



# **Desarrollo de Sistemas Distribuidos**

## **Tema 2** **Comunicación y Sincronización de** **Procesos**

# Contenidos

1. Paso de mensajes
2. Comunicación Cliente/Servidor
3. Llamada remota a procedimiento (RPC)
4. Invocaciones remotas o citas (“*rendez-vous*”)

# Paso de mensajes

- Conceptos:
  - **Formar** (“*marshalling*”): pone una colección de datos en un formato adecuado para transmitirlos en un mensaje. Consiste en:
    - **Poner en plano** (“*to flat*”) las estructuras de datos en secuencias básicas
    - Traducir los elementos básicos a una **representación de datos estándar** (Ej.: “*eXternal Data Representation*” o XDR de SUN)
    - Las operaciones para formar mensajes se pueden generar **automáticamente**
  - **Canal**: abstracción de una red de comunicación física que proporciona un camino de **comunicación** entre procesos y **sincronización** mediante 2 primitivas: *send* y *receive*

# Paso de mensajes

- Las notaciones difieren en:
  - **Ámbito y denominación** de los canales (p.ej., globales a procesos o asociados a subconjunto de ellos)
  - **Uso** (p.ej., flujos de información bidireccionales o no)
  - Cómo se **sincroniza** la comunicación (p.ej., síncrona/bloqueante, asíncrona/no bloqueante con buffer ilimitado o asíncrona con buffer limitado)
  - Operación de comunicación **no bloqueante** → su ejecución nunca retrasa al proceso que la invoca
    - Normalmente no bloquea *send*, aunque también existe *receive* no bloqueante
- La mayoría de las propuestas de notaciones son **equivalentes**, ya que un programa en una notación se puede escribir en otra, pero cada propuesta es más adecuada dependiendo del tipo de problema

# Paso de mensajes

- Notación aceptada, aunque existen variantes (“*timeouts*”, etc.):
  - *send* <puerto [**o canal**]>(<mensaje>)
  - *receive* <puerto>(<mensaje>)
  - *empty* <puerto> #para ver si la cola de un canal está vacía
- Tipo de comunicación:
  - **Síncrona**: ambas primitivas (*send* y *receive*) son bloqueantes
  - **Asíncrona**: normalmente sólo es bloqueante *receive*.
  - **Asíncrona con Buffer finito**: *receive* bloqueante y *send* cuando el buffer está lleno
    - Las implementaciones de *receive* no bloqueantes son complejas → requieren sondeos o interrupciones

# Paso de mensajes

- **Destino** de los mensajes:
  - Debe ser **conocido** por el emisor e independiente de la localización (**transparencia**), como p.ej. en DNS
  - Tipos de **destinos**:
    - **Proceso** (“referencia directa”): comunicación punto a punto
    - **Enlace** (“*link*”): punto a punto con indirección
    - **Puerto** (“*port*”): muchos a uno con indirección
    - **Buzón** (“*mailbox*”): muchos a muchos con indirección
    - **Difusión** (“*broadcast*”): muchos a muchos con indirección
    - **Selección** (“*multicast*”): muchos a muchos con indirección

