Sistemas Distribuidos -Comunicación entre Procesos (IPC)

M.C. Fernando Pech May

Instituto Tecnológico Superior de los Ríos
auxtecomp@gmail.com

Materia: Sistemas Distribuidos

Curso de verano, 2011

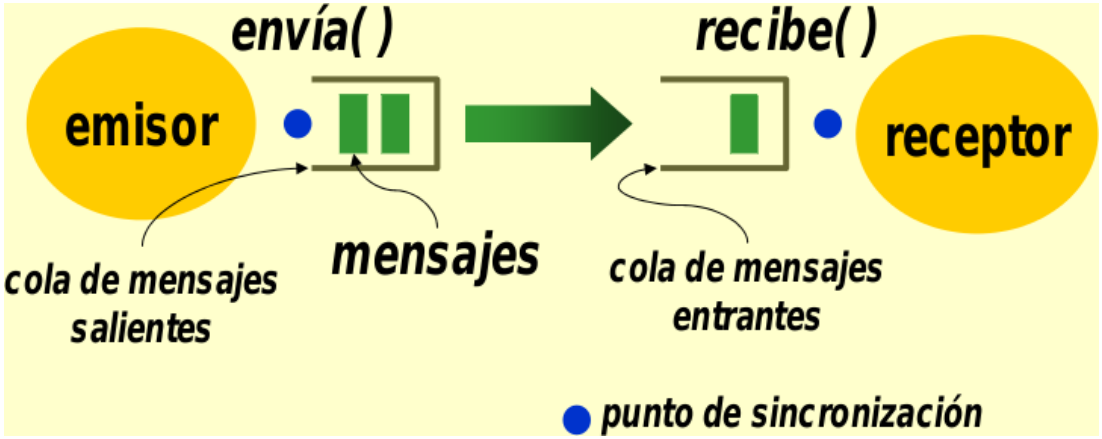
Tabla de contenido

Los sistemas distribuidos se basan en el **intercambio de mensajes** y **la sincronización** entre procesos distribuidos autónomos

- Comunicación entre procesos (IPC)
  - Variables compartidas
  - **Paso de mensajes**
- El paso de mensajes es a través de lenguajes de programación concurrentes
  - Extensiones de lenguajes
  - Llamadas de APIs

Principios de IPC
(ver en [Andrews and Schneider 83] G. Andrews and F. Schneider, Concepts and Notations for Concurrent Programming, ACM Computing Surveys, Vol. 15, No. 1, March 1983)

- **Programas concurrentes**: colección de dos o mas programas secuenciales que se ejecutan concurrentemente (al mismo tiempo)
- **Procesos concurrentes**: colección de dos o mas programas secuenciales **en operación** y se ejecutan concurrentemente
- **Sincronización**:
  - procesos que se ejecutan simultáneamente en diferentes equipos a diferentes velocidades
- Paso de mensajes primitivos
  - *send expression_list to destination_designator*
    - evalúa *expression_list*
    - agrega una instancia de un nuevo mensaje al canal *destination_designator*
  - receive variable_list from source_designator

