Sistemas distribuidos

Grado en Ingeniería Informática

Tema 02-02: Comunicación entre procesos

Departamento de Ingeniería Informática
Universidad de Cádiz

Escuela Superior de Ingeniería
Dpto. de Ingeniería Informática

Curso 2019 – 2020

Indice

Sección 1 | Introducción

Introducción (I)

- Características de la comunicación entre procesos:
  - Comunicación síncrona y asíncrona.
  - Destinos de los mensajes.
  - Fiabilidad.
  - Ordenación.
- Las entidades que se comunican son procesos cuyos papeles determinan cómo se comunican ("patrones de comunicación").
- La aplicación se comunica con:
  - UDP con "paso de mensajes".
  - TCP con "flujo de datos".

Introducción (II)

- Los patrones de comunicación principales son:
  - Comunicación cliente-servidor.
  - Comunicacion en grupo (multidifusión).
- Es necesario el diseño de protocolos de alto nivel que soporten dichos patrones.
- Tipos de comunicación:
  - Recurso compartido.
  - Paso de mensajes.

Sección 2 | API para los Protocolos de Interntet

Comunicación entre procesos (I)

- El paso de un mensaje se puede llevar a cabo mediante 2 operaciones de comunicación:
  - *send*: Un proceso envia un mensaje a un destino.
  - *receive*: Un proceso recibe el mensaje en el destino.
- Cada destino tiene asociada una cola de mensajes:
  - Los emisores añaden mensajes a la cola.
  - Los receptores los extraen.
- Características de la comunicación entre procesos:
  - Comunicación síncrona y asíncrona.
  - Destinos de los mensajes.
  - Fiabilidad.
  - Ordenación.

Comunicación entre procesos (II)

Comunicación síncrona

- El emisor y el receptor se sincronizan en cada mensaje.
- *send* y *receive* son operaciones bloqueantes:
  - El emisor se bloquea hasta que el receptor realiza la operación *receive*.
  - El receptor se bloquea hasta que le llegue un mensaje.

Comunicación entre procesos (III)

Comunicación asíncrona (I)

- La operación *send* es no bloqueante:
  - El mensaje se copia a un búfer local.
  - El mensaje continúa aunque todavía no exista un *receive* (bloqueante o no bloqueante).
- *Receive* no bloqueante:
  - El proceso receptor sigue con su programa después de invocar la operación *receive*.
  - Proporciona un búfer que se llenarán en segundo plano.
  - El proceso debe ser informado por separado de que su búfer ha sido llenado (sondeo o interrupción).

Comunicación entre procesos (III)

Comunicación asíncrona (II)

- *Receive* bloqueante:
  - En entornos que soportan múltiples hilos en un proceso:
    - Este *receive* puede ser invocado por un hilo mientras que el resto de hilos del proceso permanecen activos.
    - Simplicidad de sincronizar los hilos receptores con el mensaje entrante.
- El *receive* no bloqueante es más eficiente, pero más complejo (necesidad de capturar el mensaje entrante fuera de su flujo de control).

Comunicación entre procesos (IV)

Destino de los mensajes

- Los mensajes son enviados a direcciones construidas por pares (dirección Internet, puerto local).
- Un puerto local:
  - Es el destino de un mensaje dentro de un computador (número entero).
  - Tiene exactamente un receptor pero puede tener muchos emisores.
  - Los procesos pueden utilizar múltiples puertos desde los que recibir mensajes.
  - Cualquier proceso que conozca el número de puerto puede enviarle un mensaje.

Comunicación entre procesos IV)

Fiabilidad

- Comunicación punto a punto fiable:
  - Se garantiza la entrega, aunque se pierda un número razonable de paquetes.
- Comunicación no fiable:
  - La entrega no se garantiza, aunque sólo se pierda un único paquete.

Ordenación

Algunas aplicaciones necesitan que los mensajes sean entregados en el orden de su emisión.