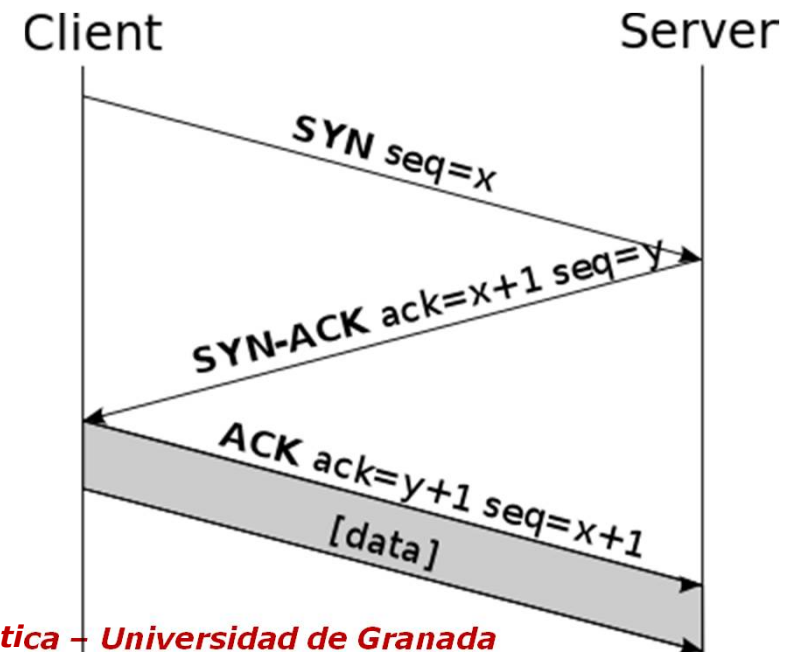
# Paso de mensajes

- **Protocolos de comunicación de grupos:**
  - Uso:
    - **Tolerancia a fallos** en servicios replicados
    - **Localización** de objetos en servicios distribuidos
    - **Mejor rendimiento** con servicios replicados
    - **Actualización múltiple** notificando eventos a varios procesos a la vez
  - Propiedades:
    - **Atomicidad:** el mensaje es recibido por todos o por ninguno
    - **Ordenación:** ejecución de operaciones en el mismo orden

# Paso de mensajes

- **Fiabilidad**

- **Fiable:** Los datos que emite el cliente serán recibidos por el servidor sin errores y en el mismo orden en que fueron emitidos
- Paso de mensajes fiable se puede construir partiendo de uno no fiable (p.ej.: mediante acuses de recibo)
- Ej.: TCP/IP





# Comunicación Cliente/Servidor

- **Protocolo petición/respuesta**
  - Comunicación típicamente **síncrona y segura/fiable** (la respuesta del servidor sirve como acuse de recibo)
  - **Alternativas de implementación:**
    - **Primitivas de comunicación** (*send* y *receive*). Inconvenientes:
      - Sobrecarga (más canales utilizados explícitamente)
      - Correspondencia entre *send* y *receive*
      - Garantía de reparto de mensajes si los servicios de red no la proporcionan
      - Ejemplo

# Comunicación Cliente/Servidor

- **Ejemplo: Asignador de recursos**

- Diseñar un asignador de recursos utilizando primitivas de paso de mensajes utilizando distintos canales de comunicación

//Algunas estructuras de datos y variables a utilizar por el asignador

```
var unidades: set of int; pendientes: queue of int;  
    disponibles: int:= MAX_UNIDADES;  
type clase_op = enum (ADQUIRIR,LIBERAR); port peticion;  
chan respuesta [1..n](int);
```

```
Cliente[i:1..n]:var unidad:int
```

```
...
```

```
    send peticion (i, ADQUIRIR, -1)...
```

```
    receive respuesta[i](unidad);
```

```
    //usar recurso...
```

```
    send peticion (i, LIBERAR, unidad);
```

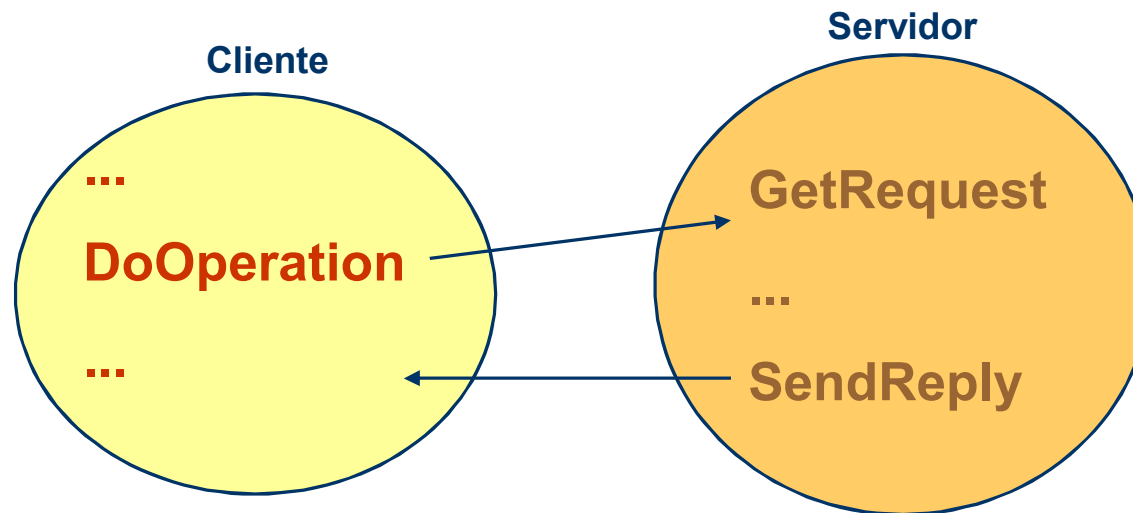
# Comunicación Cliente/Servidor

- **Protocolo petición/respuesta (request/response)**

- **Alternativas de implementación:**

- **Operaciones de comunicación:**

- Combinan aspectos de monitores (exportación de operaciones) y paso de mensajes síncrono (las peticiones bloquean a los clientes)
    - Tres primitivas de comunicación: DoOperation, GetRequest y SendReply