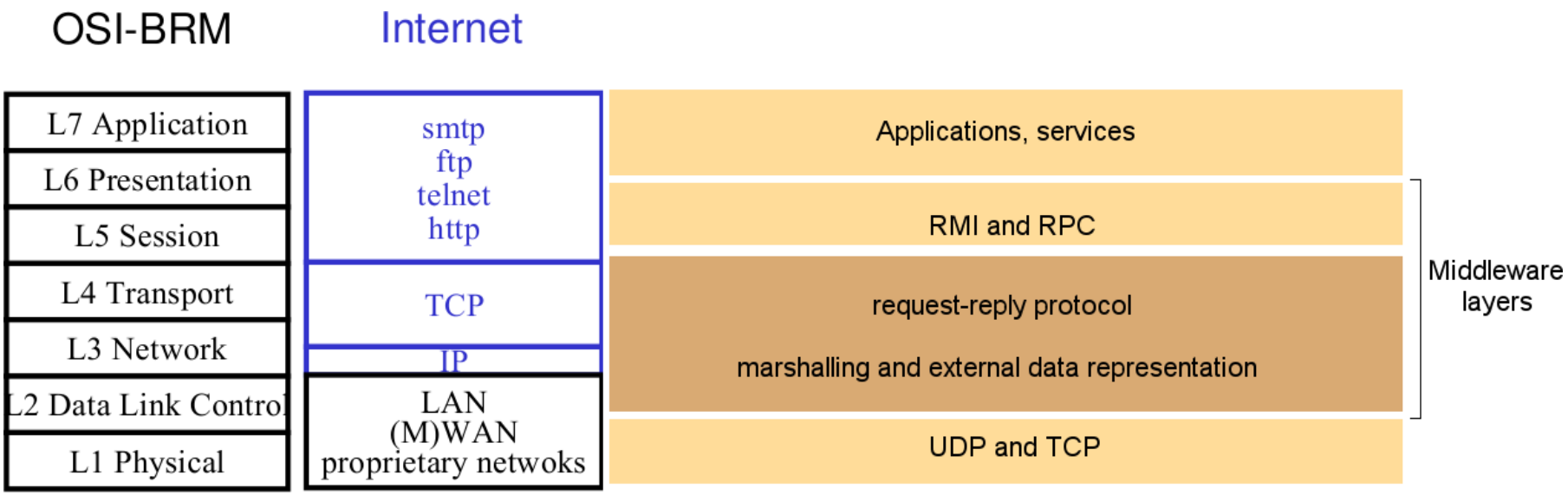


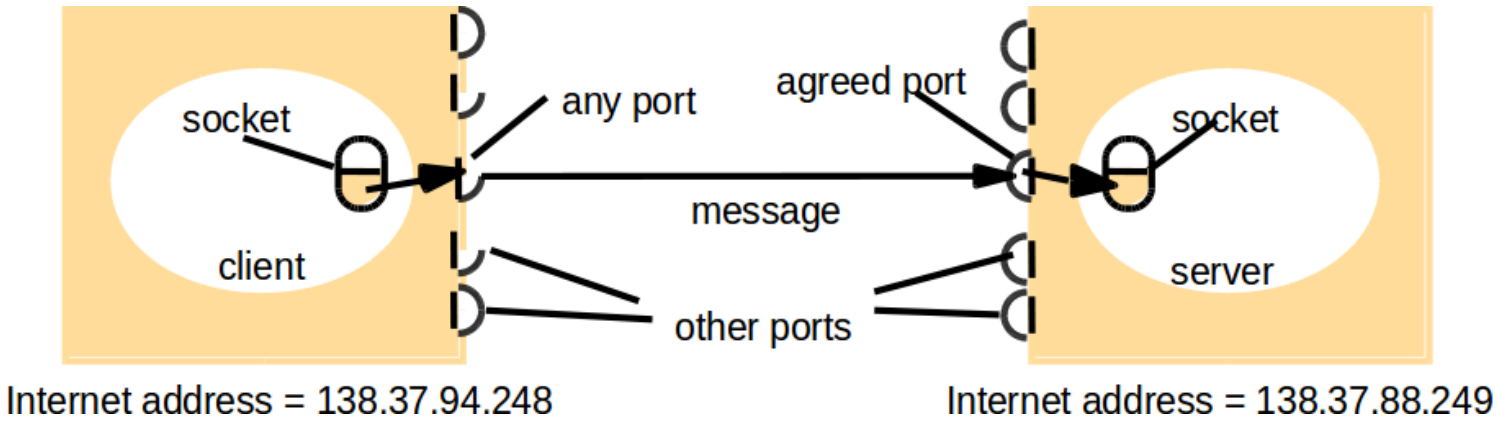
- Comunicación implica
  - comunicación por parte de los interlocutores y
  - sincronización
- Síncrona: cada operación se completan cuando se completa el par **envía() - recibe()**
- Asíncrona: pueden completarse por separado

tipo	*envía ()*	*recibe()*
síncrona	bloqueante	bloqueante
asíncrona	no bloqueante	bloqueante
asíncrona	no bloqueante	no bloqueante

- recibe( ) no bloqueante requiere notificación por interrupción o evento, o encuesta
  - es más complejo de programar.
- recibe( ) bloqueante es fácil de programar en entonos con varios hilos.
- Identificador de comunicación
  - dependiente de ubicación (p.ej.: puertos UNIX BSD)
    - debe ser bien conocido
    - un proceso puede tener varios libros
  - por nombre
    - requiere un servicio de directorio (binder)
    - permite la re-ubicación pero no la migración.
  - independiente de ubicación

- Un servicio de mensajes es fiable si se garantiza la entrega
  - aunque se pierda cierto número de ellos, que habrá que recuperar.
  - si no se recuperan: no fiable
- Integridad: si no se corrompen los mensajes y no se duplican
- El orden de entrega debe reproducir el orden de envío


