Sistemas Distribuidos

Introducción al Modelo Cliente/Servidor

Tabla de Contenidos

1. Introducción 1
2. Distintos Niveles de Abstracción 3
3. C/S con RPC o RMI 5
    3.1. Con RPC 5
    3.2. Con RMI 6
4. Conclusiones 7
5. Bibliografía 7

1. Introducción

El modelo de procesamiento c/s es de las primeros establecidos en entornos distribuidos. Entre las razones más importantes se pueden citar:

1. Todo el desarrollo de los sistemas operativos. En este contexto está claro desde hace mucho tiempo que toda la interfase con las aplicaciones (o API: Application Programming/Programmer Interface), se establece en términos de system calls con un funcionamiento (semántica) claramente conocida en términos de requerimiento/respuesta.
2. Todo el desarrollo de las redes y sus arquitecturas en capas. Más allá del crecimiento de rendimiento (sin el cual los sistemas distribuidos no tendrían el uso actual), se han establecido muy claramente las capas o niveles de interconexión de procesos. Más allá de las definiciones de cada capa (sobre las cuales se puede discutir o no), se han establecido los protocolos asociados a cada capa que tiene, como las API de los sistemas operativos con una sintaxis y semántica bien definidas.

Se podría decir (al menos en teoría) que desde la perspectiva de un proceso que necesita un servicio bien definido (en términos de sintaxis y semántica), el funcionamiento es independiente de quién proporciona ese servicio. Más específicamente, si un proceso necesita imprimir un documento da igual si es el sistema operativo local u otra aplicación esa petición y la envía a la impresora que corresponde. Básicamente, lo que se espera es la impresión de un documento, no qué proceso o aplicación será la encargada de la interacción con la impresora correspondiente. La Fig. 1 muestra esquemáticamente esta perspectiva de un proceso que realiza un requerimiento, donde normalmente se recibe una respuesta. El tiempo que transcurre entre el envío del requerimiento y la recepción de la respuesta (entre tep y trr) normalmente es sólo de espera inactiva.

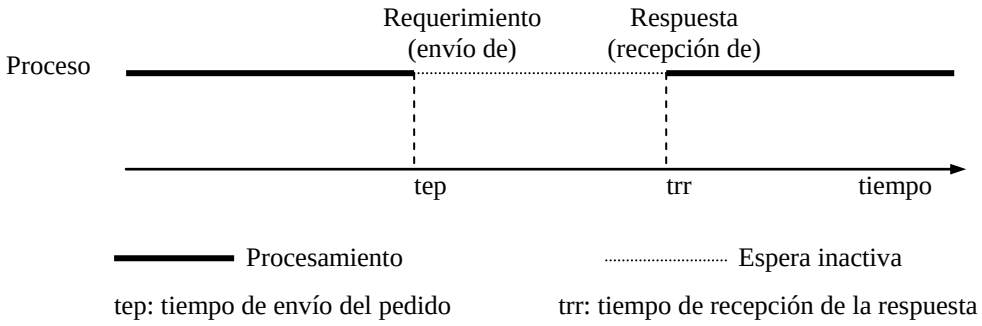

Figura 1: Proceso que Pide un Servicio.

El tiempo que transcurre desde que se pide un servicio (se realiza el requerimiento) hasta que se recibe la respuesta depende de diversos factores: la propia aplicación (su *prioridad*, por ejemplo), el requerimiento en particular, los datos involucrados (y su transferencia entre los procesos), el proceso/aplicación que proporciona o implementa el servicio. Los procesos o aplicaciones que proporcionan un servicio tienen su propio esquema de procesamiento, que se muestra en la Fig. 2.