Sockets (I)

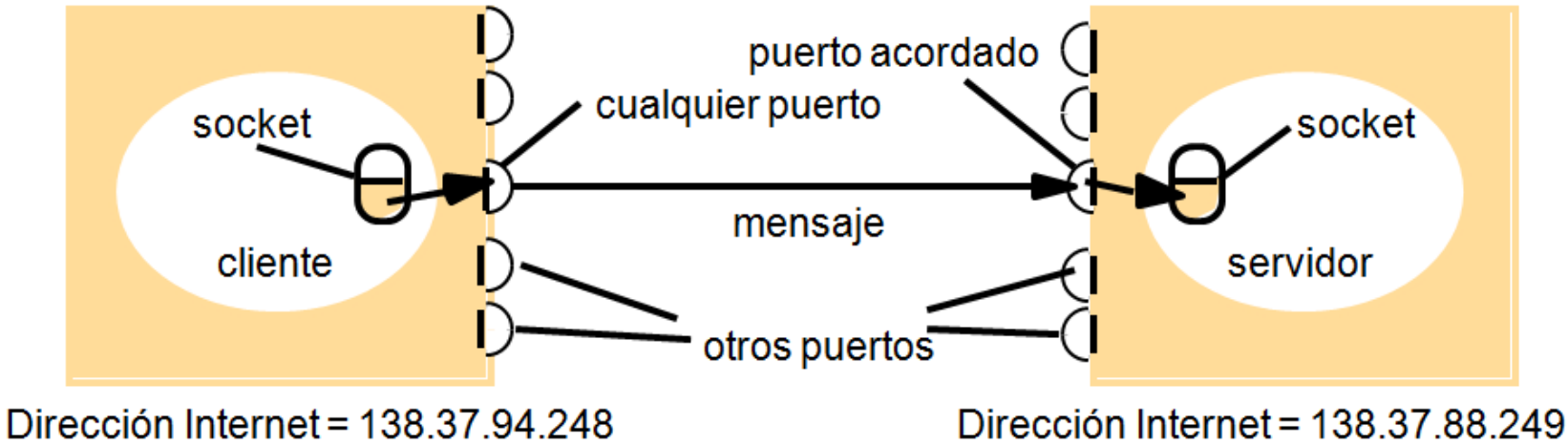
- El mecanismo original de *Unix Interprocess Communication* (IPC) son *pipes*:
  - Cauce unidireccional (flujo), sin nombre.
  - Enlazan filtros, sin sincronización explícita.
- *pipeline*: un mismo padre crea los procesos filtro y *pipes*. Ej.: *gunzip -c fichero.tar.gz | tar xvf -*.
- Útil en entornos productor-consumidor.
- No es útil en sistemas distribuidos:
  - No hay un nombre.
  - Sin posibilidad de enviar mensajes discretos.
- IPC implementada como llamadas al sistema, a partir de UNIX BSD 4.2.

Sockets (II)

- *Socket* (destino de mensajes):
  - Se pueden enviar mensajes mediante un socket.
  - Y permite recibir mensajes.
- La IPC se da entre 2 sockets.
- Cada socket debe estar asociado a:
  - Un puerto local de su máquina.
  - Una dirección IP de esa máquina.
  - Un protocolo (UDP o TCP).
- La interfaz de los sockets proporciona un conjunto de funciones para la comunicación por paso de mensajes:
  - Basada en el modelo cliente/servidor.
  - Tiene 2 servicios en TCP/IP:
    - Mensajes de conexión (TCP).
    - Mensajes sin conexión (UDP).
  - Interfaces: Sockets (Unix) y WinSock (Windows).

Sockets (III)

El proceso que posee el socket es el único que puede recibir mensajes destinados al puerto asociado.



Sockets (IV)

Interfaz Sockets: funciones UDP

- Cliente:
  - Crear socket.
  - Enviar/recibir.
  - Cerrar socket.
- Servidor:
  - Crear socket.
  - Enviar/recibir.
  - Cerrar socket.

Sockets (V)

Interfaz Sockets: funciones TCP

- Cliente:
  - Crear socket.
  - Conectarse.
  - Enviar/recibir.
  - Cerrar socket.
- Servidor:
  - Crear socket.
  - Enlazar a *port* y *port* (*bind*).
  - Escuchar.
  - Aceptar conexiones.
  - Enviar/recibir.
  - Cerrar socket.

