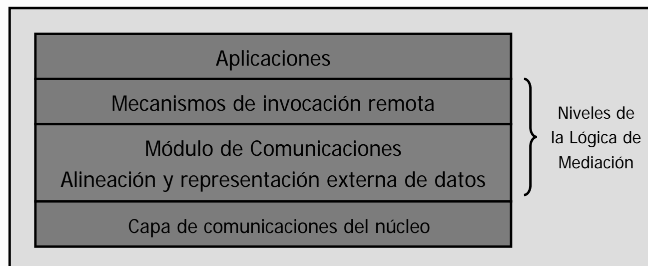
Tema 2. La comunicación entre procesos

1. Introducción
2. La API de los protocolos de Internet
3. La API *C* de *sockets* en UNIX
4. La alineación y la representación externa de datos
5. El módulo de comunicaciones cliente-servidor

Introducción y objetivos

Introducción. 1

- Estudiaremos el nivel inferior de la Lógica de Mediación (*middleware*)



- Discutiremos el uso de *TCP-UDP/IP* desde el punto de vista del programador

Introducción. 2

- Las entidades que se comunican son procesos
  - sus papeles determinan cómo se comunican
  - es decir, sus patrones de comunicación
- 2 patrones de comunicación principales:
  - comunicación cliente-servidor
  - comunicación en grupo
- Hay que diseñar protocolos de alto nivel que soporten dichos patrones

Objetivos. 1

- Desarrollar bloques constructivos para la IPC:
  - Cómo meter los datos en los mensajes
  - Cómo pasar los mensajes:
    - Qué semántica emplear
    - Transparencia, sincronismo, fiabilidad
- Construir protocolos a la medida
  - de los papeles de los procesos
  - de los patrones de comunicación

Objetivos. 2

- Comunicación cliente-servidor
  - Protocolos solicitud-respuesta
    - Tener en cuenta los papeles de los procesos
    - Qué primitivas de comunicación usar
    - Evitación de redundancias (ej.: asentimientos)
- Afrontar la ausencia de respuesta
  - ¿Es correcto reintentar una operación?

Objetivos. 3

- Presentar la interfaz de *sockets BSD* para:
  - *TCP*. Abstracción: cauce bidireccional (*stream*)
    - Información encauzada sin fronteras de mensaje
    - Uso de *buffers*: amortiguar diferencias de velocidad
    - En SD: *ftp*, *http*, *telnet*, *smtp*
    - Además: entornos productor-consumidor
  - *UDP*. Abstracción: paso de mensajes (*datagramas*)
    - Envío de un mensaje autocontenido desde un emisor hacia un receptor
    - En SD: *DNS*, *NFS*, *NTP*
- En la API *Java* y en *UNIX*: destino = *socket*
  - referencia indirecta a un puerto concreto del receptor

La API de los protocolos de Internet

Las operaciones *send* y *receive*

- El paso de un mensaje se puede soportar con dos operaciones de comunicación:
  - *send*
    - un proceso envía un mensaje a un destino
  - *receive*
    - un proceso recibe el mensaje en el destino
- Cada destino tiene asociada una *cola*
  - los emisores añaden mensajes a la cola
  - los receptores los extraen

Tema 2 La comunicación entre procesos 9

Comunicación síncrona y asíncrona. 1

- El paso de un mensaje
  - Implica: comunicación de datos
    - desde el proceso emisor hacia el receptor
  - Puede implicar: sincronización de los procesos
- Comunicación síncrona:
  - emisor y receptor se sincronizan en cada mensaje
  - el *send* y el *receive* son operaciones bloqueantes
    - el emisor se bloquea hasta que el receptor hace *receive*
    - el receptor se bloquea hasta que llega un mensaje

Tema 2 La comunicación entre procesos 10

Comunicación síncrona y asíncrona. 2

- Comunicación asíncrona:
  - el *send* es no bloqueante
    - el mensaje se copia a un *buffer* local y
    - el emisor continúa (aunque aún no haya un *receive*)
  - el *receive* puede ser bloqueante o no bloqueante
  - *receive* no bloqueante:
    - el receptor provee un *buffer* para cuando llegue el mensaje
    - y continúa tras emitir el *receive* (aunque no haya mensaje)
    - Implica notificación (sondeo, interrupción)