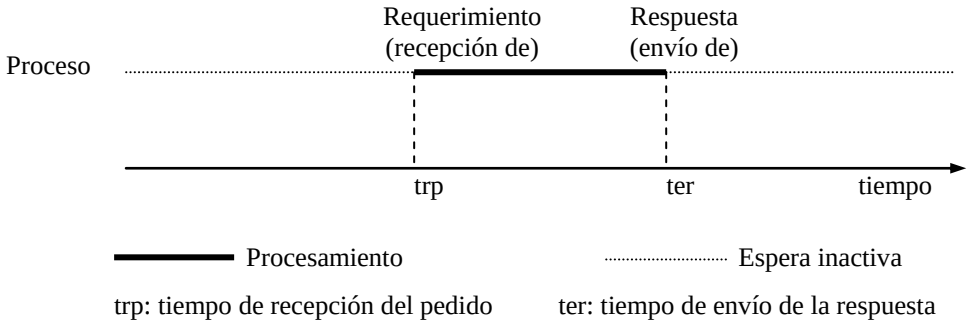

Figura 2: Proceso que Proporciona un Servicio.

Como se puede ver de la comparación de la Fig. 1 con la Fig. 2, los patrones de procesamiento son diferentes y bien definidos, específicamente respecto de los tiempos de espera.

En el contexto de las aplicaciones distribuidas c/s, a los procesos que piden algún servicio se los denomina genéricamente **clientes**. Por el otro lado, a los procesos que proporcionan los servicios se los denomina **servidores**. Los clientes son asimilables fácilmente a los procesos de usuario que requieren acceso al hardware de la computadora y análogamente los servidores son también fácilmente asimilables a los sistemas operativos que proporcionan acceso controlado y simplificado al hardware. La Fig. 3 muestra esquemáticamente la interacción en el tiempo de los procesos cliente y servidor, que no es un poco más que la *composición* de la Fig. 1 con la Fig. 2.

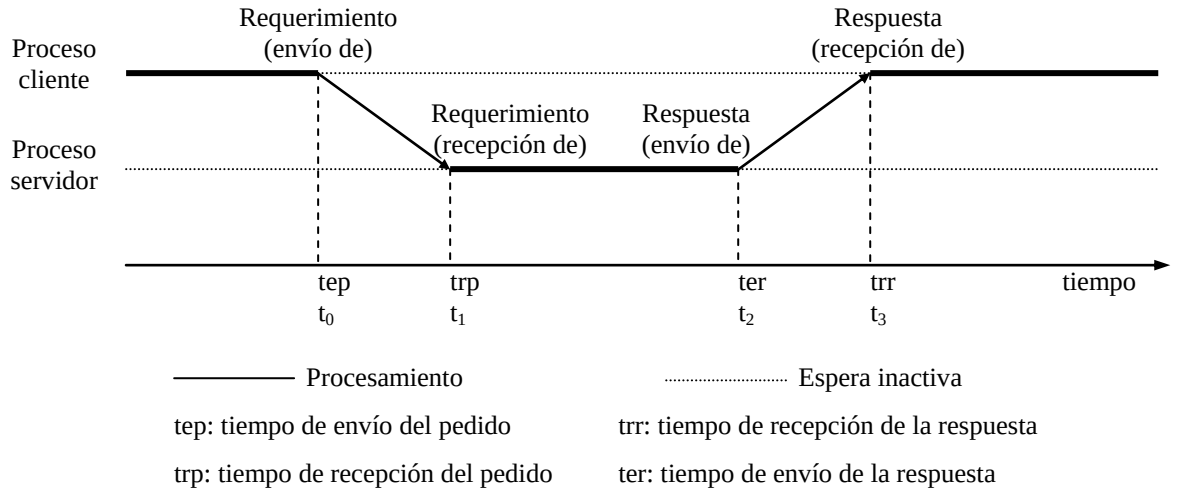

Figura 3: Procesamiento Cliente/Servidor de un Requerimiento.

También a partir de la Fig. 3 queda explícita la forma de comunicación y sincronización en el modelo c/s. Los clientes envían un requerimiento (con los datos necesarios propios del servicio) y espera por la respuesta del servidor (que puede incluir datos también). El servidor espera un requerimiento de un cliente (con los datos necesarios) procesa el requerimiento y envía la respuesta (con los datos determinados) al cliente.