OSI-BRM
Internet
L7 Application
L6 Presentation
L5 Session
L4 Transport
L3 Network
L2 Data Link Control
L1 Physical
smtp
ftp
telnet
http
TCP
IP
LAN
(M)WAN
proprietary netwoks
Applications, services
RMI and RPC
request-reply protocol
marshalling and external data representation
UDP and TCP
Middleware layers


socket
client
any port
agreed port
message
other ports
socket
server
Internet address = 138.37.94.248
Internet address = 138.37.88.249

- IPC de internet son mecanismos de Unix y otros sistemas operativos (BSD unix, Solaris, Linux, Windows NT, Macintosh OS)
- Los procesos en estos SO pueden **enviar** y **recibir** mensajes a través de los sockets (dúplex)
- socket=conector
  - ligado a un comi-d **direcc_internet:puerto**
  - y a un protocolo (TCP o UDP)
    - El socket es un elemento del proceso
    - El puerto es un elemento del núcleo del S.O.
  - 2^{16} puertos posibles (algunos reservados)
  - no se puede reabrir un puerto ya asignado a otro proceso

API java para direcciones

java.net

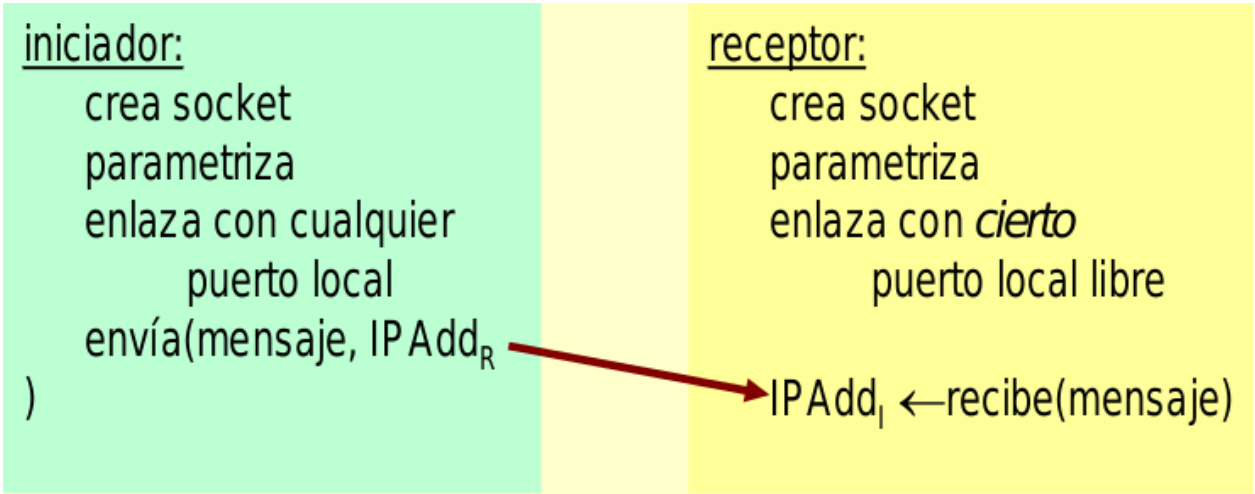
```
public final class InetAddress {
  boolean equals(Object obj); String toString( ) ;
  object byte[ ] getAddress( ); // IPAddr en crudo
  static InetAddress[ ] getAllByName(String host);
    // lanza: UnknownHostException y SecurityException
  static InetAddress getByName(String host);
    // lanza: UnknownHostException
  String getHostAddress( ); // IPAddr en forma DDN*
  String getHostName( ); // lanza: SecurityException
  static InetAddress getLocalHost( );
    // lanza: UnknownHostException y SecurityException
  int hashCode( );  // una clave hash a partir del IPAddr
  boolean isMulticastAddress( ) ; // ¿Es IPAddr Multi...
}
```

IPC basados en datagramas UDP

- **Propiedades del datagrama UDP**: no asegura el orden de preservación, pérdida y duplicación de mensajes
- etapas necesarias:
  - **crear** un socket
  - **une** el socket a un puerto y a una dirección local de internet
    - cliente: puerto libre arbitrario
    - servidor: puerto del servidor
- método de recepción: retorna la dirección de internet y el puerto del emisor, además del mensaje
- Tamaño de mensajes:Todas las IP pueden enviar mensajes de 2^{16} bytes (algunos se restringen a 8KB)