Comunicación de datagramas UDP (I)

- Mensaje autocontenido no fiable desde un emisor a un receptor.
- Único mensaje sin asentimientos ni reenvíos (no hay garantía de entrega).
- Trasmisión entre 2 procesos: *send* y *receive*.
- Cada proceso debe crear un socket y enlazarlo a un puerto local:
  - Los clientes deben enlazarlo a cualquier puerto local libre.
  - Los servidores, a un puerto de servicio determinado.
- La operación *receive* entrega el mensaje transmitido y el puerto al que está enlazado el socket emisor.
- Se utiliza comunicación asíncrona con *receive* bloqueante:
  - *send* devuelve el control cuando ha dirigido el mensaje a las capas inferiores UDP e IP (responsables de su entrega en el destino).
  - Estas capas lo transmiten y lo dejan en la cola sel socket asociado al puerto de destino.

Comunicación de datagramas UDP (II)

- Se utiliza comunicación asíncrona con *receive* bloqueante (cont.):
  - *receive* extrae el mensaje de la cola con un bloqueo indefinido (por defecto), a menos que se haya establecido un tiempo límite (*timeout*) asociado al conector.
  - En general, se puede enviar y recibir de cualquier puerto, aunque se puede limitar a uno concreto.
- No hay garantía de la recepción:
  - Los mensajes se pueden perder por errores de *checksum* o falta de espacio.
  - Los procesos deben proveer la calidad que deseen.
- Se puede dar un servicio fiable sobre uno no fiable, si se añaden asentimientos.
- No se suele utilizar una comunicación totalmente fiable:
  - No es imprescindible.
  - Provoca grandes cargas administrativas:
    - Almacena información de estado en origen y destino.
    - Transmite mensajes adicionales.
    - Puede existir latencia para emisor o receptor.

Comunicación de flujos TCP (I)

- La abstracción de flujos (*streams*) oculta:
  - Tamaño de los mensajes: los procesos leen o escriben cuanto quieren y las capas inferiores (TCP/IP) se encargan de empaquetar.
  - Mensajes perdidos: a través de asentimientos y reenvíos.
  - Control de flujo: evita desbordamiento del receptor.
  - Mensajes duplicados y/o desordenados: a través de identificadores de mensajes.
  - Destinatarios de los mensajes: tras la conexión, los procesos leen y escriben del cauce sin tener que utilizar nuevamente sus respectivas direcciones.
- Se distiguen claramente las funciones del cliente y servidor:
  - Cliente: crea un socket encauzado y solicita el establecimiento de una conexión.
  - Servidor: crea un socket de escucha con una cola de peticiones de conexión, asociado a un número de puerto y se queda a la espera de peticiones de conexión.

Comunicación de flujos TCP (II)

- Al aceptar una conexión el servidor:
  - Se crea automáticamente un nuevo socket encauzado conectado al del cliente.
  - Cada proceso lee de su cauce de entrada y escribe en su cauce de salida.
  - Si un proceso cierra su socket, los datos pendientes se transmitirán y se indicará que el cauce está roto.
- Puede haber bloqueo:
  - Lectura: no hay datos disponibles.
  - Escritura: la cola del socket de destino está llena.
- Opciones para atender a múltiples clientes:
  - Escucha selectiva.
  - Multitarea.
- La conexión se romperá si se detectan errores graves de red.

Sección 3 | Ejemplos de programación TCP/UDP

Comunicación de datagrama UDP API Java (I)

- Java proporciona 2 clases:
  - *DatagramPacket*.
  - *DatagramSocket*.

DatagramPacket

- Soporte a los datagramas.
- El constructor que utilizan los emisores recibe: un mensaje, su longitud, la dirección IP de la máquina de destino y el número de puerto local del socket destinatario.
- El constructor que usan los receptores recibe: un array de bytes para un mensaje y su longitud.
- Otros métodos adicionales: *getData*, *getAddress*, *getLenght* y *getPort*.

Comunicación de datagrama UDP API Java (II)

DatagramSocket

- Soporta a los sockets.
- El constructor que utilizan los servidores recibe: el número de puerto local que se desea asociar.
- El constructor que utilizan los clientes no proporciona ningún argumento (elige uno que esté libre).
- Otros métodos adicionales:
  - *send* y *receive*: su argumento es de *DatagramPacket*.
  - *setSoTimeout*: establece una temporización.
  - *connect*: limita que el socket local sólo pueda enviar y recibir mensajes de un puerto remoto.