Por otro lado, en la Fig. 3 se pueden discriminar todos los eventos involucrados en el servicio de un requerimiento, desde el primer evento, t_0 (tep), hasta el último, t_3 (trr). En particular, en la Fig. 3 se puede notar que el instante de envío de un requerimiento, t_0, por parte del proceso cliente no es el mismo instante en el que el proceso servidor recibe el requerimiento, t_1. La magnitud de este lapso (que transcurre entre los instantes t_0 y t_1) depende básicamente de las comunicaciones entre el proceso cliente y el proceso servidor. Volviendo al entorno de procesamiento en una única computadora con la interacción de un proceso de usuario haciendo un requerimiento al sistema operativo, este tiempo tiene relación directa con un *context switch* o cambio de contexto. De manera análoga, el instante en el que el servidor envía la respuesta una vez procesado el requerimiento, instante t_2, no es el mismo instante en el que el cliente recibe la respuesta, t_3. Una vez más, hay comunicaciones involucradas entre los procesos.

En términos generales, los procesos clientes y los procesos servidores se pueden caracterizar bastante simple y fácilmente:

- Los clientes son activos en la interacción, más específicamente son los que inician o disparan la secuencia requerimiento, procesamiento, respuesta en el tiempo. A priori, no se conoce en el servidor cuándo se producirá el requerimiento de un cliente.

- Los servidores son pasivos en la interacción, actúan por demanda de los requerimientos de los clientes. Puesto de otra manera, solamente procesan/ejecutan código *por* o *en* demanda de los clientes.
- Los clientes son normalmente los que usan o necesitan recursos (de software o hardware). Estos recursos son los necesarios para resolver las necesidades de las aplicaciones de usuario.
- Los servidores normalmente son los que administran los recursos. En algunos casos también acceden directamente a ellos por demanda de los clientes, para resolver sus requerimientos.
- Los clientes normalmente no tienen una visión completa de los recursos que utilizan, solamente los piden a los servidores y los usan (en algunos casos, también los retornan a los servidores) sin tener en cuenta la cantidad total o la cantidad de recursos utilizados en el sistema.
- Los servidores necesariamente tienen que tener una visión más global de los recursos que los clientes, dado que normalmente tienen que administrarlos o usarlos por demanda de los clientes.
- En principio, ni los procesos clientes ni los procesos servidores se tienen que dedicar al transporte de los datos. Todas las comunicaciones se pueden ocultar tanto para el cliente como para el servidor de un requerimiento. Lo que tiene que estar (y normalmente así es) muy bien definido es:
    - Cómo se *pide* un servicio y qué datos son necesarios desde el cliente hacia el servidor
    - Cómo se retorna la respuesta a un cliente, básicamente qué datos son devueltos al cliente en respuesta al servicio requerido.

  Y en este sentido es donde tanto la interacción de los procesos de usuario con el sistema operativo como la definición de protocolos han servido de base para estas definiciones.

La claridad y simplicidad de la definición del modelo cliente/servidor ha sido lo que en principio ha promovido su desarrollo a gran escala en los sistemas distribuidos. En general, los conceptos involucrados son intuitivos y conocidos para los programadores. Sin embargo, cuando se comienzan con los detalles de la programación normalmente surgen inconvenientes que hacen que el desarrollo (puesta a punto, mantenimiento, etc.) de software en los entornos de procesamiento distribuido sea particularmente complicado. Tanto en este apunte como en toda la asignatura se considerarán como sinónimos las expresiones: modelo cliente/servidor, procesamiento cliente/servidor, arquitectura (de software) cliente/servidor, aplicación cliente/servidor y sistemas cliente/servidor.

