de la compilación. Por ejemplo, un cliente puede proporcionar un nuevo tipo de forma al servidor de pizarra compartida para que lo almacene. Un servidor con esqueletos dinámicos podría hacer frente a esta situación. Aplazamos la descripción de esqueletos dinámicos hasta el capítulo sobre CORBA (Capítulo 8). Sin embargo, como veremos en la Sección 5.5, Java RMI soluciona este problema utilizando un despachador genérico y la descarga dinámica de clases al servidor.

Programas de servidor y cliente •El programa servidor contiene las clases para los despachadores y esqueletos, junto con las implementaciones de las clases de todos los servidores que soporta. Además, el programa del servidor contiene un*inicialización* sección (por ejemplo, en una*principal*método en Java o C++). La sección de inicialización se encarga de crear e inicializar al menos uno de los servidores que será alojado por el servidor. Se pueden crear servidores adicionales en respuesta a las solicitudes de los clientes. La sección de inicialización también puede registrar algunos de sus servidores con un archivador (ver más abajo). Generalmente registrará un solo servidor, que puede ser utilizado para acceder al resto.

El programa cliente contendrá las clases de los proxies para todos los objetos remotos que invocará. Puede usar un archivador para buscar referencias de objetos remotos.

Métodos de fábrica:Señalamos anteriormente que las interfaces de objetos remotos no pueden incluir constructores. Esto significa que los sirvientes no se pueden crear mediante invocación remota en constructores. Los servidores se crean en la sección de inicialización o en métodos en una interfaz remota diseñada para ese propósito. El término*método de fábrica*se usa a veces para referirse a un método que crea sirvientes, y un*objeto de fábrica*es un objeto con métodos de fábrica. Cualquier objeto remoto que necesite poder crear nuevos objetos remotos a pedido de los clientes debe proporcionar métodos en su interfaz remota para este propósito. Estos métodos se denominan métodos de fábrica, aunque en realidad son métodos normales.

El aglutinante •Los programas de cliente generalmente requieren un medio para obtener una referencia de objeto remoto para al menos uno de los objetos remotos mantenidos por un servidor. Por ejemplo, en la Figura 5.12, el objeto A requeriría una referencia de objeto remoto para el objeto B. A*aglutinante*en un sistema distribuido es un servicio separado que mantiene una tabla que contiene asignaciones de nombres textuales a referencias de objetos remotos. Los servidores lo utilizan para registrar sus objetos remotos por nombre y los clientes para buscarlos. El capítulo 8 contiene una discusión sobre el servicio de nombres CORBA. El enlazador de Java, RMIregistry, se analiza brevemente en el estudio de caso sobre Java RMI en la Sección 5.5.

Subprocesos del servidor •Cada vez que un objeto ejecuta una invocación remota, esa ejecución puede dar lugar a más invocaciones de métodos en otros objetos remotos, que pueden tardar algún tiempo en volver. Para evitar que la ejecución de una invocación remota retrase la ejecución de otra, los servidores generalmente asignan un subproceso separado para la ejecución de cada invocación remota. Cuando este es el caso, el diseñador de la implementación de un objeto remoto debe tener en cuenta los efectos sobre su estado de ejecuciones concurrentes.

Activación de objetos remotos •Algunas aplicaciones requieren que la información sobreviva durante largos períodos de tiempo. Sin embargo, no es práctico que los objetos que representan dicha información se mantengan en procesos en ejecución durante períodos ilimitados, particularmente porque no están necesariamente en uso todo el tiempo. Para evitar el desperdicio potencial de recursos que resultaría de ejecutar todos los servidores que administran objetos remotos todo el tiempo, los servidores pueden iniciarse siempre que los clientes los necesiten, como se hace para el conjunto estándar de servicios TCP como FTP, que son iniciados bajo demanda por un servicio llamado*inetd*. Los procesos que inician procesos de servidor para alojar objetos remotos se denominan*activadores*, por las siguientes razones.

Un objeto remoto se describe como*activo*cuando está disponible para su invocación dentro de un proceso en ejecución, mientras que se llama*pasivo*si no está actualmente activo pero puede activarse. Un objeto pasivo consta de dos partes:

1. la implementación de sus métodos;
2. su estado en forma ordenada.

*Activación*consiste en crear un objeto activo a partir del objeto pasivo correspondiente creando una nueva instancia de su clase e inicializando sus variables de instancia desde el estado almacenado. Los objetos pasivos se pueden activar a pedido, por ejemplo, cuando necesitan ser invocados por otros objetos.

Un*activador*es responsable de:

- registrar objetos pasivos que están disponibles para activación, lo que implica registrar los nombres de los servidores contra las URL o los nombres de archivo de los objetos pasivos correspondientes;

- iniciar procesos de servidor con nombre y activar objetos remotos en ellos;
- realizar un seguimiento de las ubicaciones de los servidores para objetos remotos que ya ha activado.

Java RMI proporciona la capacidad de hacer que algunos objetos remotos *activable*[java.sun.com IX]. Cuando se invoca un objeto activable, si ese objeto no está actualmente activo, el objeto se activa desde su estado ordenado y luego pasa la invocación. Utiliza un activador en cada computadora servidor.

El estudio de caso de CORBA en el Capítulo 8 describe el repositorio de implementación: una forma débil de activador que inicia servicios que contienen objetos en un estado inicial.

Almacenes de objetos persistentes •Un objeto que está garantizado para vivir entre activaciones de procesos se llama *objeto persistente*. Los objetos persistentes generalmente son administrados por almacenes de objetos persistentes, que almacenan su estado en forma ordenada en el disco. Los ejemplos incluyen el servicio de estado persistente CORBA (consulte el Capítulo 8), Java Data Objects [java.sun.com VIII] y Java persistente [Jordan 1996;java.sun.com IV].

En general, un almacén de objetos persistentes administrará una gran cantidad de objetos persistentes, que se almacenan en el disco o en una base de datos hasta que se necesitan. Se activarán cuando sus métodos sean invocados por otros objetos. La activación generalmente está diseñada para ser transparente, es decir, el invocador no debería poder saber si un objeto ya está en la memoria principal o si debe activarse antes de que se invoque su método. Los objetos persistentes que ya no se necesitan en la memoria principal se pueden pasivar. En la mayoría de los casos, los objetos se guardan en el almacén de objetos persistentes cada vez que alcanzan un estado coherente, por el bien de la tolerancia a errores. El almacén de objetos persistentes necesita una estrategia para decidir cuándo pasivar los objetos. Por ejemplo, puede hacerlo en respuesta a una solicitud en el programa que activó los objetos, ya sea al final de una transacción o cuando el programa sale. Los almacenes de objetos persistentes generalmente intentan optimizar la pasivación guardando solo aquellos objetos que se han modificado desde la última vez que se guardaron.