Comunicación con UDP

- Bloqueo
  - envía() no bloqueante,
  - recibe() bloqueante (posibilidad de indicar un timeout)
- Identidad del emisor
  - El socket de recepción suele estr abierto a cualquier emisor
  - es posible vincular el receptor a una sola IPAddr remota



Comunicación con UDP

- Modelo de fallo
  - Fallos por omisión
    - en el canal (que incluye los del emisor y del receptor)
    - provocados por desbordamiento de búfer, prdida de mensajes, o corrupción.
    - Para detectar la corrupción, se puede añadir un "checksum"
  - Fallos de ordenación en la llegada
- Utilización de UDP, cuando...
  - no es preciso almacenar información de estado en origen ni en destino
  - es preciso reducir el intercambio de mensajes
  - el emisor no se bloquea

Java API para datagrama UDP

Clases

- **DatagramPacket**: Constructor de generación de mensajes para ser enviado en un arreglo de bytes
  - contenido del mensaje(byte array) **getData()**
  - longitud del mensaje **getPort()**
  - dirección de internet y número de puerto (destino) **getAdress()**

Java API para datagrama UDP

Clases

- **DatagramSocket**: clase para el envío y recepción de datagramas UDP
  - un constructor con el número de puerto como argumento
  - constructor sin argumentos para utilizar el puerto local libre
  - métodos
    - **send** y **receive**

      send(DatagramPacket dP) throws IOException

      receive(DatagramPacket dP) throws IOException //dP vacío
    - **SetSoTimeout**

      setSoTimeout ( ) throws InterruptedIOException
    - **connect**: para conectar un socket a una dirección de internet remota y el puerto

      connect ( ) para conectarse a una sola dirección

Java AIP para datagrama UDP

Ejemplo

```
import java.net.*;
import java.io.*;
public class UDPClient{
  public static void main(String args[]){
     // args give message contents and destination hostname
    try {
        DatagramSocket aSocket = new DatagramSocket();
         // create socket
        byte [] m = args[0].getBytes();
        InetAddress aHost = InetAddress.getByName(args[1]); // DNS lookup
        int serverPort = 6789;
        DatagramPacket request =
            new DatagramPacket(m, args[0].length(), aHost, serverPort);
        aSocket.send(request);  //send nessage
        byte[] buffer = new byte[1000];
        DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
        aSocket.receive(reply);  //wait for reply
        System.out.println("Reply: " + new String(reply.getData()));
        aSocket.close();
    }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
    }catch (IOException e){System.out.println("IO: " + e.getMessage());}
    } // can be caused by send
```

IPC basado en streams TCP

- Crea un canal virtual de comunicación sobre streams
- Oculta las siguientes características:
  - Tamaño de los mensajes: se parte el mensaje y se reconstruye en destino
  - Mensajes perdidos
  - Control de flujo: ajusta velocidades bloqueando el emisor si el receptor no recupera los mensajes
  - Duplicación y ordenación
  - Destinos de los mensajes, una vez realizada la conexión.
- Aspectos importantes
  - Concordancia de tipos de datos
    Los procesos deben conocer el tipo de datos que se envían - reciben
  - Bloque
    El receptor se bloquea siempre, y el emisor sólo cuando el canal no puede admitir más mensajes

IPC basado en streams TCP

- Modelo de fallo
  - TCP usa: números de secuencia, checksums y timeouts.
  - Cuando el número de errores es excesivo o se sobrepasa el tiempo límite se declara rota la conexión.
  - no se distingue un fallo en el proceso del fallo en la conexión.
  - no se asegura la recepción en caso de error.
- TCP es la base de: HTTP, FTP, SMTP, Telnet
  (el cliente de telnet se puede usar para conectarse con cualquier servidor)

IPC basado en streams TCP

Clases

- ServerSocket: crear el socket en el lado del servidor para escuchar las solicitudes de conexión
- Socket: para procesos con conexión
  - constructor para crear un socket y conectarse a un host y puerto remoto de un servidor
    Socket accept ( )
    Socket throws UnknownHostException
    throws IOException
  - método para acceder a flujo de entrada y salida
    InputStream getInputStream ( )
    OutputStream getOutputStream ( )

IPC basado en streams TCP

Ejemplo

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
      try{
         int serverPort = 7896; // the server port
         ServerSocket listenSocket = new ServerSocket(serverPort);
                        // new server port generated
         while(true) {
            Socket clientSocket = listenSocket.accept();
                           // listen for new connection
            Connection c = new Connection(clientSocket);
                           // launch new thread
         }
   } catch(IOException e) {System.out.println("Listen socket:"+e.getMessage());}
   }
}
```

IPC basado en streams TCP

Ejemplo