2. Distintos Niveles de Abstracción

Tal como se explica antes, la arquitectura cliente/servidor es (o debería ser) independiente de la forma o método de comunicación entre procesos, específicamente entre los procesos cliente y servidor. De hecho, en [1] cliente/servidor se clasifica como de menor nivel de abstracción (mayor nivel de detalles) que RPC (Remote Procedure Call) y RMI (Remote Method Invocation), que se consideran equivalentes. En realidad, se puede implementar fácilmente cliente/servidor utilizando RPC [3] [4] o RMI [5]. De hecho, mucha de la bibliografía clásica de procesamiento distribuido, tal como [6], *muestra* el procesamiento cliente/servidor con el cliente ejecutando un RPC (haciendo una llamada remota) y el servidor recibiendo el requerimiento directamente en un procedimiento local que implementa el servicio y retorna la respuesta. En este sentido, se podría decir que

- RPC y RMI son mecanismos o construcciones de los lenguajes para extender la funcionalidad de los mismos en entornos distribuidos. Es decir que la idea de RPC como de RMI es ejecutar procedimientos o métodos sobre objetos que no son necesariamente locales, *como si fueran* locales. Estas extensiones de los lenguajes son justamente esto: extensiones de los lenguajes (normalmente RPC asociado a C y RMI asociado a Java) para procesamiento distribuido.
- Cliente/servidor es un modelo de procesamiento distribuido, donde en todas las aplicaciones se pueden identificar procesos clientes y uno o más procesos servidores, cada uno de ellos con las características bien definidas que se dieron anteriormente. En este sentido la arquitectura cliente/servidor es independiente de RPC o RMI aunque se puede implementar con estos mecanismos de llamada/invocación remota a procedimientos/métodos.

En la práctica, la forma de implementar cliente/servidor puede hacerse siguiendo el modelo en *capas*, tal como en el contexto de los protocolos. Entre los clientes y servidores podemos identificar la *capa* o

protocolo de aplicación. Esta capa está asociada, justamente, a la especificación de los detalles de los servicios, es decir:

- Nombre del servicio o, lo que es equivalente, cómo se hace un requerimiento al proceso servidor desde el proceso cliente.
- Datos o parámetros del servicio, es decir los datos que el proceso cliente envía al proceso servidor. Estos datos son los que el servidor usará para proveer el servicio específico del cliente. El servidor, por su parte puede hacer uso de otros datos o recursos dependiendo del requerimiento del proceso cliente.
- Datos o parámetros de respuesta del servicio, es decir los datos que el proceso servidor envía al proceso cliente una vez que ha terminado de procesar el servicio requerido. Estos datos varían según el servicio y van desde la información que indica si el servicio fue resuelto satisfactoriamente o no hasta datos que son el resultado del procesamiento del servicio mismo (un servicio de la lectura de un archivo que se ha resuelto satisfactoriamente, por ejemplo, da como resultados los datos leídos).

La Fig. 4 muestra esta visión de capas, donde está bien definido el protocolo de aplicación, y las capas inferiores de software son las que proveen localmente los servicios para la implementación del protocolo del nivel de aplicación cliente/servidor. En la Fig. 4 también se muestran los distintos ámbitos de ejecución (computadoras) del proceso cliente y del proceso servidor, aunque no habría ningún impedimento para que ambos procesos se ejecuten en la misma computadora.