# Ejemplo de operaciones para el protocolo petición-respuesta

```
public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
```

```
public byte[] getRequest ();
```

```
public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
```

En RMI, en RPC este  
campo se omite



tipoMensaje – int (0=petición, 1=Respuesta)
idPetición
refObjeto
idMétodo (int o método en sí)
argumentos

[Fuente: Colouris 2001]

# Ejemplo de serialización de objetos en Java

```
class ClaseASerializar implements java.io.Serializable {  
    String unaCadena="Hola";  
    int unEntero=23;  
    double unComaFlotante=3.0;  
  
    public void muestraCampos() {  
  
        System.out.println("unaCadena: "+this.unaCadena);  
        System.out.println("unaEntero: "+this.unEntero);  
        System.out.println("unComaFlotante: "+this.unComaFlotante);  
    }  
  
}
```

# Ejemplo de serialización de objetos en Java

```
import java.io.*;

public class EjemploSerialización {

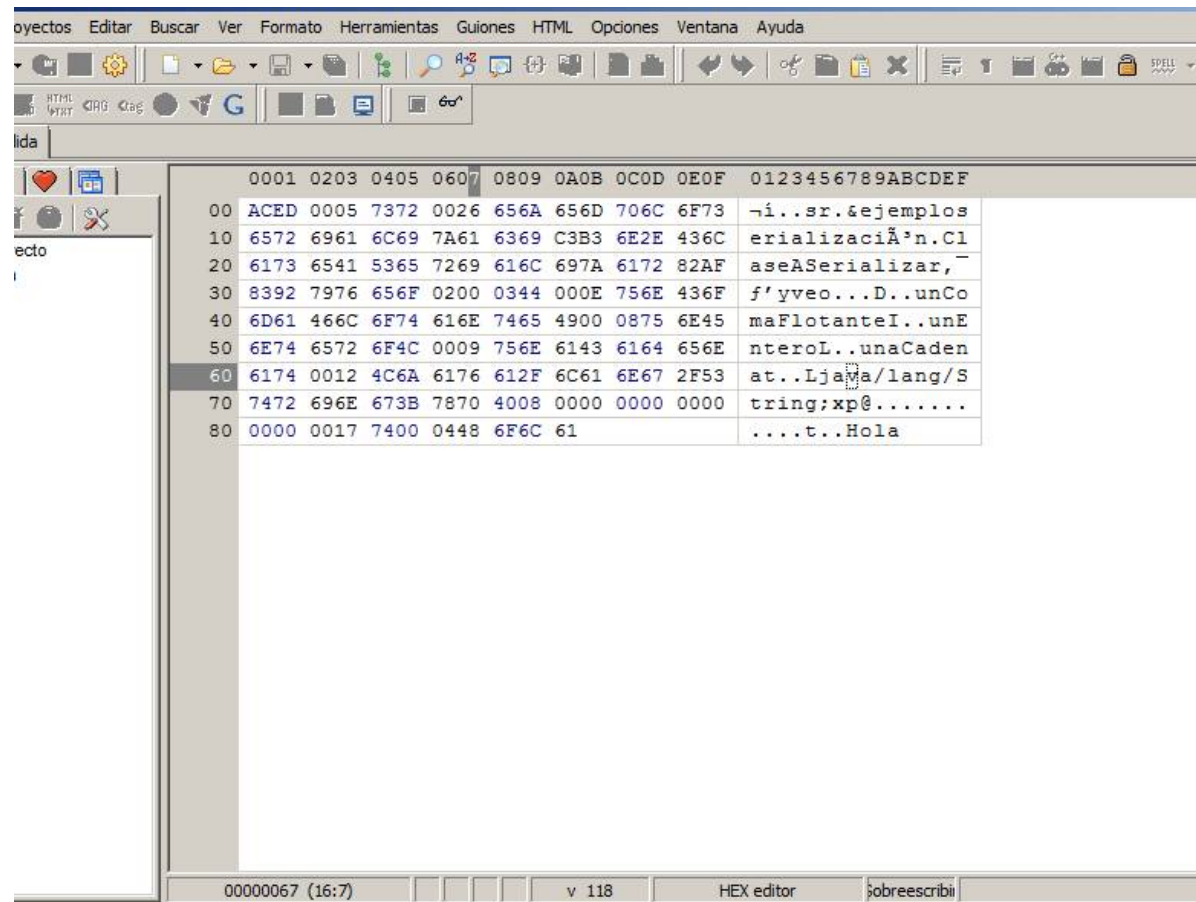
    public static void main(String[] args) {

        ClaseASerializar ejemploClase = new ClaseASerializar();

        try{
            FileOutputStream flujoSalida = new FileOutputStream("fichero.salida");
            ObjectOutputStream salida = new ObjectOutputStream(flujoSalida);
            salida.writeObject(ejemploClase);
            salida.close();
            flujoSalida.close();
            System.out.printf("Objeto serializado almacenado en 'fichero.salida'");
        }catch (EOFException e)
            {System.out.println("EOF:"+e.getMessage());}
        }catch (IOException e)
            {System.out.println("readline:"+e.getMessage());}
        }
    }
}
```

