Comunicación entre procesos

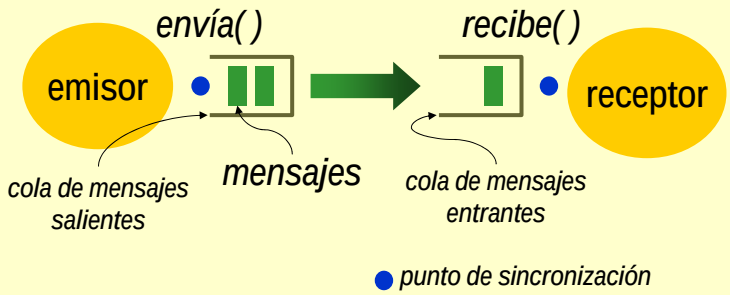
Sistemas Distribuidos – ITInformática (UVA)

César Llamas Bello - Febrero 2003

Indice

- Interfaz de programación para protocolos de Internet
- Representación externa de datos y empaquetado
- Comunicación cliente-servidor
- Comunicación en grupo
- IPC UNIX

API para Internet



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

API para Internet — *Sincronización*

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.

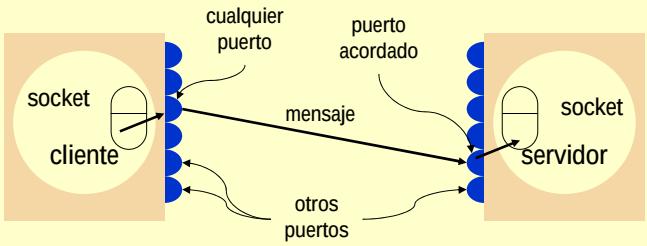
API para Internet — *Destinos de los mensajes*

- identificador de comunicación
  - dependiente de ubicación (p.ej.: puertos UNIX BSD)
    - debe ser bien conocido
    - un proceso puede tener varios puertos
  - por nombre
    - requiere un servicio de directorio (binder)
    - permite la re-ubicación pero no la migración.
  - independiente de ubicación
    (ej: Mach con puertos de alto nivel)
- proceso (ej. Sistema V)
- grupos (multidifusión)(p.ej.: Chorus)
  - procesos o puertos

API para Internet — *Fiabilidad - Ordenación*

- 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.

API para Internet — *sockets*



- socket = conector
  - ligado a un com-id. *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 para Internet — *API Java para direcciones*

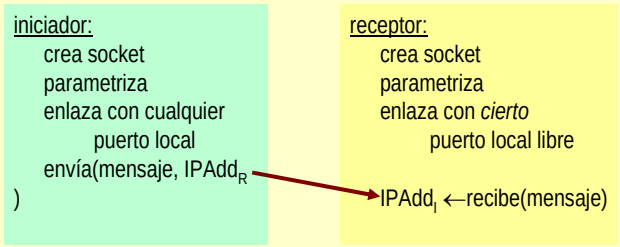
java.net

```
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...
}
```

API para Internet — *API Java para direcciones*

- Un InetAddres se crea mediante:
  getLocalHost( ), getByName( ), or getAllByName( )
- Cambia en v1.4:
  - más métodos
  - cambia en función de IPv4 o IPv6
  - Se introduce soporte para multidifusión.

API para Internet — *Comunicación con UDP*



- Tamaño: hasta 2^{16}=65kilobytes (estándar 8kB)
- Bloqueo:
  - *envía*() no bloqueante,
  - *recibe*() bloqueante (posibilidad de indicar un timeout).
- Identidad del emisor:
  - El socket de recepción suele estár abierto a cualquier emisor
  - es posible vincular el receptor a una sola IPAddr remota

API para Internet — *Comunicación con UDP*

- Modelo de fallo:
  - Fallos de omisión:
    - en el canal (que incluye los del emisor y del receptor)
    - provocados por desbordamiento de búfer, pérdida 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

API para Internet — *Comunicación con UDP*

DatagramPacket

métodos importantes:

- getData ( ) getPort ( ) getAddress ( )