Los almacenes de objetos persistentes generalmente permiten que las colecciones de objetos persistentes relacionados tengan nombres legibles por humanos, como nombres de ruta o URL. En la práctica, cada nombre legible por humanos está asociado con la raíz de un conjunto conectado de objetos persistentes.

Hay dos enfoques para decidir si un objeto es persistente o no:

- El almacén de objetos persistentes mantiene algunas raíces persistentes, y cualquier objeto al que se pueda acceder desde una raíz persistente se define como persistente. Este enfoque es utilizado por Persistent Java, Java Data Objects y PerDiS [Ferreira *et al.*2000]. Hacen uso de un recolector de basura para deshacerse de los objetos que ya no son accesibles desde las raíces persistentes.
- El almacén de objetos persistentes proporciona algunas clases en las que se basa la persistencia: los objetos persistentes pertenecen a sus subclases. Por ejemplo, en Arjuna [Parrington *et al.* 1995], los objetos persistentes se basan en clases de C++ que proporcionan transacciones y recuperación. Los objetos no deseados deben eliminarse explícitamente.

Algunas tiendas de objetos persistentes, como PerDiS y Khazana [Carter *et al.*1998], permite que los objetos se activen en múltiples cachés locales para los usuarios, en lugar de en los servidores. En este caso, se requiere un protocolo de consistencia de caché. Se pueden encontrar más detalles sobre los modelos de consistencia en el sitio web complementario, en el capítulo de la cuarta edición sobre memoria compartida distribuida [www.cdk5.net/dsm].

Ubicación del objeto •La Sección 4.3.4 describe una forma de referencia de objeto remoto que contiene la dirección de Internet y el número de puerto del proceso que creó el objeto remoto como una forma de garantizar la unicidad. Esta forma de referencia a objetos remotos también se puede usar como una dirección para un objeto remoto, siempre que ese objeto permanezca en el mismo proceso por el resto de su vida. Pero algunos objetos remotos existirán en una serie de procesos diferentes, posiblemente en diferentes computadoras, a lo largo de su vida. En este caso, una referencia de objeto remoto no puede actuar como una dirección. Los clientes que realizan invocaciones requieren tanto una referencia de objeto remoto como una dirección a la que enviar las invocaciones.

A*servicio de localización*ayuda a los clientes a localizar objetos remotos a partir de sus referencias de objetos remotos. Utiliza una base de datos que asigna referencias de objetos remotos a sus ubicaciones actuales probables; las ubicaciones son probables porque un objeto puede haber migrado nuevamente desde la última vez que se supo de él. Por ejemplo, el sistema Nubes [Dasgupta*et al.*1991] y el sistema Emerald [julio*et al.*1988] usó un esquema de caché/transmisión en el que un miembro de un servicio de ubicación en cada computadora tiene una pequeña caché de asignaciones de referencia a ubicación de objetos remotos. Si una referencia a un objeto remoto está en el caché, esa dirección se prueba para la invocación y fallará si el objeto se ha movido. Para ubicar un objeto que se ha movido o cuya ubicación no está en el caché, el sistema emite una solicitud. Este esquema se puede mejorar mediante el uso de punteros de ubicación hacia adelante, que contienen sugerencias sobre la nueva ubicación de un objeto. Otro ejemplo es el servicio de resolución requerido para convertir el URN de un recurso en su URL actual, mencionado en la Sección 9.1.

5.4.3 Recolección de basura distribuida

El objetivo de un recolector de basura distribuido es garantizar que si una referencia local o remota a un objeto todavía se mantiene en cualquier lugar de un conjunto de objetos distribuidos, el objeto en sí seguirá existiendo, pero tan pronto como ningún objeto ya no tenga una referencia. a él, se recogerá el objeto y se recuperará la memoria que utiliza.

Describimos el algoritmo de recolección de basura distribuida de Java, que es similar al descrito por Birrell*et al.*[1995]. Se basa en el conteo de referencia. Cada vez que una referencia de objeto remoto ingresa a un proceso, se creará un proxy y permanecerá allí durante el tiempo que sea necesario. El proceso donde vive el objeto (su servidor) debe ser informado del nuevo proxy en el cliente. Luego, cuando ya no haya un proxy en el cliente, se debe informar al servidor. El recolector de basura distribuido trabaja en cooperación con los recolectores de basura locales de la siguiente manera:

- Cada proceso de servidor mantiene un conjunto de nombres de procesos que contienen referencias a objetos remotos para cada uno de sus objetos remotos; Por ejemplo,*B.titulares*es el conjunto de procesos cliente (máquinas virtuales) que tienen proxies para objeto*B*. (En la figura 5.15, este conjunto incluirá el proceso de cliente ilustrado). Este conjunto se puede mantener en una columna adicional en la tabla de objetos remotos.
- cuando un cliente*C*primero recibe una referencia remota a un objeto remoto en particular,*B*, hace un*añadirRef(B)*invocación al servidor de ese objeto remoto y luego crea un proxy; el servidor agrega*C*a*B.titulares.*

- cuando un cliente *C* El recolector de basura se da cuenta de que un proxy para el objeto remoto *B* ya no es accesible, hace un *eliminarRef(B)* invocación al servidor correspondiente y luego borra el proxy; el servidor elimina *C* de *B.titulares.*
- Cuando *B.titulares* está vacío, el recolector de basura local del servidor reclamará el espacio ocupado por *B* a menos que haya titulares locales.

Este algoritmo está pensado para llevarse a cabo mediante una comunicación de petición-respuesta por pares con *como máximo una vez* semántica de invocación entre los módulos de referencia remota en los procesos: no requiere ninguna sincronización global. Tenga en cuenta también que las invocaciones adicionales realizadas en nombre del algoritmo de recolección de basura no afectan a todos los RMI normales; ocurren solo cuando se crean y eliminan proxies.

Existe la posibilidad de que un cliente pueda hacer una *eliminarRef(B)* invocación aproximadamente al mismo tiempo que otro cliente hace una *añadirRef(B)* invocación. Si el *eliminarRef* llega primero y *B.titulares* está vacío, el objeto remoto *B* podría eliminarse antes de que *añadirRef* llega Para evitar esta situación, si el conjunto *B.titulares* está vacío en el momento en que se transmite una referencia de objeto remoto, se agrega una entrada temporal hasta que el *añadirRef* llega

El algoritmo de recolección de basura distribuida de Java tolera fallas de comunicación utilizando el siguiente enfoque. El *añadirRef* y *eliminarRef* Las operaciones son idempotentes. En el caso de que un *añadirRef(B)* llamada devuelve una excepción (lo que significa que el método se ejecutó una vez o no se ejecutó en absoluto), el cliente no creará un proxy pero hará una *eliminarRef(B)* llamar. El efecto de *eliminarRef* es correcto sea o no el *añadirRef* logrado El caso donde *eliminarRef* falla es tratado por *arrendamientos*.