Comunicación de flujos TCP API JAVA (I)

- Java proporciona 2 clases:
  - *ServerSocket*.
  - *Socket*.

ServerSocket

- Soporte para los sockets de escucha (servidores).
- Método *accept*:
  - Si la cola de solicitudes de conexión está vacía, se bloquea.
  - En caso contrario, toma una solicitud, crea un ejemplar de la clase *Socket* y establece la conexión.

Comunicación de flujos TCP API JAVA (II)

Socket

- Soporte para los sockets encauzados.
- El cliente utiliza un constructor:
  - Argumentos: nombre del ordenador y el número de puerto del servidor.
  - Crea el socket y solicita automáticamente la conexión.
- Servidor: resultado del *accept*.
- *getInputStream* y *getOutputStream* son métodos que:
  - Devuelven valores de tipo *InputStream* y *OutputStream*.
  - Pueden utilizarse como argumentos para constructores de cauces de entrada/salida.

Ejemplo UDP Java

Emisor

```
InetAddress receiverHost=InetAddress.getByName("localHost");
DatagramSocket theSocket = new DatagramSocket( );
String message = "Hello␣world!";
byte[ ] data = message.getBytes( );
DatagramPacket thePacket //remote port is specified in datagram
= new DatagramPacket(data, data.length, receiverHost, 2345);
theSocket.send(thePacket);
```

Receptor

```
DatagramSocket ds = new DatagramSocket(2345);
DatagramPacket dp = new DatagramPacket(buffer, MAXLEN);
ds.receive(dp);
len = dp.getLength( );
System.out.Println(len + "␣bytes␣received.\n");
String s = new String(dp.getData( ), 0, len);
System.out.println(dp.getAddress( ) + "␣at␣port␣"
+ dp.getPort( ) + "␣says␣" + s);
```

Ejemplo TCP Java

Server

```
ServerSocket connectionSocket = new ServerSocket(19999);
Socket dataSocket = connectionSocket.accept();
OutputStream outStream = dataSocket.getOutputStream();
PrintWriter socketOutput =
new PrintWriter(new OutputStreamWriter(outStream));
socketOutput.println("the␣message");
socketOutput.flush();  dataSocket.close( ); connectionSocket.close( )
```