DatagramSocket

throws SocketException

métodos importantes:

- send(DatagramPacket dP) throws IOException
- receive(DatagramPacket dP) throws IOException // dP vacío
- setSoTimeout ( ) throws InterruptedIOException
- connect ( ) para conectarse a una sola dirección

API para Internet — *Comunicación con UDP*

```
import java.net.*;
import java.io.*;
public class UDPCliente {
   public static void main(String args[]){// contenido del datagrama y servidor
    DatagramSocket    unSocket = null;
    DatagramPacket    peticion, respuesta;
    byte [ ] bufer   = new  byte[1000];
    try { unSocket = new DatagramSocket();
          byte [ ] m = args[0].getBytes();
          InetAddress unHost = InetAddress.getByName(args[1]);
          int pServicio = 6789;
          peticion = new DatagramPacket(m, args[0].length(), unHost, pServicio);
          unSocket.send(peticion);
          respuesta = new DatagramPacket(bufer, bufer.length);
          unSocket.receive(respuesta);
          System.out.println("Respuesta: " + new String(respuesta.getData( )));
    } catch (SocketException e){ System.out.println("Socket: " +e.getMessage());
    } catch (IOException e) { System.out.println("IO: " + e.getMessage( )); }
    } finally {  if  (unSocket != null) unSocket.close();  }
  }
}
```


API para Internet — *Comunicación con UDP*

```
import java.net.*;
import java.io.*;
public class UDPServidor{
   public static void main(String args[]){
   DatagramSocket unSocket = null;
      try {     unSocket = new DatagramSocket(6789);
                byte[ ] bufer = new byte[1000];
          while (true) {
             DatagramPacket peticion =
                new DatagramPacket(buffer, buffer.length);
              unSocket.receive(peticion);
             DatagramPacket respuesta =
                new DatagramPacket(peticion.getData( ),
             peticion.getLength( ), peticion.getAddress( ), peticion.getPort( ));
             unSocket.send(respuesta);
          }
      } catch (SocketException e){
          System.out.println("Socket: " + e.getMessage( ));
     } catch (IOException e) { System.out.println("IO: " + e.getMessage( ));}
   } finally { if (unSocket != null) unSocket.close( ); }
   }}
```

API para Internet — *Comunicación con 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.

API para Internet — *Comunicación con TCP*

- Aspectos importantes
  - Concordancia de tipos de datos

    Los procesos deben conocer el tipo de datos que se envían - reciben.
  - Bloqueo

    El receptor se bloquea siempre, y el emisor sólo cuando el canal no puede admitir más mensajes.
  - Hilos
    - El servidor suele estar multi-henebrado.
    - Si no está multi-henebrado se puede evitar el bloqueo del servidor si éste comprueba la existencia de datos en el stream con select ( ).

API para Internet — *Comunicación con 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)

API para Internet — *Comunicación con TCP*

ServerSocket

para un socket de aceptación de conexiones en el servidor

métodos importantes:

- Socket accept ( )

Socket throws UnknownHostException
throws IOException

para un socket de conexión

métodos importantes:

- InputStream getInputStream ( )
- OutputStream getOutputStream ( )

API para Internet — *Comunicación con TCP*

```
import java.net.*;
import java.io.*;
public class TCPCliente {
    public static void main (String args[ ]) {   // mensaje y el destino
        Socket s = null;
        try  { int puertoServicio = 7896;
            s = new Socket(args[1], puertoServicio);
            DataInputStream in = new DataInputStream( s.getInputStream( ));
            DataOutputStream out = new DataOutputStream( s.getOutputStream( ));
            out.writeUTF(args[0]);          // UTF es una codificación de string (S.4.3)
            String datos = in.readUTF( );
            System.out.println("Received: "+ datos) ;
        } catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage( ));
        } catch (EOFException e){System.out.println("EOF: "+e.getMessage( ));
        } catch (IOException e){System.out.println("IO: "+e.getMessage( )); }
        } finally {    if (s != null)
                       try { s.close( ); }
                       catch (IOException e)
                       { System.out.println("Close: "+e.getMessage( ));  }
        }
    } // fin main( )
}
```