El algoritmo de recolección de basura distribuida de Java puede tolerar la falla de los procesos del cliente. Para lograr esto, los servidores arriendan sus objetos a los clientes por un período de tiempo limitado. El período de arrendamiento comienza cuando el cliente hace una *añadirRef* invocación al servidor. Termina cuando el tiempo ha expirado o cuando el cliente hace una *eliminarRef* invocación al servidor. La información almacenada por el servidor relativa a cada arrendamiento contiene el identificador de la máquina virtual del cliente y el período del arrendamiento. Los clientes son responsables de solicitar al servidor que renueve sus arrendamientos antes de que caduquen.

Arrendamientos en Jini •El sistema distribuido Jini incluye una especificación para arrendamientos [Arnold *et al.*1999] que se puede usar en una variedad de situaciones cuando un objeto ofrece un recurso a otro objeto, por ejemplo, cuando los objetos remotos ofrecen referencias a otros objetos. Los objetos que ofrecen dichos recursos corren el riesgo de tener que mantener los recursos cuando los usuarios ya no están interesados o sus programas se han cerrado. Para evitar protocolos complicados para descubrir si los usuarios de los recursos todavía están interesados, los recursos se ofrecen por un período de tiempo limitado. La concesión del uso de un recurso por un período de tiempo se denomina *alquiler*. El objeto que ofrece el recurso lo mantendrá hasta que expire el tiempo de la concesión. Los usuarios de los recursos son responsables de solicitar su renovación cuando caduquen.

El período de un arrendamiento se puede negociar entre el otorgante y el destinatario en Jini, aunque esto no sucede con los arrendamientos utilizados en Java RMI. En Jini, un objeto que representa un arrendamiento implementa el *Alquiler* interfaz. Contiene información sobre el período del arrendamiento y los métodos que permiten renovar o cancelar el arrendamiento. El otorgante devuelve una instancia de un *Alquiler* cuando proporciona un recurso a otro objeto.

5.5 Estudio de caso: Java RMI

Java RMI amplía el modelo de objetos de Java para proporcionar soporte para objetos distribuidos en el lenguaje Java. En particular, permite que los objetos invoquen métodos en objetos remotos utilizando la misma sintaxis que para las invocaciones locales. Además, la comprobación de tipos se aplica por igual a las invocaciones remotas que a las locales. Sin embargo, un objeto que realiza una invocación remota sabe que su objetivo es remoto porque debe manejar*RemoteException*s; y el implementador de un objeto remoto es consciente de que es remoto porque debe implementar el *Remoto*interfaz. Aunque el modelo de objetos distribuidos está integrado en Java de forma natural, la semántica del paso de parámetros difiere porque el invocador y el destino están alejados entre sí.

La programación de aplicaciones distribuidas en Java RMI debería ser relativamente simple porque es un sistema de un solo idioma: las interfaces remotas se definen en el lenguaje Java. Si se utiliza un sistema de múltiples idiomas como CORBA, el programador necesita aprender un IDL y comprender cómo se asigna al lenguaje de implementación. Sin embargo, incluso en un sistema de un solo idioma, el programador de un objeto remoto debe considerar su comportamiento en un entorno concurrente.

En el resto de esta introducción, damos un ejemplo de una interfaz remota, luego analizamos la semántica de paso de parámetros con referencia al ejemplo. Finalmente, discutimos la descarga de clases y el archivador. La segunda sección de este estudio de caso analiza cómo crear programas de cliente y servidor para la interfaz de ejemplo. La tercera sección se ocupa del diseño y la implementación de Java RMI. Para obtener detalles completos de Java RMI, consulte el tutorial sobre invocación remota[java.sun.com I].

En este caso de estudio, el caso de estudio de CORBA en el Capítulo 8 y la discusión de los servicios web en el Capítulo 9, usamos un*pizarra compartida*como ejemplo. Este es un programa distribuido que permite a un grupo de usuarios compartir una vista común de una superficie de dibujo que contiene objetos gráficos, como rectángulos, líneas y círculos, cada uno de los cuales ha sido dibujado por uno de los usuarios. El servidor mantiene el estado actual de un dibujo proporcionando una operación para que los clientes le informen sobre la última forma que ha dibujado uno de sus usuarios y manteniendo un registro de todas las formas que ha recibido. El servidor también proporciona operaciones que permiten a los clientes recuperar las últimas formas dibujadas por otros usuarios consultando el servidor. El servidor tiene un número de versión (un número entero) que incrementa cada vez que llega una nueva forma y se adjunta a la nueva forma. El servidor proporciona operaciones que permiten a los clientes consultar su número de versión y el número de versión de cada forma,

Interfaces remotas en Java RMI •Las interfaces remotas se definen extendiendo una interfaz llamada*Remoto*provisto en el*java.rmi*paquete. Los métodos deben arrojar *RemoteException*, pero también se pueden lanzar excepciones específicas de la aplicación. La Figura 5.16 muestra un ejemplo de dos interfaces remotas llamadas*Forma*y*lista de formas*. En este ejemplo, *Objeto gráfico* es una clase que contiene el estado de un objeto gráfico, por ejemplo, su tipo, su posición, el rectángulo que lo encierra, el color de línea y el color de relleno, y proporciona operaciones para acceder y actualizar su estado.*Objeto gráfico*debe implementar la *Serializable*interfaz. Considere la interfaz*Forma*Primero el*getVersion*método devuelve un número entero, mientras que el *getAllState*método devuelve una instancia de la clase *Objeto gráfico*. Ahora considere la interfaz *lista de formas*: es*nueva forma*método pasa una instancia de*Objeto gráfico*como su argumento pero devuelve un objeto con un control remoto

Figura 5.16 Interfaces remotas de Java*Forma*y *lista de formas*

```
importar java.rmi.*;
importar java.util.Vector;
La forma de la interfaz pública extiende el control remoto {
    int getVersion() lanza RemoteException; GraphicalObject
    getAllState() lanza RemoteException;                          1
}
La interfaz pública ShapeList extiende Remote {
    Shape newShape(GraphicalObject g) lanza RemoteException;      2
    Vector allShapes() lanza RemoteException;
    int getVersion() lanza RemoteException;
}
```

interfaz (es decir, un objeto remoto) como su resultado. Un punto importante a tener en cuenta es que tanto los objetos ordinarios como los objetos remotos pueden aparecer como argumentos y resultados en una interfaz remota. Estos últimos siempre se indican con el nombre de su interfaz remota. En la siguiente subsección, analizamos cómo los objetos ordinarios y los objetos remotos se pasan como argumentos y resultados.