Cliente

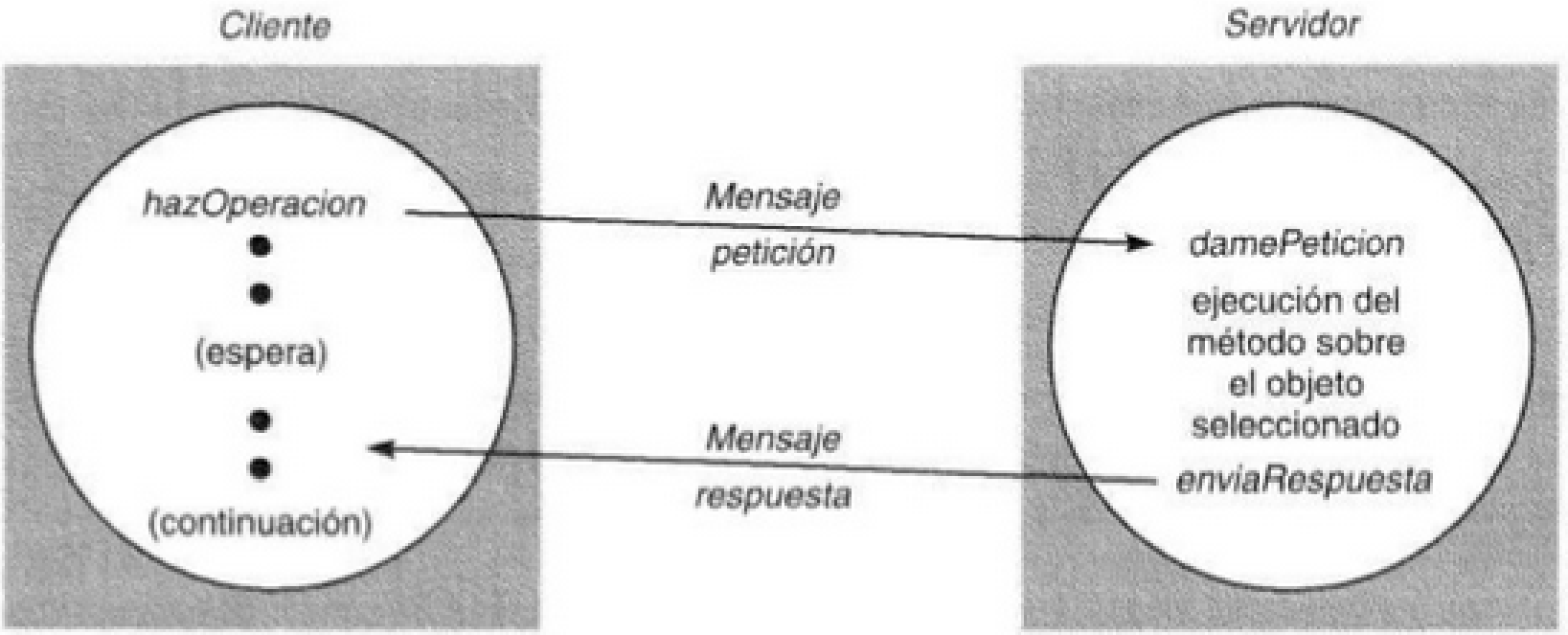
```
InetAddress acceptorHost = InetAddress.getByName("server.com");
Socket mySocket = new Socket(acceptorHost, 19999);
InputStream inStream = mySocket.getInputStream();
BufferedReader socketInput =
new BufferedReader(new InputStreamReader(inStream));
String message = socketInput.readLine( );
System.out.println("\t" + message);
mySocket.close( );
```

Sección 4 | Comunicación cliente-servidor

Protocolo petición-respuesta (I)

- Protocolo:
  - Identificadores de mensaje: tipoMensaje (0=petición, 1=respuesta), idPetición (int), referenciaObjeto (RemoteObjectRef), idMétodo (int o Method) y argumentos (cadena de bytes).
  - UDP.
- Modelo de fallos:
  - Tiempo de espera límite.
  - Eliminación de mensajes de petición duplicados.
  - Pérdida de mensajes de respuesta.
  - Historial.
- La comunicación cliente-servidor es síncrona (el cliente se bloquea hasta recibir una respuesta).
- *send-receive* requiere 4 llamadas al sistema para un intercambio.
- Existen protocolos (Chorus, Amoeba, Mach y V) que requieren sólo 3 llamadas al sistema: *doOperation*, *getRequest* y *sendReply*.

Protocolo petición-respuesta (II)



Identificadores de la invocación (I)

- El cliente genera una *idInvocacion* única donde:
  - Se añade al mensaje de solicitud.
  - El servidor la copia a su respuesta.
  - El cliente comprueba si es la esperada.
- Deben ser únicos y tienen 2 partes:
  - *IdInvocacion*: entero secuencial elegido por el emisor que ofrece unicidad entre invocaciones de un mismo emisor.
  - Un identificador del remitente que ofrece unicidad global. Por ejemplo, la dirección IP y el número de puerto donde se espera la respuesta.

Fallos en la entrega

- Los mensajes se pueden perder.
- Puede haber rotura: parte de la red aislada.
- Los procesos pueden fallar.
- Dificultad de elegir un N reintentos.
- Los datos si se reciben serán correctos.
- Se incorpora un temporizador al *receive* del cliente.

Temporizadores

- Cuando vence el temporizador del *receive* del cliente, el módulo de comunicaciones puede:
  - Retornar inmediatamente indicando el fallo ocurrido (poco común).
  - Reintentarlo repetidamente hasta obtener una respuesta o cuando exista probabilidad de que el servidor ha fallado.

Solicitudes duplicadas

- El servidor puede recibir solicitudes duplicadas, si los reintentos llegan antes de tiempo (servidor lento o sobrecargado).
- Para evitar estas ejecuciones repetidas:
  - Reconocer los duplicados del mismo cliente (igual *idInvocacion*).
  - Filtarlos (descartarlos)

Respuestas perdidas

- Provocan que el servidor repita una operación.
- No es un problema cuando las operaciones del servidor son idempotentes:
  - Se pueden de realizar de forma repetida.
  - Los resultados son los mismos que si se ejecutasen una sola vez.