API para Internet — *Comunicación con TCP*

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

public class TCPServidor {
    public static void main (String args[]) {
        try {
            int puertoServicio= 7896;
            ServerSocket socketEscucha = new ServerSocket(puertoServicio);
            while (true) {
                    Socket clientSocket = socketEscucha.accept();
                    Connection c = new Connection(socketCliente);
            }
        } catch (IOException e) {
            System.out.println("Listen: "+e.getMessage( ));
        }
    }
}
```

API para Internet — *Comunicación con TCP*

```
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket socketCliente;
    public Connection (Socket unSocketCliente) {
        try {   socketCliente = unSocketCliente;
                in = new DataInputStream( socketCliente.getInputStream( ));
                out =new DataOutputStream( socketCliente.getOutputStream( ));
                this.start();
        } catch (IOException e)  {System.out.println("Connection:
    "+e.getMessage()); }
    }
    public void run( ){  // un servidor de eco
        try {   String datos = in.readUTF( );
                out.writeUTF(datos);
        } catch (EOFException e) {
        System.out.println("EOF:"+e.getMessage( ));
        } catch (IOException e) {   System.out.println("IO:"+e.getMessage( ));
        } finally {
                try { socketCliente.close( ); }
                catch (IOException e) { /*close falló*/  } }
     }
}
```


Representación externa de datos — *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 envía series de bytes

Representación externa de datos *mecanismos de solución*

- Convertir los datos a un formato de «*representación externa de datos*» 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.

Representación externa de datos *mecanismos de solución*

- Alternativas usuales:
  - Java RMI: utiliza un procedimiento de serialización, basado en el conocimiento de las clases.
    - También .NET
  - CORBA: usa un lenguaje para la representación externa de datos (CDR) y un lenguaje de definición de interfaces (IDL)
  - Sun RPC: emplea también un lenguaje común (XDR) Variantes:
    - 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: ...

Representación externa de datos — *Common Data Representation*

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.

CDR -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;
}
```

CDR -Ejemplo

- La estructura de la 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*

Representación externa de datos — *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
}
```

Representación externa de datos *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***

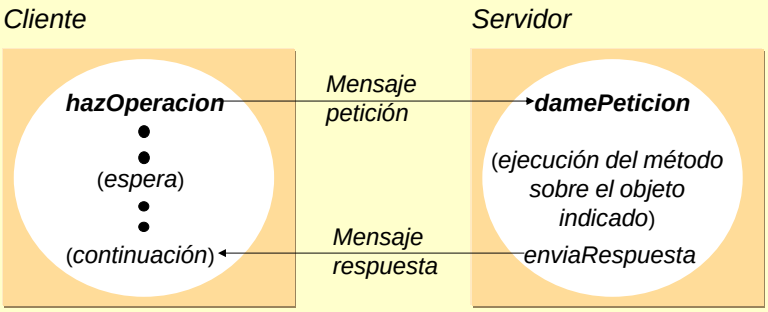
- **La forma serializada real contiene marcadores de tipo adicionales**
- **a0 y a1 son apuntadores**

Representación externa de datos *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

Comunicación cliente-servidor



- 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

Comunicación cliente-servidor *y RMI*

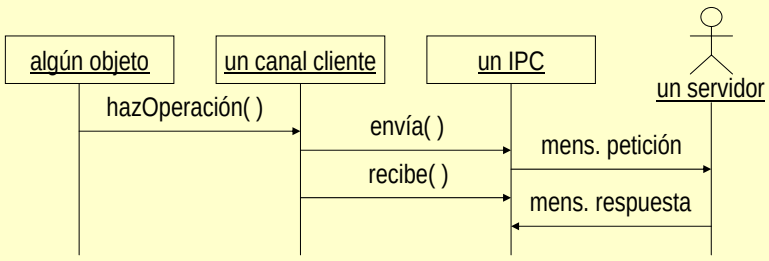
```
public byte[ ] hazOperacion( RemoteObjectRef objeto,
                             int idMetodo, byte [ ] argumentos)
   //   Empaqueta los datos;
   //   envía idMetodo y argumentos al objeto indicado en obj;
   //   se bloquea hasta que recibe la respuesta