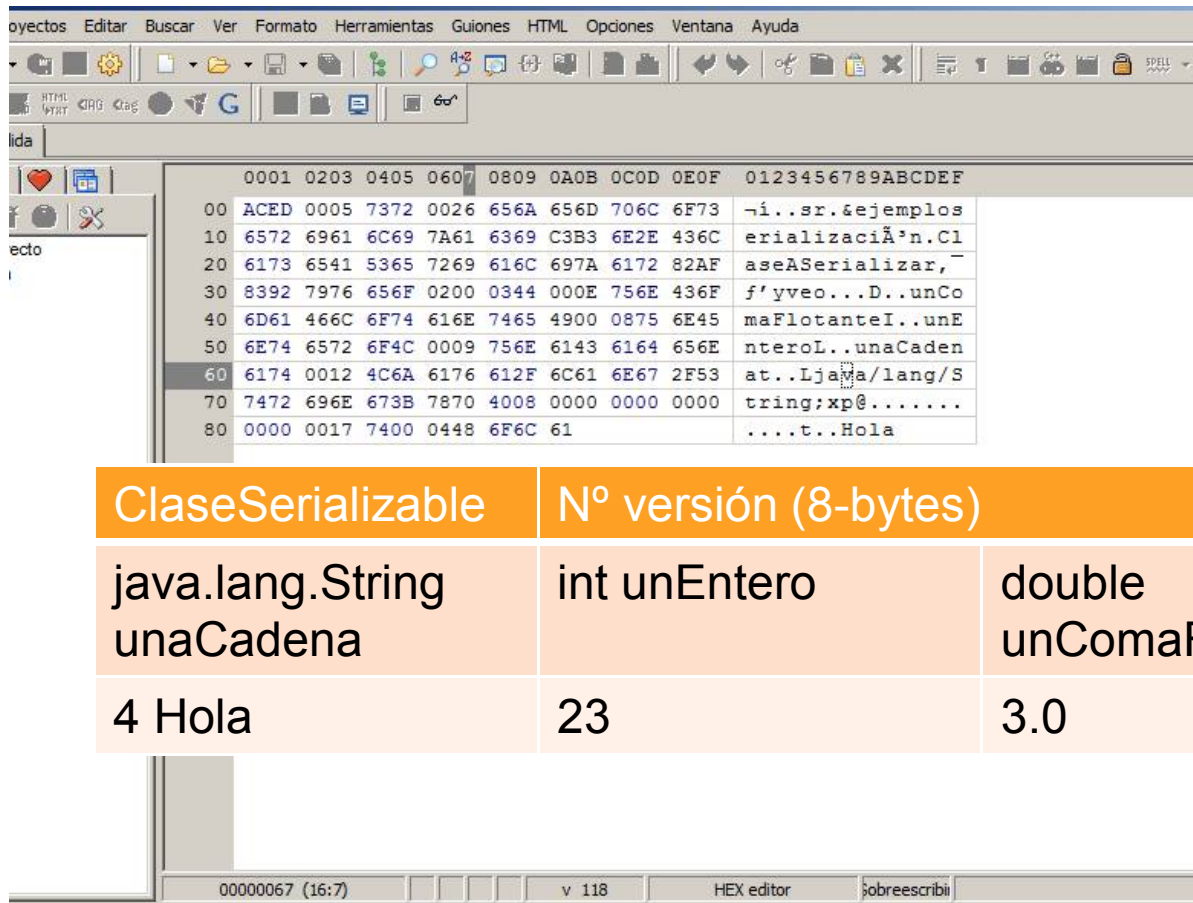
# Ejemplo de serialización de objetos en Java

**ACED000573720026656A656D706C6F73657269616C697A616369C3B36E2E  
436C6173654153657269616C697A617282AF83927976656F02000344000E  
756E436F6D61466C6F74616E7465490008756E456E7465726F4C0009756E  
61436164656E617400124C6A6176612F6C616E672F537472696E673B7870  
400800000000000000000000000017740004486F6C61**