Tema 2 La comunicación entre procesos 11

Comunicación síncrona y asíncrona. 3

- Comunicación asíncrona con *receive* bloqueante
  - En entornos multitarea
    - pocas desventajas: otras tareas pueden seguir activas
    - grandes ventajas: tareas receptoras sincronizadas a mensajes entrantes
  - el *receive* no bloqueante parece más eficiente
    - pero es más complejo (¿interrupciones? no, gracias)
  - Los servidores pueden esperar indefinidamente, pero
    - en otros casos: temporización (*timeout*)

Tema 2 La comunicación entre procesos 12

La fiabilidad

- Comunicación punto a punto fiable:
  - se garantiza la entrega
  - aunque se pierdan paquetes
    - es decir, al menos hasta que se pierda un número *razonable* de paquetes
- No fiable:
  - la entrega no se garantiza
  - aunque sólo se pierda un único paquete

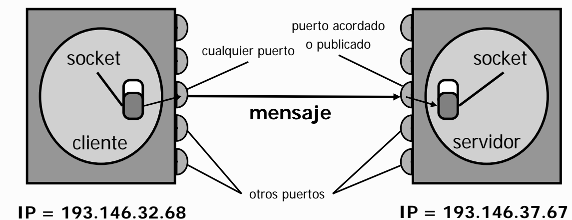
Los *sockets*. 1

- Mecanismo original de IPC en *UNIX*: *pipes*
  - cauce unidireccional (*stream*) y sin nombre
  - enlazan filtros, sin sincronización explícita
    - pipeline. Ej.: *gunzip -c fich.tar.gz | tar xvf -*
    - un mismo padre crea los procesos filtro y los pipes
- Útil en entornos productor-consumidor
- No es útil en SD:
  - no hay nombre ➔ no hay enlace
  - no hay posibilidad de envío de mensajes discretos

Los *sockets*. 2

- A partir de *BSD 4.2*, IPC implementada como:
  - llamadas al sistema: capa por encima de *TCP-UDP*
- *socket* = destino de mensajes
  - a través del que se pueden enviar mensajes, y
  - por medio del cual se pueden recibir mensajes
- La IPC tiene lugar entre 2 *sockets*
- Cada *socket* debe estar asociado a:
  - un puerto local de su máquina
  - una dirección IP de dicha máquina
  - un protocolo (*UDP* o *TCP*)

Los *sockets*. 3



- Sólo el proceso que posee el socket puede recibir mensajes destinados al puerto asociado
- En Java:
  - clase *InetAddress*
  - nombramiento mediante *DNS* ■

La comunicación mediante datagramas *UDP*. 1

- Datagrama:
  - mensaje autocontenido no fiable desde un emisor a un receptor
  - único, sin asentimientos ni reenvíos
    - no hay garantía de la entrega
  - transmisión entre 2 procesos ⇔ *send* + *receive*
  - cada proceso debe crear un socket
    - y enlazarlo a un puerto local
      - los servidores, a un puerto de servicio determinado
      - los clientes, a cualquier puerto local libre
  - la operación *receive* entrega:
    - el mensaje transmitido
    - el puerto al que está enlazado el *socket* emisor

La comunicación mediante datagramas *UDP*. 2

- Se usa comunicación asíncrona con *receive* bloqueante:
  - el *send* retorna tras pasar el mensaje a las capas inferiores (UDP/IP)
  - éstas lo transmiten y lo dejan en la cola del *socket* asociado al puerto de destino
  - el *receive*
    - (pendiente o futuro) extrae el mensaje de la cola
    - por defecto, bloqueo indefinido; si se necesita:
      - multitarea ⇒ crear procesos y/o hilos
      - espera acotada ⇒ temporización
  - por defecto, se puede enviar a (y recibir de) cualquier puerto
    - es posible limitarlo a uno concreto: conexión (👁)

La comunicación mediante datagramas *UDP*. 3

- 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
- Añadiendo asentimientos, se puede dar un servicio fiable sobre uno no fiable
- ¿Por qué no usar una comunicación perfectamente fiable?
  - no suele ser imprescindible
  - causa cargas administrativas grandes:
    - almacena información de estado en origen y destino
    - transmite mensajes adicionales
    - posible latencia para emisor o receptor