Historial

- Objetivo: retransmitir una respuesta sin volver a ejecutar la operación.
- El historial es el registro de las respuestas enviadas:
  - Mensaje con *idInvocacion* y su destinatario.
  - Gran consumo de memoria: se podrían descartar las respuestas tras algún tiempo.

Protocolos de intercambio RPC (I)

- 3 protocolos que se suelen utilizar:
  - R (*request* o petición).
  - RR (*request-reply* o petición-respuesta).
  - RRA (*request-reply-acknowledge reply* o petición-respuesta-confirmación de la respuesta).

Nombre	Cliente	Mensajes enviados por Servidor	Mensajes enviados por Cliente
R	Petición		
RR	Petición	Respuesta	
RRA	Petición	Respuesta	Confirmación respuesta

Protocolos de intercambio RPC (II)

Protocolo R

- Sólo es útil cuando no hay valor de retorno del procedimiento y el cliente no necesita información.
- El cliente continúa tras enviar la solicitud.

Protocolo RR

- Común en entornos cliente-servidor.
- La respuesta asiente la solicitud.
- Una solicitud posterior del mismo cliente asiente la respuesta.

Protocolos de intercambio RPC (III)

Protocolo RRA

- La respuesta asiente la solicitud.
- El asentimiento de la respuesta lleva la *idInvocacion* de la respuesta a la que se refiere, asiente dicha respuesta y la de *Idvocacion* anterior, y permite vaciar entradas del historial.
- El envío del asentimiento de respuesta no bloquea al cliente pero consume recursos de procesador y red.

Ejemplo de protocolo de comunicación: HTTP

- La longitud de UDP podría no ser adecuada.
- Gracias a TCP se asegura que los datos sean entregados de forma fiable.
- El protocolo de petición-respuesta más conocido que utiliza TCP es HTTP:
  - Métodos: GET, PUT, POST, HEAD, DELETE, OPTIONS.
  - Permite: autenticación y negociación del contenido.
  - Establece conexiones persistentes, y abiertas durante el intercambio de mensajes.

Ejemplo en Python

socketHTTP.py

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ip = socket.gethostbyname("www.marca.com")
sock.connect((ip, 80))
sock.sendall( b"GET␣/␣HTTP/1.1\nHost:␣www.marca.com\n\n" )
bufsize = 4000
output = "␣"
buf = sock.recv ( bufsize )
while (buf):
        output += buf #buf.encode("utf-8")
        print(output)
        print("-------------------------")
        buf = sock.recv ( bufsize )
```

Sección 5 | Comunicación multicast

Multidifusión

- Multidifusión envía un único mensaje a todos los miembros de un grupo de forma transparente.
- No hay garantía de la entrega del mensaje ni del orden de los mensajes.
- Proporciona la infraestructura para desarrollar sistemas distribuidos con las características:
  - Tolerancia a fallos.
  - Búsqueda de los servidores de descubrimiento de redes espontáneas.
  - Mejores prestaciones basadas en datos replicados.
  - Propagación de notificaciones de eventos.

Sección 6 | Tareas

Tareas

1. Describa dos escenarios en los que sea necesario comunicación síncrona, y dos escenarios de comunicación asíncrona.
2. ¿Resulta razonablemente útil que un puerto tenga varios receptores?
3. Un servidor crea un puerto que utiliza para recibir peticiones de sus clientes. Discuta los problemas de diseño concernientes a las relaciones entre el nombre de este puerto y los nombres utilizados por los clientes:
   - ¿Cómo sabe el cliente qué puerto y dirección IP utilizar para acceder a un servicio?
   - Eficiencia de acceso a puertos e identificadores locales.

Bibliografía

Coulouris, G.; Dollimore, J.; Kindberg, T.
Distributed Systems: Concepts and Design (5ª ed.)
Addison-Wesley, 2012.
(Trad. al castellano: Sistemas distribuidos: conceptos y diseño, 3ª ed., Pearson 2001)