```
class Connection extends Thread {
   DataInputStream in;
   DataOutputStream out;
   Socket clientSocket;
   public Connection (Socket aClientSocket) {
      try {
         clientSocket = aClientSocket;
         in = new DataInputStream( clientSocket.getInputStream());
         out =new DataOutputStream( clientSocket.getOutputStream());
         this.start();
      } catch(IOException e){System.out.println("Connection:"+e.getMessage());}
   }
   public void run(){
      try {     // an echo server
         String data = in.readUTF();
                // read a line of data from the stream
         out.writeUTF(data);
                // write a line to the stream
         clientSocket.close();
      } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
      } catch (IOException e) {System.out.println("readline:"+e.getMessage());}
   }
}
```

Problemas involucrados

1. cada máquina representa los tipos de datos básicos de formas diferentes
   - *little-endian, big-endian*
   - *UNIX-char=1byte,UNICODE=2bytes*
2. los tipos de datos compuestos se organizan de forma diferente según el lenguaje, el compilador y la arquitectura
   - tipos estructurados
   - tipos dinámicos (con referencias)
3. el canal de transmisión solo envá series de bytes

Mecanismos de solución

- Convertir los datos a un formato de $\ll$ *representacin externa de datos* $\gg$ común
  - incluye un aplanado de datos (XML?)
- Enviar los datos en el formato del emisor + indicación de la organización de los datos
  - solo aplanado de datos y envío del tipo de datos.
  - si los procesos son similares se puede evitar enviar el tipo de los datos
- *En cualquier caso hay que eliminar las referencias a memoria*

Mecanismos de solución

- Alternativas usuales
  - **Java RMI**: utiliza un procedimiento de serialización, basado en el conocimiento de las clases.
  - **CORBA**: usa un lenguaje para la representación externa de datos (CDR) y un lenguaje de definición de interfaces (IDL)
  - **Sun RPC**: emplea tambin un lenguaje comn (XDR)
    Variables:
    - XML-RPC
    - SOAP (con XML, para web services)
  - Representación textual de los datos (como en los tipos MIME)
  - OSF: DCE-IDL, Microsoft: DCOM IDL, Xerox: ...

Corba Common Data Representation (CDR)

15 datos primitivos: short, long, float, double, char, boolean, octet, ...

tipos compuestos (any)

Tipo	*Representación*
sequence	longitud (unsigned long-entero largo sin signo-) seguida de los elementos en orden.
string	longitud (unsigned long) seguida de los caracteres en orden (también puede tener caracteres anchos-2bytes-).
array	elementos de la cadena en orden (no se especifica la longitud porque es fija).
struct	en el orden de declaración de los componentes.
enumerated	unsigned long (los valores son especificados por el orden declarado).
union	etiqueta de tipo seguida por el miembro seleccionado.

Ejemplo

Ejemplo de especificación IDL:

- Los procedimientos de marshalling y unmarshalling se generan con el compilador IDL (ej:idltoC, idltojava, ...)

```
struct Persona {
  string nombre;
  string lugar;
  long año;
}
```

Ejemplo

La estructura de forma aplanada es:

Posición en la secuencia de bytes	4 bytes	Notas
0 - 3	5	*Longitud del string*
4 - 7	"Pére"	«Pérez»
8 - 11	"z___"	
12 - 15	6	*Longitud del string*
16 – 19	"Madr"	«Madrid»
20 – 23	"id__"	
24 – 27	1934	*unsigned long*

Serialización en Java