La API *Java* para datagramas *UDP*. 1

- *Java* proporciona 2 clases:
  - *DatagramPacket* y
  - *DatagramSocket*
- *DatagramPacket*:
  - soporte a los datagramas
  - el constructor que usan los emisores toma:
    - un mensaje, su longitud, la dirección IP de la máquina destinataria y el número de puerto local del socket destinatario
  - el constructor que usan los receptores toma:
    - un array de bytes para un mensaje y su longitud
  - métodos adicionales:
    - *getData*, *getLength*, *getAddress*, *getPort*

La API *Java* para datagramas *UDP*. 2

- *DatagramSocket*:
  - soporte a los *sockets*
  - el constructor que usan los servidores toma:
    - el número de puerto local que se quiere asociar
  - el constructor que usan los clientes
    - sin argumentos: elige uno cualquiera que esté libre
  - métodos adicionales:
    - *send* y *receive*: su argumento es un ejemplar de *DatagramPacket*
    - *setSoTimeout*: permite establecer una temporización
    - *connect*: limita al *socket* local a enviar y a recibir mensajes sólo a/de un puerto remoto

Cliente UDP en *Java*

```
import java.net.*;
import java.io.*;

public class ClienteUDP {
  public static void main(String args []) {
    // Los argumentos dan el mensaje y el nombre de la máquina servidora
    try{
      DatagramSocket elSocket = new DatagramSocket();
      byte [] msjSol = args[0].getBytes();
      InetAddress maquina = InetAddress.getByName(args[1]);
      int puerto = 6789;
      DatagramPacket solicitud =
              new DatagramPacket(msjSol, args[0].length(), maquina, puerto);
      elSocket.send(solicitud);
      byte [] msjRes = new byte[1000];
      DatagramPacket respuesta = new DatagramPacket(msjRes, msjRes.length);
      elSocket.receive(respuesta);
      System.out.println("Respuesta: " + new String(respuesta.getData()));
      elSocket.close( );
    } catch(UnknownHostException e) {
      System.out.println("Desconocido: " + e.getMessage());
    } catch(SocketException e) {
      System.out.println("Socket: " + e.getMessage());
    } catch(IOException e) { System.out.println("E/S: " + e.getMessage()); }
  }
}
```

Servidor UDP en *Java*

```
import java.net.*;
import java.io.*;

public class ServidorUDP {
  public static void main(String args []) {
    try {
      DatagramSocket elSocket = new DatagramSocket(6789);
      byte [] msjSol = new byte[1000] ;
      while (true) {
        DatagramPacket solicitud = new DatagramPacket(msjSol,
                                                      msjSol.length);
        elSocket.receive(solicitud);
        DatagramPacket respuesta = new DatagramPacket(solicitud.getData(),
                          solicitud.getLength(), solicitud.getAddress(),
                          solicitud.getPort());
        elSocket.send(respuesta);
      }
    } catch(SocketException e) {
      System.out.println("Socket: " + e.getMessage());
    } catch(IOException e) {
      System.out.println("E/S: " + e.getMessage());
    }
  }
}
```

La comunicación mediante cauces *TCP*. 1

- La abstracción de cauce (*stream*) oculta:
  - tamaño de los mensajes
    - los procesos leen o escriben cuanto quieren
    - las capas inferiores (TCP/IP) empaquetan
  - mensajes perdidos
    - mediante asentimientos y reenvíos
  - control de flujo
    - evita desbordamiento del receptor
  - mensajes duplicados y/o desordenados
    - mediante identificadores de mensajes
  - destinatarios de los mensajes
    - una vez establecida una conexión, los procesos leen y escriben del cauce sin necesidad de volver a usar sus respectivas direcciones

La comunicación mediante cauces *TCP*. 2

- Dos papeles diferenciados:
  - cliente, crea un *socket* encauzado y:
    - solicita el establecimiento de una conexión (*connect*)
  - servidor, crea un *socket* de escucha
    - con una cola de peticiones de conexión
    - lo asocia a un número de puerto determinado
    - espera la llegada de peticiones de conexión
- Cuando el servidor acepta una conexión:
  - se crea automáticamente un nuevo *socket* encauzado
    - conectado al del cliente por un par de cauces (streams):
      - cada proceso lee de su cauce de entrada y
      - escribe en su cauce de salida
  - si un proceso cierra su *socket*
    - se transmiten a la cola del destino los datos pendientes e
    - indicación de cauce roto