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.
```

Comunicación cliente-servidor *y RMI*

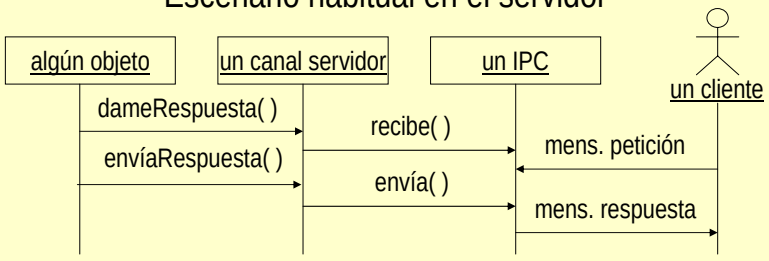
Estructura de un mensaje petición-respuesta

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

Comunicación cliente-servidor *operaciones*



Escenario habitual en el servidor



Escenario habitual en el cliente

Comunicación c-s *Modelo de fallos*

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:

Comunicación c-s *Modelo de fallos*

- Tiempos de espera límite (*timeouts*)
  - hazOperación puede encapsular reintentos
- Eliminación de peticiones duplicadas
  - a) El servidor puede repetir la operación si es preciso (si la op. es idempotente)
  - b) 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)
- Utilización de históricos
  - Permite retransmitir la respuesta en base al identificador.

Comunicación c-s *protocolos de intercambio RPC*

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: más estricta: cuando se requiere operaciones atómicas, o se requiere vaciar el historial.
También, una petición posterior puede servir como ACK.

Comunicación c-s *protocolos de intercambio RPC*

- 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 petición suele incluir el número de petición del cliente, el servidor pueden entender que una llamada RR reconozca un RRA anterior.

Comunicación en grupo

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

Comunicación en grupo *Problemas*

- Los de la comunicación unicast

Más

Identificador de comunicación para grupos

- Soporte de comunicación
- Semántica de entrega de mensajes:
  - Relación de pertenencia al grupo
  - Ordenamiento de los mensajes

Comunicación en grupo — *Tipos*

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

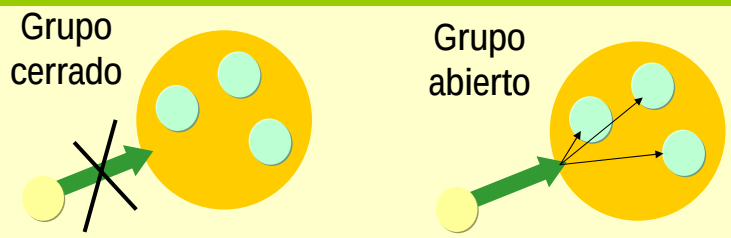
- Todos ellos utilizan las mismas dos primitivas:
  - envía(destino, mensaje)
  - origen ← recibe(mensaje)

  donde el identificador de comunicación destino y origen pueden ser el nombre de:
  - un proceso
  - un grupo de procesos

Comunicación en grupo — *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

Comunicación en grupo — *Administración de grupos*



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.

Comunicación en grupo — *Administración de grupos*

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)

Comunicación en grupo *direccionamiento de grupo*

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

Posibilidades:

a) Mecanismo de creación de direcciones multidifusión

b) Solo broadcast: varios procesos deben compartir su dirección de bajo nivel

c) Solo uno a uno: el nombre de bajo nivel debe contener la lista de elementos.

Comunicación en grupo *direccionamiento de grupo*

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)

Comunicación en grupo *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) A la dirección multicast o broadcast
     b) A las direcciones de los nodos
  4. El núcleo del receptor extrae los PID y envía los mensajes a los receptores locales

Comunicación en grupo *Multidifusión con/sin búfer*

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

Comunicación en grupo *Semánticas*

Send to All

Se envía una copia del mensaje a cada proceso, y con búfer.

Comunicación en grupo *Semánticas*

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.