```
public class Persona implements Serializable {
   private String nombre;
   private String lugar;
   private int año;
   public Persona(String unNombre, String unLugar, int unAño) {
      nombre = unNombre;
      lugar = unLugar;
      año = unAño;
      }
      // siguientes métodos de la clase
     }
```

Serialización en Java

- Serialización ≈ marshalling
  - La reflexión de Java permite efectuarla de modo transparente.
  - Es recursiva
  - Los métodos de serialización/deserialización son internos a Java

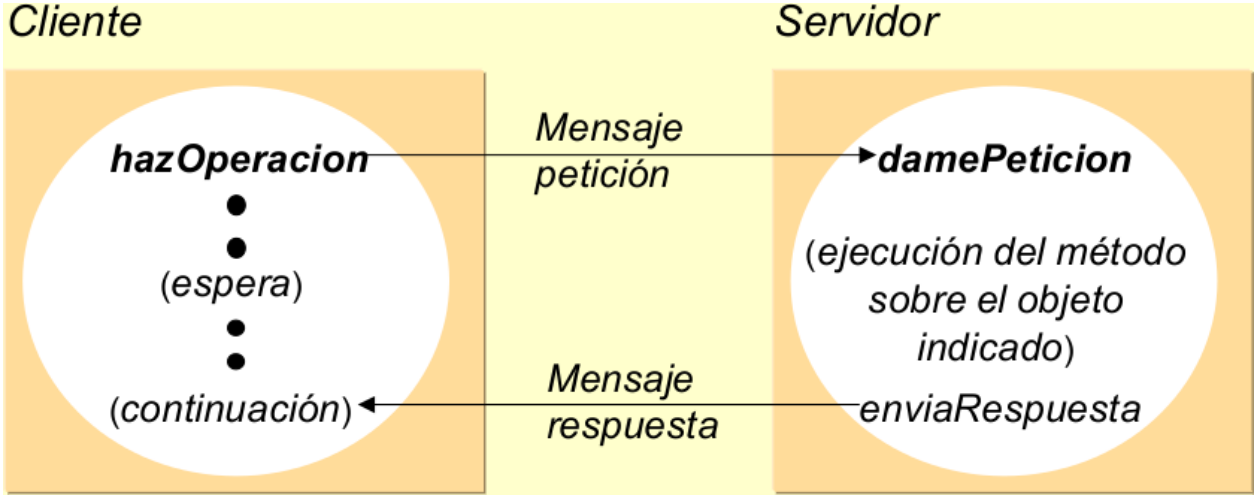
Persona	Número de versión de 8-bytes		a0	*Nombre de la clase, número de versión*
3	int año	java.lang.String nombre:	java.lang.String lugar:	*Número, tipo y nombre de las variables de instancia*
1934	5 Pérez	6 Madrid	a1	*Valores de las variables de instancia*