# Ejemplo de serialización de objetos en Java

ACED000573720026656A656D706C6F73657269616C697A616369C3B36E2E  
436C6173654153657269616C697A617282AF83927976656F02000344000E  
756E436F6D61466C6F74616E7465490008756E456E7465726F4C0009756E  
61436164656E617400124C6A6176612F6C616E672F537472696E673B7870  
400800000000000000000000017740004486F6C61





# Comunicación Cliente/Servidor

- Implementación del **protocolo** petición/respuesta (**modelo de [gestión] de fallos**) → depende de las garantías de envío/entrega que se ofrezcan
  - **Plazo de tiempo** ("*time-out*"). [Tras *DoOperation*] Alternativas cuando se cumple:
    - Devolver fallo
    - Repetir petición y, eventualmente, devolver fallo si es el caso (si es lo + probable)
  - **Filtrar mensajes** de petición duplicados. Formato mensaje:
    - **Tipo:** petición o respuesta
    - **Identificador petición:** número petición
    - **Referencia objeto remoto:** estará serializada
    - **Identificador procedimiento:** pueden estar numerados o empaquetarse/serializarse
    - **Argumentos**

tipoMensaje – int (0=petición, 1=Respuesta)
idPetición
refObjeto
idMétodo (int o método en sí)
argumentos

# Comunicación Cliente/Servidor

- Implementación del **protocolo** petición/respuesta (**modelo de [gestión] de fallos**)
  - Mensajes de respuesta perdidos
    - Operaciones **idempotentes** en servidores permiten reejecución proporcionando los mismos resultados
  - Gestión de históricos
    - Retransmisión de respuestas sin reejecución de operaciones
    - Manejo eficiente de la estructura de historia:
      - Nueva petición como reconocimiento de la respuesta previa
      - Reconocimientos del cliente (*acknowledgments*) ayudan a descartar entradas en la historia
      - También pueden descartarse transcurrido un periodo de tiempo

# Comunicación Cliente/Servidor

- Modelo de fallos protocolo **petición/respuesta**: garantías de envío/informar sobre errores con *doOperation*

<i>Garantías de envío</i>			<i>Semántica de invocación</i>
<i>Reintentar petición</i>	<i>Filtrar duplicados</i>	<i>Reejecutar procedimiento "o" Retransmitir respuesta</i>	
No	No aplicable	No aplicable	Quizás
Sí	No	Reejecutar procedimiento	Al menos una vez
Sí	Sí	Retransmitir Respuesta	Exactamente una vez

# Comunicación Cliente/Servidor

- Modelo de fallos protocolo **petición/respuesta**: garantías de envío/informar sobre errores con *doOperation*

Garantías de envío			Semántica de invocación
Reintentar petición	Filtrar duplicados	Reejecutar procedimiento "o" Retransmitir repuesta	
No	No aplicable	No aplicable	Quizás
Sí	No	Reejecutar procedimiento	Al menos una vez
Sí	Sí	Retransmitir Respuesta	Exactamente una vez

--- Sun RPC (idempotente)  
 --- RPC, RMI, CORBA

# RPC

- **Introducción:**
  - En el modelo C/S los servicios **proporcionan varias operaciones**
  - La comunicación C/S se basa en un protocolo **petición/respuesta**
  - Los mecanismos de RPC integran esta organización con lenguajes de programación procedurales convencionales
  - Se modela y diseña como una llamada a procedimiento local, pero esta se ejecuta remotamente
  - El servidor es visto como un módulo con **una interfaz que exporta operaciones** y con un tiempo de vida distinto
  - **Biblioteca de soporte a servicios para aislar cuestiones como:** diferencias entre procedimientos locales y remotos, localización del servidor, mejora de rendimiento por medio de cachés

# RPC

- Semántica:
  - Parámetros de **entrada/salida** (comunicación bidireccional)
  - Sólo uso de variables locales (no hay variables globales para servidor y cliente)
  - No tienen sentido punteros, por tanto, se hacen **clonaciones** (estructura interna oculta por modularidad)
  - Los servidores pueden devolver **referencias opacas** que no pueden ser interpretadas en el entorno del cliente (p.ej.: cookies, datos de autenticación para seguridad y eficiencia)

# RPC

- Cuestiones de diseño:
  - **Clases de sistemas o middlewares RPC:** mecanismo integrado en el lenguaje de programación (permite que algunos requisitos, p. ej. excepciones, puedan tratarse con construcciones del lenguaje → Argus) “o” de propósito general (no dependen de un entorno particular → Sun RPC)
  - **Características** del lenguaje de definición de interfaces (**IDL**).  
Deben permitir especificar: nombres procedimientos, tipo de parámetros y dirección (E,S, E/S)
  - **Manejo de excepciones:** Notificar errores debidos a distribución, plazos de tiempo para comunicaciones, errores de ejecución del procedimiento