Paso de parámetros y resultados •En Java RMI, se supone que los parámetros de un método son *aporte*parámetros y el resultado de un método es un único*producción*parámetro. La sección 4.3.2 describe la serialización de Java, que se utiliza para ordenar argumentos y resultados en Java RMI. Cualquier objeto que sea serializable, es decir, que implemente el*Serializable* interfaz: se puede pasar como argumento o resultado en Java RMI. Todos los tipos primitivos y objetos remotos son serializables. El sistema RMI descarga las clases para los argumentos y los valores de los resultados al destinatario cuando es necesario.

Pasar objetos remotos: cuando el tipo de un parámetro o valor de resultado se define como una interfaz remota, el argumento o resultado correspondiente siempre se pasa como una referencia de objeto remoto. Por ejemplo, en la figura 5.16, línea 2, el valor de retorno del método *nueva forma*Se define como*Forma*–una interfaz remota. Cuando se recibe una referencia de objeto remoto, se puede utilizar para realizar llamadas RMI en el objeto remoto al que se refiere.

Pasar objetos no remotos: Todos los objetos no remotos serializables se copian y pasan por valor. Por ejemplo, en la Figura 5.16 (líneas 2 y 1) el argumento de *nueva forma*y el valor de retorno de*getAllState*ambos son de tipo*Objeto gráfico*, que es serializable y se pasa por valor. Cuando se pasa un objeto por valor, se crea un nuevo objeto en el proceso del receptor. Los métodos de este nuevo objeto se pueden invocar localmente, lo que posiblemente provoque que su estado difiera del estado del objeto original en el proceso del remitente.

Así, en nuestro ejemplo, el cliente utiliza el método*nueva forma*para pasar una instancia de *Objeto gráfico*al servidor; el servidor crea un objeto remoto de tipo*Forma* que contiene el estado de la*Objeto gráfico*y devuelve una referencia a un objeto remoto. Los argumentos y los valores devueltos en una invocación remota se serializan en un flujo utilizando el método descrito en la Sección 4.3.2, con las siguientes modificaciones:

Figura 5.17 El*Denominación*clase de registro RMI de Java

void rebind (nombre de cadena, objeto remoto)

Este método es utilizado por un servidor para registrar el identificador de un objeto remoto por nombre, como se muestra en la Figura 5.18, línea 3.

enlace vacío (nombre de cadena, objeto remoto)

Este método también puede ser utilizado por un servidor para registrar un objeto remoto por nombre, pero si el nombre ya está vinculado a una referencia de objeto remoto, se genera una excepción.

void unbind (nombre de cadena, objeto remoto)

Este método elimina un enlace.

Búsqueda remota (nombre de cadena)

Los clientes utilizan este método para buscar un objeto remoto por nombre, como se muestra en la figura 5.20, línea 1. Se devuelve una referencia de objeto remoto.

Cadena [] lista()

Este método devuelve una matriz de*Instrumentos de cuerda*que contiene los nombres atados en el registro.

1. Siempre que un objeto que implementa el*Remoto*la interfaz se serializa, se reemplaza por su referencia de objeto remoto, que contiene el nombre de su clase (del objeto remoto).
2. Cuando cualquier objeto se serializa, su información de clase se anota con la ubicación de la clase (como una URL), lo que permite que el receptor descargue la clase.

Descarga de clases •Java está diseñado para permitir que las clases se descarguen de una máquina virtual a otra. Esto es particularmente relevante para objetos distribuidos que se comunican por medio de invocaciones remotas. Hemos visto que los objetos no remotos se pasan por valor y los objetos remotos se pasan por referencia como argumentos y resultados de RMI. Si el destinatario aún no posee la clase de un objeto pasado por valor, su código se descarga automáticamente. De manera similar, si el destinatario de una referencia de objeto remoto aún no posee la clase para un proxy, su código se descarga automáticamente. Esto tiene dos ventajas:

1. No es necesario que todos los usuarios mantengan el mismo conjunto de clases en su entorno de trabajo.
2. Tanto los programas de cliente como los de servidor pueden hacer un uso transparente de instancias de nuevas clases cada vez que se agregan.

Como ejemplo, considere el programa de pizarra y suponga que su implementación inicial de*Objeto gráfico*no permite texto. Un cliente con un objeto textual puede implementar una subclase de*Objeto gráfico*que trata con texto y pasa una instancia al servidor como argumento del*nueva forma*método. Después de eso, otros clientes pueden recuperar la instancia usando el*getAllState*método. El código de la nueva clase se descargará automáticamente desde el primer cliente al servidor y luego a otros clientes según sea necesario.

Figura 5.18 clase Java*ShapeListServer*con*principal*método

```
importar java.rmi.*;
importar java.rmi.server.UnicastRemoteObject;
clase pública ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(nuevo RMISecurityManager());
        intentar{
            ShapeList aShapeList = new ShapeListServant();                    1
            Trozo de ShapeList =                                              2
                (ShapeList) UnicastRemoteObject.exportObject(aShapeList,0);3
            Naming.rebind("//bruno.ShapeList", stub );                        4
            System.out.println("Servidor ShapeList listo"); }
        atrapar(Excepción e) {
            System.out.println("ShapeList servidor principal" + e.getMessage());}
    }
}
```

Registro RMI •El registro RMI es el archivador de Java RMI. Una instancia de RMIregistry normalmente debería ejecutarse en cada computadora servidor que aloja objetos remotos. Mantiene una tabla que asigna nombres textuales de estilo URL a referencias a objetos remotos alojados en esa computadora. Se accede por métodos de la*Denominación*clase, cuyos métodos toman como argumento una cadena con formato de URL de la forma:

```
//nombreDeEquipo:puerto/nombreDeObjeto
```

dónde*nombre de la computadora*y*puerto*consulte la ubicación del registro RMI. Si se omiten, se asume la computadora local y el puerto predeterminado. Su interfaz ofrece los métodos que se muestran en la Figura 5.17, en los que no se enumeran las excepciones: todos los métodos pueden generar un*RemoteException*.