La comunicación mediante cauces *TCP*. 3

- Bloqueo:
  - lectura: si no hay datos disponibles
  - escritura: si la cola del *socket* de destino está llena
- Opciones para atender a múltiples clientes:
  - escucha selectiva (en UNIX, *select*)
  - multitarea:
    - se crea un nuevo proceso que atiende la conexión establecida
    - el proceso original sigue atendiendo el *socket* de escucha
    - multiproceso (en UNIX, *fork*)
    - hilos (*threads*)
- Los procesos deben ocuparse de la concordancia de los datos
- Si errores graves de red $\Rightarrow$ conexión rota

La API *Java* para cauces TCP. 1

- *Java* proporciona 2 clases:
  - *ServerSocket* y
  - *Socket*
- *ServerSocket*:
  - soporte a los *sockets* de escucha (servidores)
  - método *accept*:
    - si cola de solicitudes de conexión vacía, se bloquea
    - si no,
      - toma una solicitud
      - crea un ejemplar de la clase *Socket*
      - establece la conexión
        - » con sus dos cauces
      - retorna una referencia al *Socket* creado

La API *Java* para cauces TCP. 2

- *Socket*:
  - soporte a los *sockets* encauzados
  - el cliente usa un constructor que:
    - toma como argumentos el nombre del ordenador y el número de puerto del servidor
    - crea el *socket* y
    - solicita automáticamente la conexión
  - servidor: resultado del *accept*
  - métodos: *getInputStream* y *getOutputStream*
    - retornan valores de tipo *InputStream* y *OutputStream*
    - se pueden usar como argumentos para constructores de cauces de E/S
      - ej: *DataInputStream* y *DataOutputStream*

Cliente TCP en *Java*

```
import java.net.*;
import java.io.*;

public class ClienteTCP {
  public static void main(String args []) {
    // Los argumentos dan el mensaje y el nombre de la máquina servidora
    try {
      int puerto = 5678;
      Socket elSocket = new Socket(args[1], puerto);
      DataInputStream in = new DataInputStream(elSocket.getInputStream());
      DataOutputStream out =
                        new DataOutputStream(elSocket.getOutputStream());
      out.writeUTF(args[0] );
      String datos = in.readUTF( );
      System.out.println("Respuesta: " + datos);
      elSocket.close();
    } catch(UnknownHostException e) {
      System.out.println("Desconocido: " + e.getMessage());
    } catch(EOFException e) {
      System.out.println("EOF: " + e.getMessage());
    } catch(IOException e) {
      System.out.println("E/S: " + e.getMessage());
    }
  }
}
```

Servidor TCP en *Java*

```
import java.net.*;
import java.io.*;

public class ServidorTCP {
  public static void main(String args []) {
    try {
      int puerto = 5678;
      ServerSocket socketEscucha = new ServerSocket(puerto);
      while (true) {
        Socket socketCliente = socketEscucha.accept( );
        Conexion c = new Conexion(socketCliente);
      }
    } catch(IOException e) {
      System.out.println("Escucha: " + e.getMessage( ));
    }
  }
}

//... sigue
```

Servidor TCP en *Java* (cont.)

```
class Conexion extends Thread {
  DataInputStream in;
  DataOutputStream out;
  Socket socketCliente;
  public Conexion (Socket elSocketCliente) {
    try {
      socketCliente = elSocketCliente;
      in = new DataInputStream(socketCliente.getInputStream());
      out = new DataOutputStream(socketCliente.getOutputStream());
      this.start( );
    } catch(IOException e) {
      System.out.println("Conexion: " + e.getMessage( ));
    }
  }
  public void run () {    // servidor de "eco" (1 sola vez)
    try {
      String datos = in.readUTF();
      out.writeUTF(datos);
      socketCliente.close();
    } catch(EOFException e) {
      System.out.println("EOF: " + e.getMessage());
    } catch(IOException e) {
      System.out.println("E/S: " + e.getMessage());
    }
  }
}
```

La API *C* de *sockets* en UNIX

Creación de los *sockets*

- Antes de comunicarse, cada proceso debe crear, explícitamente, un *socket*
- Se crean mediante la llamada al sistema *socket*
- Hay que especificar:
  - el protocolo: *TCP* o *UDP*
  - el dominio: *UNIX* o *Internet*
    - determina el formato del identificador del *socket*
- Devuelve: el descriptor del *socket*
  - para usar en las llamadas al sistema posteriores