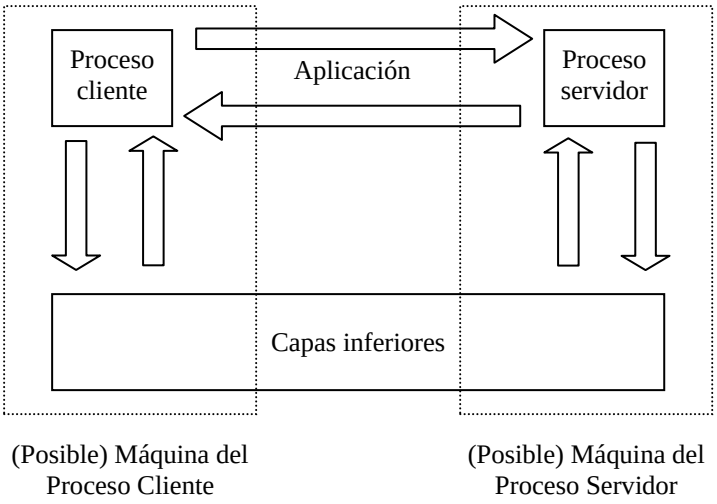

Figura 4: Cliente/Servidor a Nivel de Aplicación.

En realidad, cuando se implementa una aplicación cliente/servidor y es posible enfocarse *solamente* en el nivel aplicación, las capas inferiores no son más que una *referencia*, es decir que se conoce su existencia pero no se debe dedicar mucho esfuerzo a ella. Normalmente, todo lo que se hace es una especificación en algún lenguaje del protocolo de aplicación de forma tal que esa especificación pueda trasladarse automáticamente (normalmente esto está asociado a la etapa de compilación de la especificación) a las capas inferiores. Esto es posible de manera directa tanto con RPC como con RMI, aunque cada una de estos mecanismos son específicos/asociados a un lenguaje de programación en particular.

Si no se tiene la posibilidad de especificar solamente la capa de aplicación, evidentemente se deberán resolver también los detalles de una o más de las capas inferiores que se muestran *ocultas* en la Fig. 4, bajo el título “capas inferiores”. En estas capas inferiores se deberán resolver detalles como:

- Representación de parámetros en general, los datos a ser enviados/recibidos entre el cliente y el servidor, que son propios de los servicios.
- Representación de estructuras de datos compuestas/complejas definidas por el usuario. Deben resolverse los detalles de, por ejemplo, los datos que se mantienen en estructuras dinámicas (con punteros de memoria).

- Representación de datos simples. Deben resolverse los detalles de, por ejemplo, las diferentes representaciones en longitud (cantidad) y orden ("*endian*") de los bytes de un dato simple como un número en punto flotante.
- Forma o capa de transporte de los datos. En este caso, los detalles están relacionados directamente con el protocolo utilizado (TCP, por ejemplo) y con la interfase (API) usada para acceder a ese protocolo desde el lenguaje elegido (sockets en C, por ejemplo).

Como se aclara antes, la implementación del modelo cliente/servidor no determina o restringe el modelo mismo (no debería, al menos) sino que debería hacer uso de lo más apropiado para resolver la aplicación. En este sentido, no es mucho más que seguir los pasos apropiados de ingeniería de software para cualquier aplicación. Así como para implementar una aplicación se tienen distintos lenguajes de programación, cuando se resuelven aplicaciones cliente/servidor en particular se utiliza (o se debería utilizar) lo más apropiado para implementar. En la sección siguiente se dan las ideas introductorias para dos de las formas de implementación de cliente/servidor que son análogas: RPC para los entornos de programación procedurales y RMI para Java.

3. C/S con RPC o RMI

Tanto con RPC (Remote Procedure Call) como con RMI (Remote Method Invocation) es posible implementar el modelo cliente/servidor desde la perspectiva de la capa de aplicación. Como se aclara antes, RPC es el mecanismo que intenta trasladar el modelo de programación y procesamiento procedural a las arquitecturas distribuidas. La idea subyacente en este sentido es que sea posible hacer una llamada a procedimiento en otro proceso de manera similar (o "igual") a como se hace una llamada a procedimiento estándar. El mecanismo definido por RMI, por su lado, intenta trasladar la programación y el procesamiento con objetos a las arquitecturas distribuidas. En este contexto, la idea subyacente es similar a la anterior pero con objetos: que la invocación a un método de un objeto se pueda llevar a cabo más allá de que ese objeto/método no estén en la máquina virtual donde se ejecuta la invocación.