Utilizado de esta manera, los clientes deben dirigir sus*buscar*Consultas a anfitriones particulares. Alternativamente, es posible configurar un servicio de vinculación en todo el sistema. Para lograr esto, es necesario ejecutar una instancia de RMIregistry en el entorno de red y luego usar la clase*LocalizarRegistro*, Qué esta en*java.rmi.registry*, para descubrir este registro. Más específicamente, esta clase contiene un*obtenerRegistro*método que devuelve un objeto de tipo *Registro*que representa el servicio de enlace remoto:

```
Registro estático público getRegistry() lanza RemoteException
```

Después de esto, es necesario emitir una llamada de*reencuadernar*en este regreso*Registro* objeto para establecer una conexión con el registro RMI remoto.

5.5.1 Creación de programas cliente y servidor

Esta sección describe los pasos necesarios para producir programas cliente y servidor que utilicen el *Remoto*interfaces*Formay lista de formas*se muestra en la Figura 5.16. El programa del servidor es una versión simplificada de un servidor de pizarra que implementa las dos interfaces*Formay lista de formas*. Describimos un programa de cliente de sondeo simple y luego presentamos la devolución de llamada

Figura 5.19 clase Java*ShapeListServant*interfaz de implementos*lista de formas*

```
importar java.util.Vector;
la clase pública ShapeListServant implementa ShapeList {
    vector privado theList;              // contiene la lista de Formas
    versión int privada;
    ShapeListServant público(){...}
    Forma pública nuevaForma(ObjetoGráfico g) {                    1
        versión++;
        Forma s = nuevo ShapeServant( g, versión);                 2
        laLista.addElement(s);
        devoluciones;
    }
    public Vector allShapes(){...}
    public int getVersion() { ... }
}
```

técnica que se puede utilizar para evitar la necesidad de sondear el servidor. Las versiones completas de las clases ilustradas en esta sección están disponibles enwww.cdk5.net/rmi .

Programa de servidor •El servidor es un servidor de pizarra: representa cada forma como un objeto remoto instanciado por un sirviente que implementa el*Forma*interfaz y contiene el estado de un objeto gráfico, así como su número de versión; representa su colección de formas usando otro sirviente que implementa el*lista de formas*interfaz y contiene una colección de formas en un *Vector.*

El programa del servidor consta de un*principal*método y una clase sirviente para implementar cada una de sus interfaces remotas. El*principal*El método de la clase de servidor se muestra en la Figura 5.18, con los pasos clave contenidos en las líneas marcadas del 1 al 4:

- En la línea 1, el servidor crea una instancia de*ShapeListServant.*
- Las líneas 2 y 3 usan el método*exportarObjeto*(definido en*UnicastRemoteObjectUnicastRemoteObject*) para hacer que este objeto esté disponible para el tiempo de ejecución de RMI, de modo que esté disponible para recibir invocaciones entrantes. El segundo parámetro de*exportarObjeto*especifica el puerto TCP que se utilizará para las invocaciones entrantes. Es una práctica normal establecerlo en cero, lo que implica que se usará un puerto anónimo (uno generado por el tiempo de ejecución de RMI). Usando*UnicastRemoteObjectUnicastRemoteObject*asegura que el objeto resultante viva solo mientras el proceso en el que se crea (una alternativa es hacer de esto un *Activable*es decir, uno que vive más allá de la instancia del servidor).
- Finalmente, la línea 4 vincula el objeto remoto a un nombre en el registro RMI. Tenga en cuenta que el valor vinculado al nombre es una referencia de objeto remoto y su tipo es el tipo de su interfaz remota:*lista de formas*.

Las dos clases de sirvientes son*ShapeListServant*, que implementa la*lista de formas* interfaz, y*ShapeServant*, que implementa la*Forma*interfaz. La Figura 5.19 da un esquema de la clase*ShapeListServant*.

Figura 5.20 cliente Java de*lista de formas*

```
importar java.rmi.*;
importar java.rmi.servidor.*;
importar java.util.Vector;

clase pública ShapeListClient{
    public static void main(String args[]){
        System.setSecurityManager(nuevo RMISecurityManager());
        ShapeList aShapeList = null;

        intentar{
            aShapeList = (ShapeList) Naming.lookup("//bruno.ShapeList");      1
            Vector sList = aShapeList.allShapes();                            2
        } catch(RemoteException e) {System.out.println(e.getMessage()); }
        catch(Excepción e) {System.out.println("Cliente: " + e.getMessage());}
    }
}
```

La implementación de los métodos de la interfaz remota en una clase de servidor es completamente sencilla porque se puede hacer sin preocuparse por los detalles de la comunicación. Considere el método*nueva forma*en la Figura 5.19 (línea 1), que podría denominarse método de fábrica porque permite al cliente solicitar la creación de un servidor. Utiliza el constructor de*ShapeServant*, que crea un nuevo sirviente que contiene el *Objeto gráfico*y el número de versión pasados como argumentos. El tipo del valor de retorno de*nueva forma*es*Forma*–la interfaz implementada por el nuevo servidor. Antes de regresar, el método*nueva forma*agrega la nueva forma a su vector que contiene la lista de formas (línea 2).

El*principal*El método de un servidor necesita crear un administrador de seguridad para permitir que la seguridad de Java aplique la protección adecuada para un servidor RMI. Un administrador de seguridad predeterminado llamado*RMISecurityManager*está provisto. Protege los recursos locales para garantizar que las clases que se cargan desde sitios remotos no puedan tener ningún efecto en recursos como archivos, pero se diferencia del administrador de seguridad estándar de Java en que permite que el programa proporcione su propio cargador de clases y utilice la reflexión. Si un servidor RMI no establece un administrador de seguridad, los proxies y las clases solo se pueden cargar desde el classpath local, para proteger el programa del código que se descarga como resultado de invocaciones de métodos remotos.

Programa de clientes •Un cliente simplificado para el*lista de formas*servidor se ilustra en la Figura 5.20. Cualquier programa cliente necesita comenzar usando un archivador para buscar una referencia de objeto remoto. Nuestro cliente configura un administrador de seguridad y luego busca una referencia de objeto remoto para el objeto remoto usando el*buscar*funcionamiento del registro RMI (línea 1). Habiendo obtenido una referencia de objeto remoto inicial, el cliente continúa enviando RMIs a ese objeto remoto oa otros descubiertos durante su ejecución según las necesidades de su aplicación. En nuestro ejemplo, el cliente invoca el método*todas las formas*en el objeto remoto (línea 2) y recibe un vector de referencias de objetos remotos a todas las formas actualmente almacenadas en el servidor. Si el cliente estuviera implementando una pantalla de pizarra, usaría el servidor*getAllState*método en el*Forma*interfaz para recuperar cada uno de los objetos gráficos en el vector y mostrarlos en una ventana. Cada vez que el usuario termina

dibujar un objeto gráfico, invocará el método*nueva forma*en el servidor, pasando el nuevo objeto gráfico como su argumento. El cliente mantendrá un registro del último número de versión en el servidor y, de vez en cuando, invocará*getVersion*en el servidor para averiguar si otros usuarios han agregado nuevas formas. Si es así, los recuperará y los mostrará.