Nombramiento de los *sockets*

- Hay que darles un identificador que los demás procesos puedan usar como dirección de destino
- Mediante la llamada al sistema *bind*
  - asocia descriptor ↔ identificador del *socket*
  - 👁 no confundir *bind* y el *binder* (capítulo 3)

bind	***binder***
Llamada al sistema Al núcleo local {descriptor} ↔ {identificador} socket	Operación de servicio De nivel de red {identificador socket} ↔ {interfaz IR}

Comunicación por *datagrama*

- Primitiva de envío de mensajes: *sendto*
  - Además del mensaje, hay que especificar:
    - el descriptor del *socket* propio
    - la dirección (identificador) del *socket* de destino
- Primitiva de recepción de mensajes: *recvfrom*
  - Además del descriptor del *socket* propio, especifica:
    - el *buffer* para el mensaje y el remitente
- La comunicación tiene lugar ⇔ se emparejan:
  - un *sendto* hacia la dirección de un *socket*
  - un *recvfrom* sobre el *socket* asociado a esa dirección

Comunicación por *datagrama*

- Comunicación cliente-servidor:
  - El servidor crea un *socket*, le asocia una dirección, y
    - publica dicha dirección (¿cómo?)
  - El cliente crea un *socket*, le asocia una dirección, y
    - obtiene la dirección del *socket* del servidor (¿cómo?)
  - Mensaje de solicitud:
    - el cliente lo envía con *sendto* y
    - el servidor lo recoge con *recvfrom*
  - Mensaje de respuesta:
    - se invierten los papeles

Comunicación por *datagrama*

Cliente

```
dsc = socket (AF_INET,
              SOCK_DGRAM,
              PF_UNSPEC);
bind (dsc, &idCliente, longCli);
/* obtiene idServ */
sendto (dsc, solicitud, longSol,
        idServ, long);
n = recvfrom (dsc, &buffer, longBuf,
              flags, &idRemite, &long);
    :
    : (se suspende)
    :
/* continúa su ejecución */
```

Servidor

```
dss = socket (AF_INET,
              SOCK_DGRAM,
              PF_UNSPEC);
bind (dss, &idServ, longServ);
/* publica idServ */
n = recvfrom (dss, &buffer, longBuf,
              flags, &idRemite, &long);
    :  (se suspende)
/* ejecuta lo solicitado */
sendto (dss, respuesta, longResp,
        idRemite, long);
```


Comunicación encauzada (*stream*)

- 1º hay que crear e identificar el par de *sockets*
  - igual que en datagrama
- El servidor "escucha" llamadas: *listen*
- El cliente solicita la conexión: *connect*
- El servidor acepta la conexión: *accept*
- Conexión ⇔ se emparejan 1 *connect* y 1 *accept*
  - el SO crea 1 nuevo *socket* y lo conecta al del cliente
  - el servidor puede seguir escuchando en el socket original

Comunicación encauzada (*stream*)

- La comunicación entre el par de *sockets* conectados se hace por medio de las llamadas:
  - *write* ↔ *read*, o
  - *send* ↔ *recv*
- La conexión se cierra al cerrar un *socket*:
  - llamada *close*
- La conexión implica:
  - actividad adicional de proceso y comunicación

Comunicación encauzada (*stream*)

Cliente

```
dsc = socket (AF_INET,
              SOCK_STREAM,
              PF_UNSPEC);
bind (dsc, &idCliente, longCli);
/* obtiene idServ */
connect (dsc, idServ, longServ);
    :
    :  (se suspende)
    :
 n = write (dsc, buffer, longBuf);
```

Servidor

```
dss = socket (AF_INET,
              SOCK_STREAM,
              PF_UNSPEC);
bind (dss, &idServ, longServ);
/* publica idServ */
listen (dss, 5);
    :  (se suspende)
dsn = accept (dss, &remite, &long);
    :
n = read (dsn, &buffer, longBuf);
    :  (se suspende)
/* continúa */
```

La Alineación y la Representación Externa de Datos

Correspondencia datos ↔ mensajes

- Datos de los programas: estructurados
- Información de los mensajes: secuencial
- Luego, hay que aplanar antes de transmitir y reconstruir tras recibir
- Diferentes representaciones de los datos, luego:
  - común acuerdo sobre forma externa
  - si ordenadores del mismo tipo, no es necesaria
    - en *stream*, se puede negociar
  - pasar datos nativos + identificador de arquitectura