# RPC

- **Protocolo petición/respuesta**
  - (Normalmente) Tres protocolos diferentes para informar sobre errores en **RPC**
  - Diferentes semánticas en presencia de fallos

<b>Nombre</b>	<b>Mensaje enviado por</b>		
	<b>Cliente</b>	<b>Servidor</b>	<b>Cliente</b>
<b>R</b>	Petición		
<b>RR</b>	Petición	Respuesta	
<b>RRA</b>	Petición	Respuesta	Reconocimiento

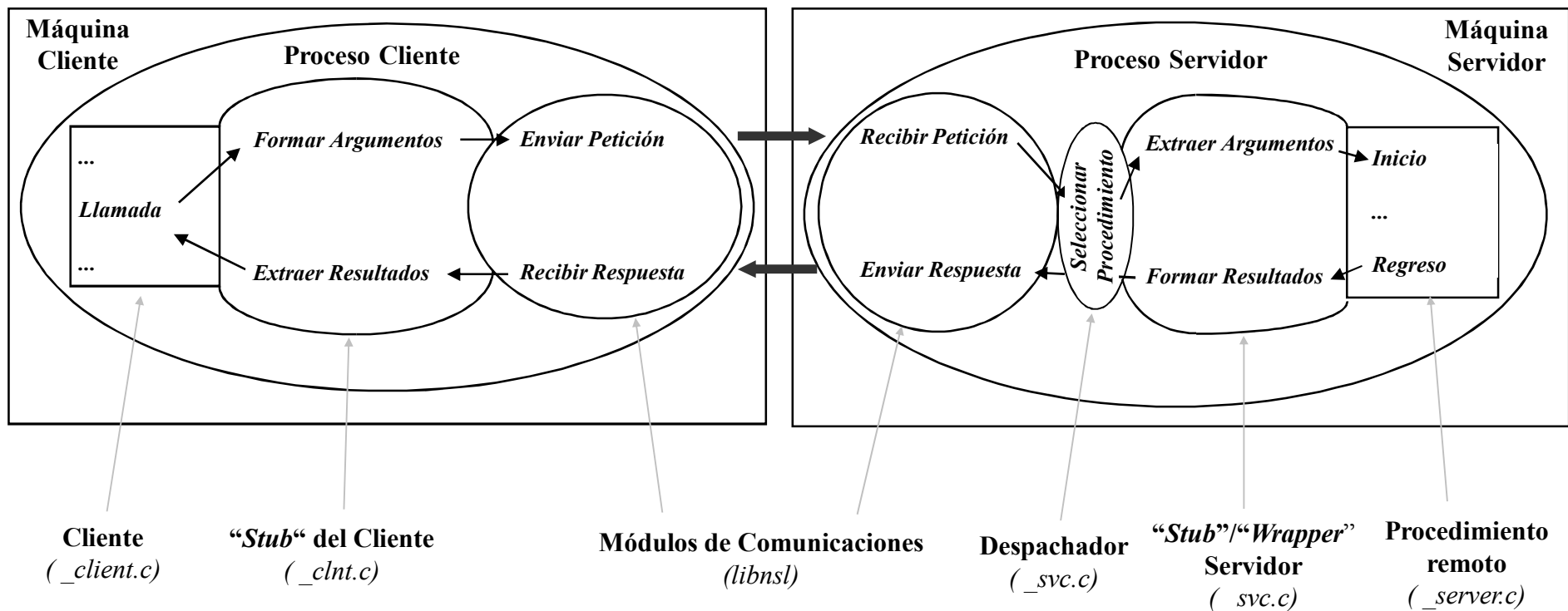


# RPC

- Cuestiones de diseño:
  - **Transparencia:**
    - **Manejar errores** debido a que RPC:
      - **Más vulnerable** (red, otra computadora, otro proceso)
      - Toma **más tiempo** que una local
    - Por tanto, **no debería ser transparente**, sino explícita al programador
    - Aunque debería **ocultar detalles de bajo nivel** de paso de mensajes, pero no retardos o fallos

# RPC

- Implementación:



# RPC

## 1. Procesamiento Interfaz

- **Integra** el mecanismo RPC con programas cliente y servidor formando y extrayendo argumentos y resultados
- Se **compila una especificación** escrita en un lenguaje de definición de interfaces (IDL) y genera cabeceras, plantillas y *stubs*:
  - En el cliente convierte llamada local a remota
  - En el servidor selecciona y llama al procedimiento adecuado