  - La forma serializada real contiene marcadores de tipo adicionales
  - a0 y a1 son apuntadores

Referencias remotas en Java

32 bits	32 bits	32 bits	32 bits	
dirección Internet	**número de puerto**	**tiempo**	**número de objeto**	**interfaz de objeto remoto**

- Los identificadores a objetos remotos tienen ámbito global y son únicos
- Pueden enviarse como parámetro y recibirse desde un método
- Pueden compararse
- Tiene un ámbito de utilización
  - Se puede garantizar incluyendo periodos de validez



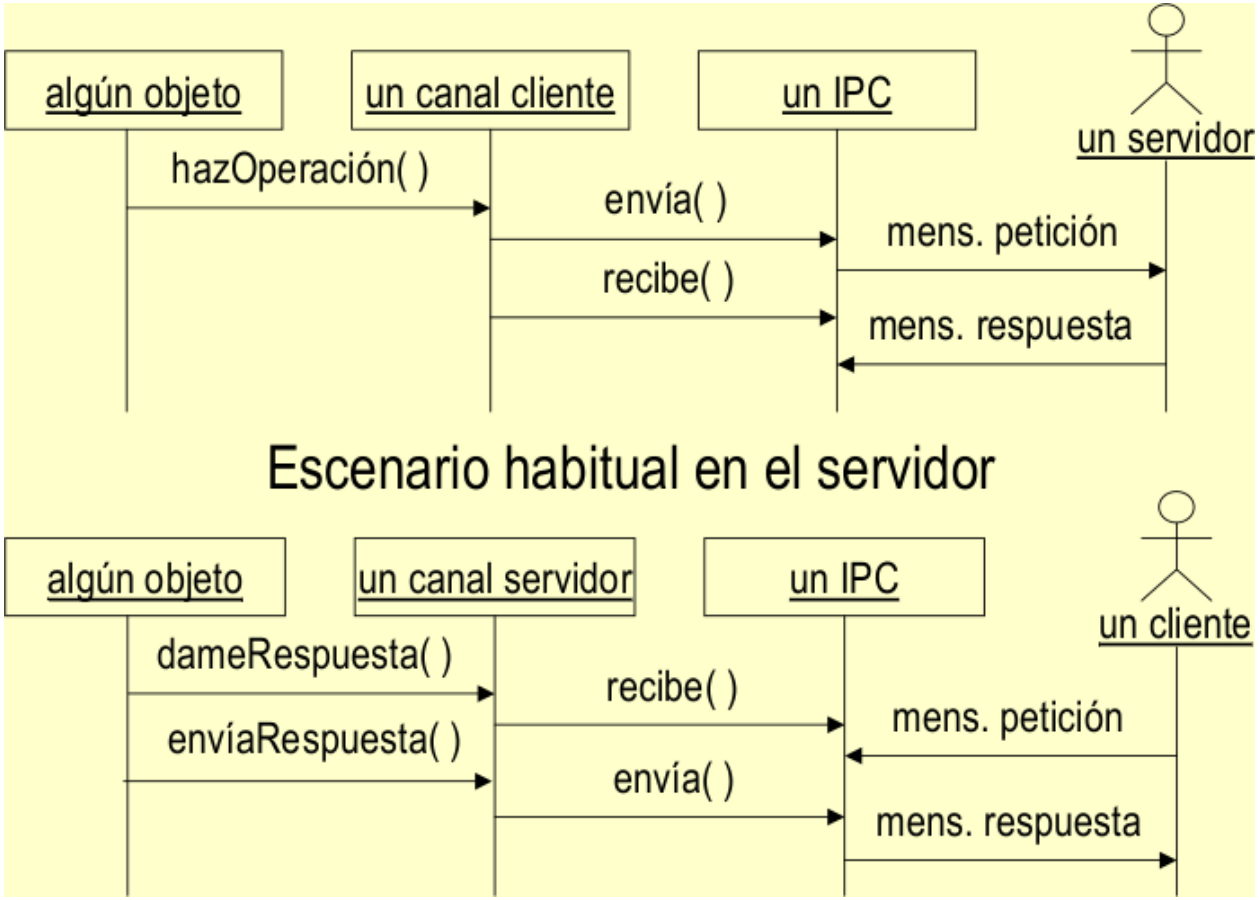
- Protocolo petición-respuesta
  - RPC y RMI se basan en él
- hazOperacion deberá encapsular las garantías de espera especificadas.
- hazOperación puede ser bloqueante o no bloqueante
- damePetición suele ser bloqueante