La representación externa de datos

- Hay normas que definen la representación de:
  - los tipos simples más comunes
  - algunos tipos estructurados
- Ejemplos:
  - *Sun XDR*: para los mensajes de *Sun RPC*
  - *CORBA CDR*: para los mensajes de *CORBA*
  - *Java OS*: para los mensajes de *Java RMI*

La alineación. 1

- Alinear (*marshall*):
  - ensamblar una colección de datos de forma que se puedan transmitir en un mensaje
- Desalinear (*unmarshall*):
  - desensamblarlos en destino para producir una colección equivalente a la original
- En ambos:
  - aplanado + representación externa

La alineación. 2

- Se podría hacer "a mano"
- Ej. en C:

```
char *nombre="Pérez", localidad="Madrid"; int anho=1934;
sprintf (mensaje, "%d %s %d %s %d",
        strlen(nombre),nombre,strlen(localidad),localidad,anho);
```

- Se puede automatizar el proceso:
  - Por parte de alguna capa de la lógica de mediación
    - de forma transparente al programador
  - A partir de la especificación de los tipos de datos
  - Exige definir en una notación adecuada:
    - los tipos de las estructuras de datos, y
    - los tipos de los datos básicos

La alineación. 3

- Las especificaciones de tipos acompañan a programas
- Un fuente con especificaciones de tipos:
  - primero se pre-procesa
  - esto hace que se inserten las operaciones
    - de alineación para los mensajes salientes
    - de desalineación para los entrantes
  - los procesos implicados deben usar las mismas definiciones de tipos en los mensajes intercambiados

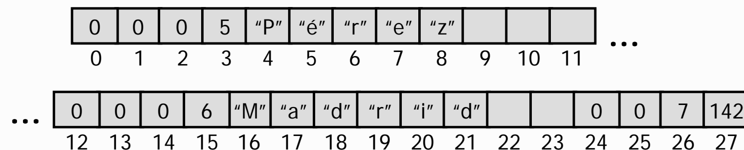
Ejemplo: *CORBA CDR*

- 1 cadena = 1 unsigned long + 1 byte por cada carácter
- Cada entero empieza en un índice múltiplo de su tamaño

Para la estructura:

```
struct Persona {
    string nombre;
    string localidad;
    long anho;
}
```

Con valores:
{"Pérez", "Madrid", 1934}



0	0	0	5	"P"	"é"	"r"	"e"	"z"				...
0	1	2	3	4	5	6	7	8	9	10	11	

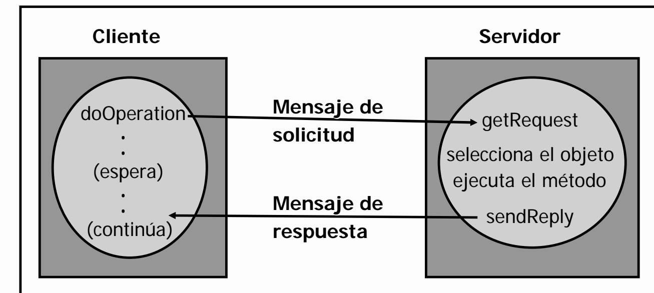
...	0	0	0	6	"M"	"a"	"d"	"r"	"i"	"d"			0	0	7	142
	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

El módulo de comunicaciones cliente-servidor

Protocolos solicitud-respuesta

- La comunicación cliente-servidor es síncrona
  - el cliente se bloquea hasta recibir una respuesta
- *send-receive*: 4 operaciones para un intercambio
  - 4 llamadas al sistema
- Protocolos de *Amoeba*, *Mach*, *Chorus* y *V*:
  - soporte más directo de este patrón de comunicación
  - 3 primitivas de comunicación:
    - *doOperation*, *getRequest* y *sendReply*
    - menores cargas administrativas
      - sólo 3 llamadas al sistema en cada intercambio

Protocolos solicitud-respuesta



- La respuesta se usa como asentimiento de la solicitud
- Puede dar ciertas garantías de entrega

Protocolos solicitud-respuesta

- Mismo tipo de datos para cualesquiera mensajes intercambiados:

tipoMensaje	*(Solicitud, Respuesta)*
idInvocacion	*int*
refObjeto	*// si IMR*
idMetodo	*int*
argumentos	*// array de bytes*

  - El cliente genera una *idInvocacion* única
    - la añade al mensaje de solicitud
    - el servidor la copia a su respuesta
    - el cliente puede comprobar que es la esperada
  - *idMetodo* identifica a la operación del servidor pedida por el cliente