Devoluciones de llamadas •La idea general detrás de las devoluciones de llamada es que, en lugar de que los clientes consulten al servidor para averiguar si ha ocurrido algún evento, el servidor debe informar a sus clientes cada vez que ocurra ese evento. El término*llamar de vuelta*se utiliza para referirse a la acción de un servidor de notificar a los clientes sobre un evento. Las devoluciones de llamada se pueden implementar en RMI de la siguiente manera:

- El cliente crea un objeto remoto que implementa una interfaz que contiene un método para que el servidor llame. Nos referimos a esto como un*objeto de devolución de llamada*.
- El servidor proporciona una operación que permite a los clientes interesados informarle de las referencias de objetos remotos de sus objetos de devolución de llamada. Los registra en una lista.
- Cada vez que ocurre un evento de interés, el servidor llama a los clientes interesados. Por ejemplo, el servidor de pizarra llamaría a sus clientes siempre que se agregue un objeto gráfico.

El uso de devoluciones de llamada evita la necesidad de que un cliente sondee los objetos de interés en el servidor y sus desventajas concomitantes:

- El rendimiento del servidor puede verse degradado por el sondeo constante.
- Los clientes no pueden notificar a los usuarios las actualizaciones de manera oportuna.

Sin embargo, las devoluciones de llamada tienen sus propios problemas. En primer lugar, el servidor debe tener listas actualizadas de los objetos de devolución de llamada de los clientes, pero es posible que los clientes no siempre informen al servidor antes de salir, lo que deja al servidor con listas incorrectas. El*arrendamiento*técnica discutida en la Sección 5.4.3 se puede utilizar para superar este problema. El segundo problema asociado con las devoluciones de llamada es que el servidor necesita realizar una serie de RMI síncronos para los objetos de devolución de llamada de la lista. Consulte el Capítulo 6 para obtener algunas ideas sobre cómo resolver el segundo problema.

Ilustramos el uso de devoluciones de llamada en el contexto de la aplicación de pizarra. El *Devolución de llamada de pizarra*La interfaz se puede definir de la siguiente manera:

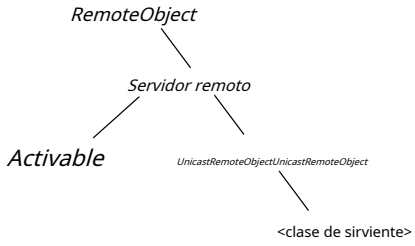
```
interfaz pública WhiteboardCallback implementa Remote {
    devolución de llamada nula (versión int) lanza RemoteException;
};
```

El cliente implementa esta interfaz como un objeto remoto, lo que permite que el servidor envíe al cliente un número de versión cada vez que se agrega un nuevo objeto. Pero antes de que el servidor pueda hacer esto, el cliente debe informar al servidor sobre su objeto de devolución de llamada. Para que esto sea posible, el*lista de formas*interfaz requiere métodos adicionales como*registro*y *darse de baja*, definido de la siguiente manera:

```
registro int (devolución de llamada de WhiteboardCallback) lanza
RemoteException; void deregister(int callbackId) lanza RemoteException;
```

Después de que el cliente haya obtenido una referencia al objeto remoto con el*lista de formas*interfaz (por ejemplo, en la Figura 5.20, línea 1) y creó una instancia de su objeto de devolución de llamada, utiliza el*registro*método de*lista de formas*para informar al servidor que está interesado en recibir

Figura 5.21 Clases que admiten Java RMI



devoluciones de llamada El*registro*método devuelve un número entero (el*ID de devolución de llamada*) referente al registro. Cuando el cliente haya terminado debe llamar*darse de baja*para informar al servidor que ya no requiere devoluciones de llamada. El servidor es responsable de mantener una lista de clientes interesados y notificar a todos ellos cada vez que aumenta su número de versión.

5.5.2 Diseño e implementación de Java RMI

El sistema Java RMI original usaba todos los componentes que se muestran en la Figura 5.15. Pero en Java 1.2, las funciones de reflexión se utilizaron para crear un despachador genérico y evitar la necesidad de esqueletos. Antes de J2SE 5.0, los proxies de los clientes los generaba un compilador llamado*rmico*de las clases de servidor compiladas (no de las definiciones de las interfaces remotas). Sin embargo, este paso ya no es necesario con las versiones recientes de J2SE, que contienen soporte para la generación dinámica de clases de código auxiliar en tiempo de ejecución.

Uso de la reflexión •La reflexión se utiliza para pasar información en mensajes de solicitud sobre el método que se va a invocar. Esto se logra con la ayuda de la clase.*Método*en el paquete de reflexión. Cada instancia de*Método*epresenta las características de un método particular, incluyendo su clase y los tipos de sus argumentos, valor de retorno y excepciones. La característica más interesante de esta clase es que una instancia de*Método*puede ser invocado en un objeto de una clase adecuada por medio de su*invocar*método. El*invocar*El método requiere dos argumentos: el primero especifica el objeto para recibir la invocación y el segundo es una matriz de*Objeto*que contiene los argumentos. El resultado se devuelve como tipo*Objeto*.

Para volver al uso de la*Método*clase en RMI: el proxy tiene que reunir información sobre un método y sus argumentos en el*pedido*mensaje. Para el método ordena un objeto de clase*Método*. Pone los argumentos en una matriz de*Objeto*s y luego ordena esa matriz. El despachador desarma el*Método*objeto y sus argumentos en la matriz de*Objeto*s de la*pedido* mensaje. Como de costumbre, la referencia de objeto remoto del objetivo se habrá desclasificado y la referencia de objeto local correspondiente se habrá obtenido del módulo de referencia remota. Luego, el despachador llama al *Método*objetos*invocar*método, proporcionando el destino y la matriz de valores de argumento. Cuando se ha ejecutado el método, el despachador ordena el resultado o cualquier excepción en el*responder*mensaje. Por lo tanto, el despachador es genérico, es decir, el mismo despachador se puede usar para todas las clases de objetos remotos y no se requieren esqueletos.