## 2. Módulo comunicaciones

- Implementa protocolo petición-respuesta

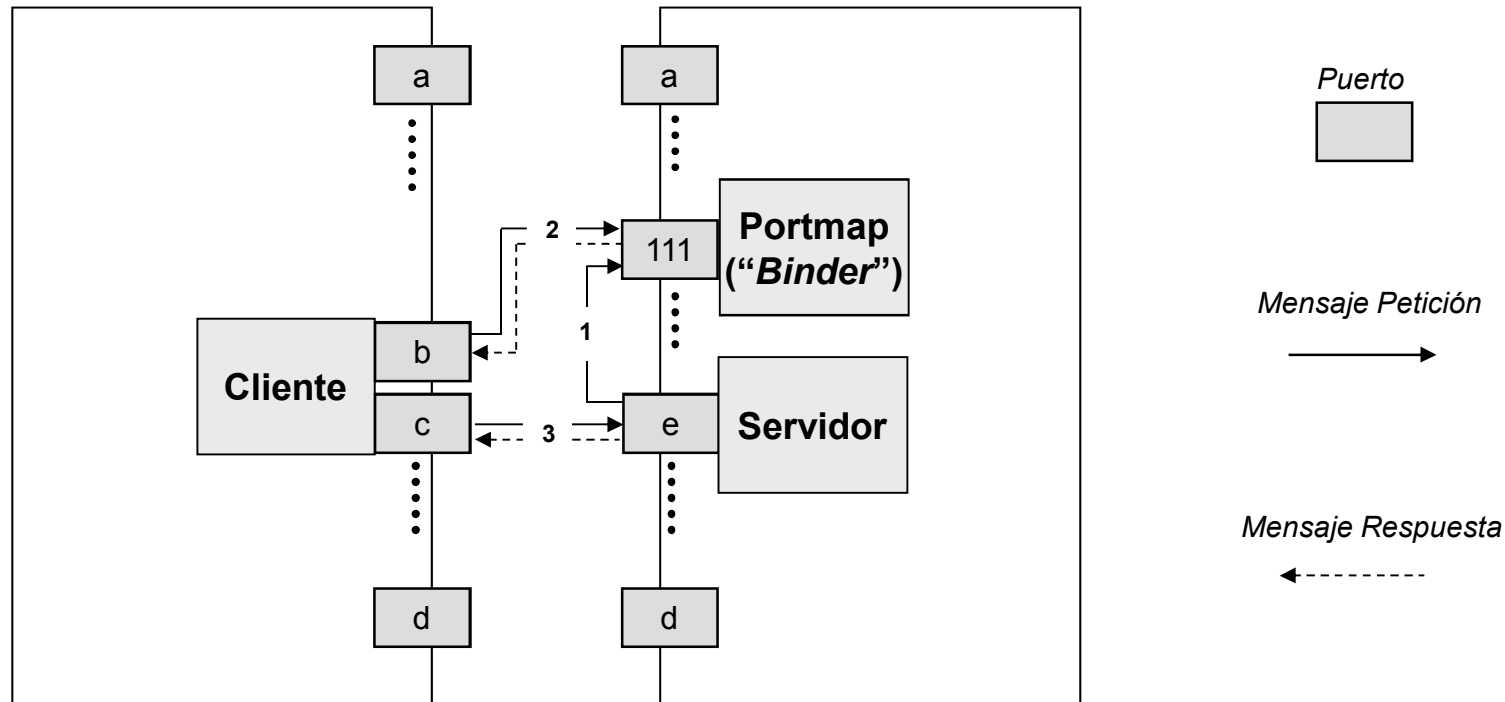
# RPC

## 3. Servicio de ligadura ("*binding*")

- Mecanismo para localización del servidor
- **Asociación** de un nombre a un identificador de comunicación
- Mensaje de petición se dirige a un puerto concreto
- Se evalúa cada vez que el cliente lo requiera, ya que el servidor puede ser **relocalizado**
- Servicio del cual dependen otros, por tanto, debe ser **tolerante a fallos**
- Los servidores exportan (**registran**) sus servicios y los clientes los importan (**buscan**)

# RPC

## 3. Servicio de ligadura ("*binding*")



**Operaciones** {

- Registro** (<nombre\_servicio>, <puerto\_servicio>, <versión>);
- Retirar** (<nombre\_servicio>, <puerto\_servicio>, <versión>);
- Buscar** (<nombre\_servicio>, <versión>) : <puerto>;