Identificadores de las invocaciones

- Es necesario que sean únicos
- Tienen 2 partes:
  - *idInvocacion*: entero secuencial elegido por el emisor
    - provee: unicidad entre las invocaciones de un mismo emisor
    - exige: no pronta reutilización
  - un identificador del remitente
    - p.ej. la dirección IP y el número del puerto en el que espera la respuesta
    - provee: unicidad global

Fallos en la entrega

- Consideraciones:
  - los mensajes se pueden perder
  - una parte de la red puede quedar aislada (rotura)
  - los procesos pueden fallar
    - y esto es indistinguible de un fallo de comunicaciones
    - tras *N* reintentos ⇒ proceso no disponible
      - dificultad de elegir un *N* adecuado
  - no corrupción de datos: si se reciben, son correctos
- Se incorpora una temporización al *receive* del cliente
  - problema: ¿qué hacer cuando vence?

Temporizaciones

- Tras vencer el tiempo, el módulo de comunicaciones puede:
  - retornar inmediatamente, con indicación de fallo
    - no se suele hacer: puede haberse perdido la respuesta
    - en ese caso, la operación sí se habría realizado
  - reintentarlo repetidamente, hasta
    - obtener una respuesta, o
    - tener una "duda razonable" sobre el servidor
      - en este caso, informar al cliente de que el servidor ha fallado
      - ojo: no certeza, sólo probabilidad

Solicitudes duplicadas

- El servidor puede recibir solicitudes duplicadas
  - los reintentos pueden llegar antes de tiempo:
    - si servidor más lento de lo esperado o sobrecargado
- Esto puede causar ejecuciones repetidas
- Solución:
  - reconocer los duplicados del mismo cliente:
    - tienen la misma *idInvocacion*
  - y filtrarlos (olvidarse de ellos)
    - cuando termine con la primera solicitud, ya contestará
    - ¿y si el reintento se debe a que se perdió la respuesta?

Respuestas perdidas

- Causan que el servidor repita una operación
- Sólo no es problema si el servidor es tal que
  - todas sus operaciones son idempotentes
- Operación idempotente es la que:
  - se puede realizar de forma repetida
  - y los resultados son los mismos que si se ejecutase una sola vez
- ¿Qué hacer si hay operaciones no idempotentes?

Historial

- Se trata de poder retransmitir una respuesta
  - sin volver a ejecutar la operación
- Historial = registro de respuestas enviadas
  - mensaje (con su *idInvocacion*) y su destinatario
  - Muy costoso en términos de memoria
    - si vemos una solicitud nueva como el asentimiento de la anterior respuesta: sólo hay que guardar la última respuesta enviada a cada cliente, pero:
      - un servidor tiene muchos clientes
      - un cliente puede no tener nuevas solicitudes por largo tiempo
    - solución "provisional": descartarlas tras algún tiempo

Protocolos de intercambio de IR. 1

- Hay 3 protocolos que se suelen emplear:
  - **R** (*request*)
  - **RR** (*request-reply*)
  - **RRA** (*request-reply-acknowledge reply*)

Nombre	*Mensajes enviados por*		
	Cliente	*Servidor*	*Cliente*
R	*Solicitud*		
RR	*Solicitud*	*Respuesta*	
RRA	*Solicitud*	*Respuesta*	*Asentimiento de respuesta*

- Pueden ofrecer diferentes semánticas ante los fallos

Protocolos de intercambio de IR. 2

- Protocolo **R**:
  - útil sólo en casos como los *Servidores de Ventanas*:
    - no hay valor de retorno del procedimiento
    - el cliente no necesita confirmación
  - el cliente continúa tras enviar la solicitud
- Protocolo **RR**:
  - habitual en entornos cliente-servidor
  - la respuesta asiente la solicitud
  - una posterior solicitud del mismo cliente asiente la respuesta

Protocolos de intercambio de IR. 3

- Protocolo **RRA**:
  - la respuesta asiente la solicitud
  - el asentimiento de respuesta:
    - lleva la *idInvocacion* de la respuesta a la que se refiere
    - asiente dicha respuesta y las de *idInvocacion* anterior
      - si se pierde uno, no es grave
    - permite vaciar entradas del historial
  - el envío del asentimiento de respuesta no bloquea al cliente
  - pero consume recursos de procesador y de red