Clases de Java compatibles con RMI •La Figura 5.21 muestra la estructura de herencia de las clases que soportan servidores Java RMI. La única clase que el programador debe tener en cuenta es *UnicastRemoteObjectUnicastRemoteObject*, que toda clase de sirviente simple necesita extender. La clase *UnicastRemoteObjectUnicastRemoteObject*extiende una clase abstracta llamada*Servidor remoto*, que proporciona

versiones abstractas de los métodos requeridos por los servidores remotos. *UnicastRemoteObjectUnicastRemoteObject*fue el primer ejemplo de*Servidor remoto*ser provisto. otro llamado*Activable*está disponible para proporcionar objetos activables. Otras alternativas podrían proporcionar objetos replicados. La clase*Servidor remoto*es una subclase de*RemoteObject* que tiene una variable de instancia que contiene la referencia del objeto remoto y proporciona los siguientes métodos:

*es igual*Este método compara referencias de objetos remotos.

: Este método da el contenido de la referencia del objeto remoto como un*Cadena*.

leerObjeto, escribirObjeto: Estos métodos deserializan/serializan objetos remotos.

además, el*en vez de*El operador se puede utilizar para probar objetos remotos.

5.6 Resumen

Este capítulo ha discutido tres paradigmas para la programación distribuida: protocolos de solicitud-respuesta, llamadas a procedimientos remotos e invocación de métodos remotos. Todos estos paradigmas proporcionan mecanismos para que las entidades independientes distribuidas (procesos, objetos, componentes o servicios) se comuniquen directamente entre sí.

Los protocolos de solicitud y respuesta brindan un soporte ligero y mínimo para la computación cliente-servidor. Dichos protocolos se utilizan a menudo en entornos donde se deben minimizar los gastos generales de comunicación, por ejemplo, en sistemas integrados. Su función más común es admitir RPC o RMI, como se explica a continuación.

El enfoque de llamada a procedimiento remoto fue un avance significativo en los sistemas distribuidos, proporcionando soporte de alto nivel para los programadores al extender el concepto de llamada a procedimiento para operar en un entorno de red. Esto proporciona importantes niveles de transparencia en los sistemas distribuidos. Sin embargo, debido a sus diferentes fallas y características de rendimiento ya la posibilidad de acceso simultáneo a los servidores, no es necesariamente una buena idea hacer que las llamadas a procedimientos remotos parezcan exactamente iguales a las llamadas locales. Las llamadas a procedimientos remotos proporcionan una variedad de semánticas de invocación, desde *tal vez*invocaciones hasta*como máximo una vez*semántica.

El modelo de objetos distribuidos es una extensión del modelo de objetos locales que se utiliza en los lenguajes de programación basados en objetos. Los objetos encapsulados forman componentes útiles en un sistema distribuido, ya que la encapsulación los hace completamente responsables de administrar su propio estado, y la invocación local de métodos puede extenderse a la invocación remota. Cada objeto en un sistema distribuido tiene una referencia de objeto remoto (un identificador único global) y una interfaz remota que especifica cuáles de sus operaciones se pueden invocar de forma remota.

Las implementaciones de middleware de RMI proporcionan componentes (incluidos proxies, esqueletos y despachadores) que ocultan los detalles de clasificación, paso de mensajes y ubicación de objetos remotos de los programadores de clientes y servidores. Estos componentes pueden ser generados por un compilador de interfaz. Java RMI extiende la invocación local a la invocación remota usando la misma sintaxis, pero las interfaces remotas deben especificarse extendiendo una interfaz llamada*Remoto*y hacer que cada método arroje un*RemoteException*. Esto garantiza que los programadores sepan cuándo realizan invocaciones remotas o implementan objetos remotos, lo que les permite manejar errores o diseñar objetos adecuados para el acceso simultáneo.

EJERCICIOS

5.1 Defina una clase cuyas instancias representen mensajes de solicitud y respuesta, como se ilustra en la figura 5.4. La clase debe proporcionar un par de constructores, uno para los mensajes de solicitud y otro para los mensajes de respuesta, que muestren cómo se asigna el identificador de solicitud. También debe proporcionar un método para clasificarse a sí mismo en una matriz de bytes y desagrupar una matriz de bytes en una instancia. *página 188*

5.2 Programe cada una de las tres operaciones del protocolo de solicitud-respuesta en la Figura 5.3, usando comunicación UDP, pero sin agregar ninguna medida de tolerancia a fallas. Debe usar las clases que definió en el capítulo anterior para referencias a objetos remotos (Ejercicio 4.13) y superiores para mensajes de solicitud y respuesta (Ejercicio 5.1). *página 187*

5.3 Dé un resumen de la implementación del servidor, mostrando cómo las operaciones *obtenerSolicitud* y *enviar respuesta* son utilizados por un servidor que crea un nuevo hilo para ejecutar cada solicitud del cliente. Indique cómo el servidor copiará el *ID de solicitud* del mensaje de solicitud al mensaje de respuesta y cómo obtendrá la dirección IP y el puerto del cliente. *página 187*

5.4 Definir una nueva versión del *doOperation* método que establece un tiempo de espera para el mensaje de respuesta. Después de un tiempo de espera, retransmite el mensaje de solicitud *norte* veces. Si aún no hay respuesta, informa a la persona que llama. *página 188*

5.5 Describa un escenario en el que un cliente podría recibir una respuesta de una llamada anterior. *página 187*

5.6 Describir las formas en que el protocolo de solicitud-respuesta enmascara la heterogeneidad de los sistemas operativos y de las redes informáticas. *página 187*

5.7 Comenta si las siguientes operaciones son *idempotente*:

i) presionar un botón de solicitud de ascensor (ascensor);

ii) escribir datos en un archivo;

iii) agregar datos a un archivo.

¿Es condición necesaria para la idempotencia que la operación no esté asociada a ningún estado? *página 190*

5.8 Explique las opciones de diseño que son relevantes para minimizar la cantidad de datos de respuesta almacenados en un servidor. Compare los requisitos de almacenamiento cuando se utilizan los protocolos RR y RRA. *página 191*

5.9 Suponga que el protocolo RRA está en uso. ¿Cuánto tiempo deben conservar los servidores los datos de respuesta no reconocidos? ¿Deberían los servidores enviar repetidamente la respuesta en un intento de recibir un acuse de recibo? *página 191*

5.10 ¿Por qué la cantidad de mensajes intercambiados en un protocolo podría ser más importante para el rendimiento que la cantidad total de datos enviados? Diseñe una variante del protocolo RRA en la que el acuse de recibo se superponga (es decir, se transmita en el mismo mensaje que) a la siguiente solicitud cuando corresponda y, de lo contrario, se envíe como un mensaje separado. (Sugerencia: use un temporizador adicional en el cliente). *página 191*

5.11 Un *Elección* La interfaz proporciona dos métodos remotos:

votar: Este método tiene dos parámetros a través de los cuales el cliente proporciona el nombre de un candidato (una cadena) y el 'número de votante' (un número entero que se utiliza para garantizar que cada usuario vote una sola vez). Los números de los votantes se asignan escasamente del rango de números enteros para que sean difíciles de adivinar.

resultado: Este método tiene dos parámetros a través de los cuales el servidor proporciona al cliente el nombre de un candidato y el número de votos para ese candidato.