3.1. Con RPC

En el caso de RPC, la idea básica es recurrir a un lenguaje "nuevo" de especificación. En este lenguaje de RPC (derivado del lenguaje XDR [2] para representación de datos) se especifican, justamente, los detalles de los procedimientos que se invocarán de forma remota, es decir todo lo relacionado con la interfase de estos procedimientos. Estos detalles incluyen los nombres de los procedimientos y los parámetros involucrados por cada uno de estos procedimientos. Con RPC se debe tener alguna forma de compilador que haga, justamente, la traducción de esta especificación a un lenguaje de programación en el cual se resuelven/programan no solamente los procedimientos sino toda la aplicación cliente/servidor. De hecho, se asume que los clientes son los que ejecutarán las llamadas a procedimientos remotos, donde cada uno de estos procedimientos pertenecerá a un servidor. Por lo tanto, el lenguaje de RPC en sí mismo no resuelve la aplicación cliente/servidor sino lo relacionado a comunicación y sincronización entre el cliente y el servidor, en este caso con llamadas a procedimientos desde el cliente hacia el servidor. La Fig. 5 muestra esquemáticamente la idea de desarrollo/implementación/ejecución con RPC. Como paso previo a las etapas tradicionales de programación procedural, con sus tareas asociadas de compilación y enlce (linking), se agrega la especificación de interfases para RPC, que normalmente se traducen automáticamente a un lenguaje de programación procedural, como el lenguaje C.

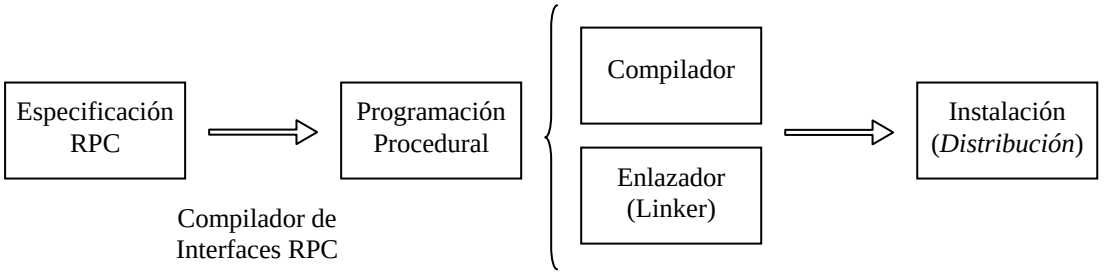

Figura 5: Pasos de Desarrollo y Ejecución con RPC.

En la Fig. 5 aparece como una etapa posterior a la de la creación de los ejecutables (binarios) la etapa de instalación. Aunque siguiendo el modelo cliente/servidor esta etapa se simplifica bastante, en los ambientes distribuidos la instalación/distribución de software tiene un costo extra relativo a la instalación de software en un ambiente acotado de una única computadora. En el caso particular de las aplicaciones cliente/servidor, se tiene que poner cierta atención a la ubicación (computadora) donde se ejecutará el servidor y, por otro lado, donde se ejecutará/n el o los clientes. El caso más sencillo de un único cliente y un único servidor no representa mayores complicaciones. Sin embargo, esta configuración de un único cliente con un único servidor no es la más común, dado que en general existen muchas otras alternativas más complicadas, tales como:

- Varios clientes con un único servidor.
- Varios clientes con varios servidores.
- Varios clientes con varios servidores replicados (varios servidores pueden dar un servicio determinado).

Es así que en los sistemas distribuidos más complejos, se llega a planificar toda una etapa de deployment, que es justamente dedicada a la distribución del software que corresponde en cada plataforma (o máquina) particular del hardware distribuido.