```
public byte[ ] hazOperacion( RemoteObjectRef objeto,
        int idMetodo, byte [ ] argumentos)
    // Empaqueta los datos;
    // envía idMetodo y argumentos al objeto indicado en obj;
    // se bloquea hasta que recibe la respuesta
public byte[ ] damePeticion( );
    // se bloquea hasta recibir peticiones;
    // desempaqueta los argumentos;
    // invoca el método;
    // empaqueta los resultados
public void enviaRespuesta(byte [ ] respuesta,
    InetAddress hostCliente, int puertoCliente);
    // envía la respuesta al cliente.
```

Estructura de un mensaje petición-respuesta

- tipoMensaje int (0, 1)
- idPeticion int
- referenciaObjeto RemoteObjectRef
- idMetodo int o Method
- argumentos cadena de bytes

operaciones



Escenario habitual en el cliente

Puede implementarse sobre un nivel

- con conexión y control de flujo, o
- sin conexión

En este último caso hay que tener en cuenta ciertas circunstancias:

- Tiempos de espera límite (timeouts)
  hazOperacin puede encapsular reintentos
- Eliminación de peticiones duplicadas
  - a)El servidor puede repetir la operación si es preciso (si la op. es idempotente)
  - El servidor descarta los duplicados y respuesta en función de si se ha perdido la respuesta o no
- Pérdidas de respuestas
  (idem que en el caso anterior)
- Utilizaciónde históricos
  Permite retransmitir la respuesta en base al identificador.

Nombre	*Mensajes enviados por:*		
	Cliente	*Servidor*	*Cliente*
R	Petición		
RR	Petición	Respuesta	
RRA	Petición	Respuesta	ACK

- R: cuando el cliente no requiere respuesta ni confirmación (no bloqueante)
- RR: se toma la respuesta como ACK.
- Una petición posterior puede servir como ACK del cliente
- RRA: ms estricta: cuando se requiere operaciones atmicas, o se requiere vaciar el historial. También, una petición posterior puede servir como ACK

- Es posible mezclar diversos tipos (R, RR, RRA) sobre el mismo servicio
  - En la interfaz del servidor se especifica a qué tipo pertenece la petición.
- Como cada peticin suele incluir el número de petición del cliente, el servidor pueden entender que una llamada RR reconozca un RRA anterior.

La comunicación se produce entre cada proceso y el grupo al que pertenece
Útil en caso de:

- Tolerancia a fallos: servicios replicados
- Aumento de prestaciones con réplicas éste y el anterior comparten el mismo objetivo: disponibilidad.
- Ubicación de objetos en servicios distribuidos
- Actualización múltiple

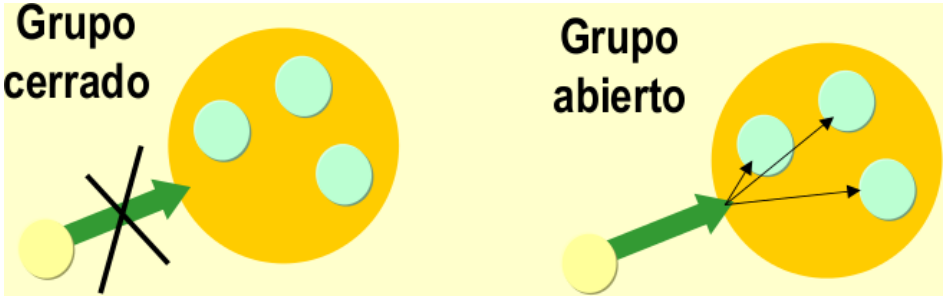
- Los problemas de comunicación unicast + Identificador de comunicación para grupos
- Soporte de comunicación
- Semántica de entrega de mensajes:
  - Relación de pertenencia al grupo
  - Ordenamiento de los mensajes

1. Uno a uno (unicast)
2. Uno a muchos
3. Muchos a uno
4. Muchos a muchos

- Todos ellos utilizan las mismas dos primitivas:
  - envía(destino, mensaje)
  - origen $\leftarrow$ recibe(mensaje)
    donde el identificador de comunicación destino y origen pueden ser el nombre de:
  - un proceso
  - un grupo de procesos

Uno a muchos

- Varios receptores por cada emisor-multidifusión
- Caso especial: emisión general-difusión
- Aplicaciones:
  - Todos los miembros de una comunidad deben recibir la emisión (ej: actualizaciones de información, ...)
    - Caso: Todos los miembros deben responder
    - Caso: No todos deben responder
  - No todos los miembros tienen por qué recibir la emisión (ej: búsqueda de espacio de disco, acceso a información, ...)
    - Caso: No todos los miembros deben responder



- Grupo cerrado: solos los miembros pueden difundir
  ej: comunidad de cálculo
- Grupo abierto: cualquier proceso puede difundir
  ej: grupo de servidores replicados
- Un grupo puede ser mixto y tener operaciones internas y externas.
  Suele implementarse usando dos identificadores de comunicación

Opciones de gestión de la entrega (delivery)

- Mecanismo de multidifusión
- Mediante proceso de comunicación (poco fiable y poco escalable)
- Mediante proceso de comunicación con réplicas (carga en las tareas de consistencia)

Direccionamiento de grupo

- De alto nivel: cadena texto
- De bajo nivel: dependiente del nivel de comunicación

Posibilidades:

1. Mecanismo de creación de direcciones multidifusión
2. Solo broadcast: varios procesos deben compartir su dirección de bajo nivel
3. Solo uno a uno: el nombre de bajo nivel debe contener la lista de elementos.

En a) y b) solo se envía una trama por mensaje
En c) el coste aumenta con el número de nodos (útil cuando los nodos del grupo están muy diseminados)

reparto de mensajes

- El servidor centralizado del grupo tiene tuplas de nombres de grupo (alto y bajo) nivel y lista de PIDs del grupo
  1. El emisor envía el mensaje: contacta con el servidor del grupo y envía el nombre de bajo nivel del grupo y la lista de PIDs
  2. El servidor compone el nombre sumando ambos campos
  3. Se emite el mensaje:
     - A la dirección multicast o broadcast
     - A las direcciones de los nodos
  4. El núcleo del receptor extrae los PID y envía los mensajes a los receptores locales

La multidifusión es asíncrona por naturaleza:

- El emisor no puede esperar a todos los receptores.
- El emisor puede no conocer a todos los receptores
- Multidifusión sin búfer: el mensaje se pierde en el receptor si éste no escucha
- Multidifusión con búfer: cualquier proceso receptor puede recibir el mensaje asíncronamente

- Send to All
  - Se envía una copia del mensaje a cada proceso, y con búfer.
- Bulletin Board
  - Se envía el mensaje a un canal (post). El receptor copia el mensaje del canal cuando puede.
    - Los procesos con permisos para recibe( ) constituyen el grupo de multidifusión.
      El receptor puede recuperar los mensajes según su relevancia, o su propia disponibilidad.
      Se puede fijar un tiempo de caducidad de cada mensaje según la necesidad del emisor.
    - Ej: pool processor flotante.