¿Cuáles de los parámetros de estos dos procedimientos son *aporte* y cuales son *producción* parámetros? *página 195*

5.12 Discuta la semántica de invocación que se puede lograr cuando el protocolo de solicitud-respuesta se implementa sobre una conexión TCP/IP, lo que garantiza que los datos se entregan en el orden en que se envían, sin pérdida ni duplicación. Tenga en cuenta todas las condiciones que provocan la interrupción de una conexión. *Sección 4.2.4 y página 198*

5.13 Defina la interfaz para el *Elección* servicio en CORBA IDL y Java RMI. Tenga en cuenta que CORBA IDL proporciona el tipo *largo* para enteros de 32 bits. Compare los métodos en los dos idiomas para especificar *aporte* y *producción* argumentos *Figura 5.8, Figura 5.16*

5.14 El *Elección* El servicio debe asegurarse de que se registre un voto cada vez que un usuario crea que ha emitido un voto.

Discutir el efecto de *tal vez* semántica de llamadas en el *Elección* servicio.

haría *al menos una vez* la semántica de llamadas sea aceptable para el *Elección* servicio o recomendaría *como máximo una vez* llamar semántica? *página 199*

5.15 Se implementa un protocolo de solicitud-respuesta sobre un servicio de comunicación con fallas de omisión para proporcionar *al menos una vez* semántica de invocación. En el primer caso, el implementador asume un sistema distribuido asíncrono. En el segundo caso el implementador asume que el tiempo máximo para la comunicación y la ejecución de un método remoto es *T*. ¿De qué manera la última suposición simplifica la implementación? *página 198*

5.16 Esbozar una implementación para el *Elección* servicio que asegura que sus registros permanezcan consistentes cuando varios clientes acceden a él simultáneamente. *página 199*

5.17 asumir el *Elección* El servicio se implementa en RMI y debe garantizar que todos los votos se almacenen de forma segura incluso cuando el proceso del servidor falla. Explique cómo se puede lograr esto con referencia al esquema de implementación en su respuesta al Ejercicio 5.16. *páginas 213–214*

5.18 Muestre cómo usar la reflexión de Java para construir la clase de proxy de cliente para el *Elección* interfaz. Proporcione los detalles de la implementación de uno de los métodos en esta clase, que debe llamar al método *doOperation* con la siguiente firma:

byte[] doOperation (RemoteObjectRef o, Method m, byte[] argumentos);

Sugerencia: una variable de instancia de la clase proxy debe contener una referencia a un objeto remoto (vea el Ejercicio 4.13). *Figura 5.3, página 224*

5.19Muestre cómo generar una clase de proxy de cliente utilizando un lenguaje como C++ que no admita la reflexión, por ejemplo, a partir de la definición de la interfaz CORBA proporcionada en su respuesta al Ejercicio 5.13. Proporcione los detalles de la implementación de uno de los métodos en esta clase, que debe llamar al método *doOperation* definido en la Figura 5.3.

página 211

5.20Explicar cómo usar la reflexión de Java para construir un despachador genérico. Proporcione código Java para un despachador cuya firma sea:

envío de vacío público (Objeto de destino, Método aMethod, byte [] args)

Los argumentos proporcionan el objeto de destino, el método que se invocará y los argumentos para ese método en una matriz de bytes. *página 224*

5.21El ejercicio 5.18 requería que el cliente convirtiera *Objeto* argumentos en una matriz de bytes antes de invocar *doOperation* y el Ejercicio 5.20 requería que el despachador convirtiera una matriz de bytes en una matriz de *Objeto*s antes de invocar el método. Discutir la implementación de una nueva versión de *doOperation* con la siguiente firma:

Object[] doOperation (RemoteObjectRef o, Method m, Object[] argumentos);

que utiliza el *ObjectOutputStream* *ObjectOutputStream* y *ObjetoEntradaStream* clases para transmitir los mensajes de solicitud y respuesta entre el cliente y el servidor a través de una conexión TCP. ¿Cómo afectarían estos cambios al diseño del despachador? *Sección 4.3.2 y página 224*

5.22Un cliente realiza invocaciones de métodos remotos a un servidor. El cliente tarda 5 milisegundos en calcular los argumentos de cada solicitud y el servidor tarda 10 milisegundos en procesar cada solicitud. El tiempo de procesamiento del sistema operativo local para cada operación de envío o recepción es de 0,5 milisegundos, y el tiempo de red para transmitir cada mensaje de solicitud o respuesta es de 3 milisegundos. Marshalling o unmarshaling tarda 0,5 milisegundos por mensaje.

Calcular el tiempo que tarda el cliente en generar y devolver dos solicitudes: (i) si es de un solo hilo;

(ii) si tiene dos subprocesos que pueden realizar solicitudes simultáneamente en un solo procesador.

Puede ignorar los tiempos de cambio de contexto. ¿Existe la necesidad de una invocación asíncrona si los procesos del cliente y del servidor están enhebrados? *página 213*

5.23Diseñe una tabla de objetos remotos que pueda admitir la recolección de basura distribuida, así como la traducción entre referencias de objetos locales y remotos. Dé un ejemplo que involucre varios objetos remotos y proxies en varios sitios para ilustrar el uso de la tabla. Muestra los cambios en la tabla cuando una invocación hace que se cree un nuevo proxy. Luego muestre los cambios en la tabla cuando uno de los proxies se vuelve inalcanzable. *página 215*

5.24Una versión más simple del algoritmo de recolección de basura distribuida descrito en la Sección 5.4.3 solo invoca *añadirRef* en el sitio donde vive un objeto remoto cada vez que se crea un proxy y *eliminarRef* cada vez que se elimina un proxy. Resuma todos los posibles efectos de las fallas de comunicación y proceso en el algoritmo. Sugiera cómo superar cada uno de estos efectos, pero sin usar arrendamientos. *página 215*