3.2. Con RMI

La diferencia fundamental del ciclo de desarrollo de software distribuido usando RMI (más allá de las diferencias propias de la programación procedural con la programación orientada a objetos), es que RMI en realidad está directamente definido en/para Java [5]. Esto hace que el desarrollo no implica mucho más que conocer la forma de especificar en Java cómo hacer RMI. Por un lado, esto significa una simplificación para quienes desarrollan aplicaciones distribuidas en Java (sea siguiendo el modelo cliente/servidor o no) pero por otro también restringe el uso de RMI directamente a Java, habiendo otros lenguajes orientados a objetos. La fig. 6 muestra el caso particular del desarrollo de aplicaciones con RMI, donde todo lo que hay que hacer “extra” respecto de las aplicaciones no distribuidas es la identificación de las clases que contendrán la definición de métodos que luego podrán ser invocados de manera remota.

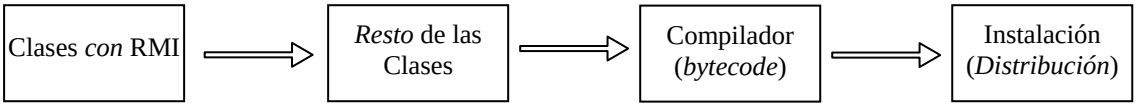

Figura 6: Pasos de Desarrollo y Ejecución con RMI.

Es interesante notar que, a pesar de que se *siguen* programando en Java las aplicaciones distribuidas es necesaria la identificación de las clases con métodos que se pueden invocar de manera remota. Por un lado, esto hace que la programación de aplicaciones distribuidas implique una tarea más en cuanto a que es necesaria la identificación explícita de las estas clases “especiales”. Por otro lado, también aporta “claridad” desde la perspectiva de que estas clases son efectivamente diferentes del resto justamente por estar en entorno de desarrollo y ejecución de software distribuido. En el caso particular de las aplicaciones cliente/servidor, las clases *con* RMI serán, justamente, las asociadas al o a los servidores, mientras que el resto de las clases serán posiblemente *propias* de los clientes o necesarias para el funcionamiento del o de los servidores. Las ideas que se comentaron antes respecto de instalación/distribución/deployment del software es la misma que en el caso de RPC. Necesariamente se tienen que planificar los detalles de dónde se instala cada clase, cliente/s y servidor/es en el caso particular de aplicaciones cliente/servidor.

4. Conclusiones

El modelo cliente/servidor es de los primeros y más usados en entornos de procesamiento distribuido. Quizás una de las razones más importantes para su utilización es, justamente, su simplicidad la claridad en su definición, muy relacionada con ideas bien conocidas de sistemas operativos, redes y programación *tradicionales*.

Tanto RPC como RMI no necesariamente están asociados a la implementación del modelo cliente/servidor pero sin lugar a dudas proveen todo lo necesario para mantener la visión de cliente/ servidor en la capa de aplicación, sin necesidad de conocer detalles de codificación y transporte de datos. RPC es, en principio, independiente del lenguaje procedural en el que se programen las aplicaciones, mientras que RMI está asociado/definido estrictamente para Java. En cualquier caso, la instalación de software para ejecución en un ambiente distribuido *también* es más compleja que en un entorno no distribuido.

5. Bibliografía

- [1] Liu, M. L., Distributed Computing: Principles and Applications, Addison Wesley; 1st edition, ISBN: 0201796449, June, 2003.
- [2] Sun Microsystems, XDR: External Data Representation Standard, RFC 1014, June 1987.
- [3] Sun Microsystems, RPC: Remote procedure call protocol specification, RFC 1050, April 1988.
- [4] Sun Microsystems, RPC: Remote Procedure Call Protocol Specification, Version 2, RFC 1057, June 1988.
- [5] Sun Microsystems, Inc, Java Remote Method Invocation Specification, http://java.sun.com/ j2se/1.4.2/docs/guide/rmi/spec/rmi-title.html
- [6] Tanenbaum, A. S., M. Van Steen, Distributed Systems – Principles and Paradigms, Prentice Hall, 2002.