# RPC

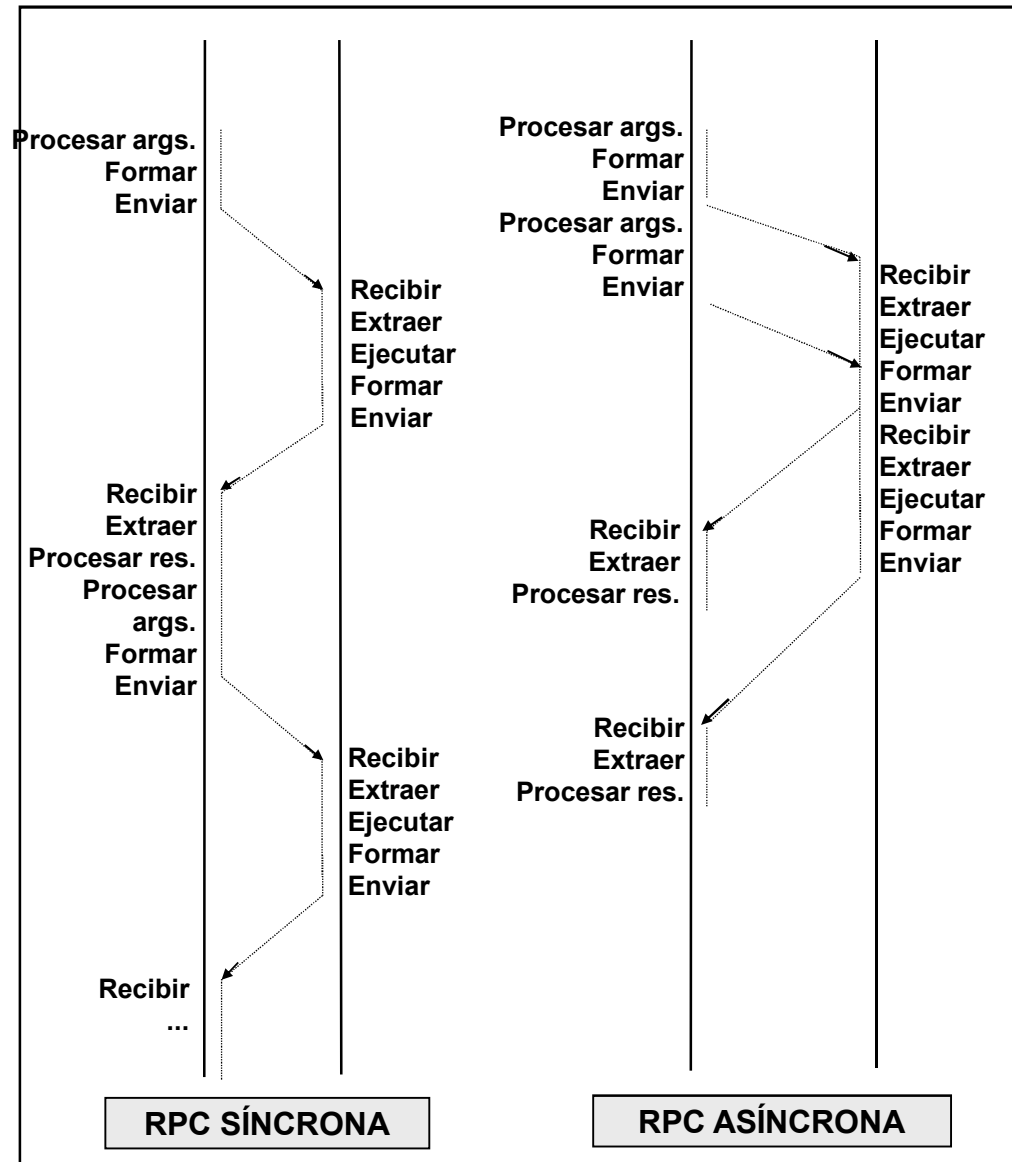
## 3. Servicio de ligadura ("*binding*")

- Alternativas para localizar el ligador:
  1. **Dirección conocida:** El cliente/servidor han de ser recompilados cuando el ligador se relocaliza
  2. El **sistema operativo** proporciona la información en tiempo de ejecución (p.ej. mediante variables de entorno)
  3. Cuando cliente/servidor se lanzan, envían **mensajes de difusión (*broadcast*)** para que así el ligador (*binder*) responda con la dirección

# RPC Asíncrona

- Requisitos comunes:
  - El cliente envía muchas peticiones al servidor
  - No se necesita una respuesta a cada petición
- Ventajas:
  - El servidor puede **planificar operaciones** más eficientemente
  - El cliente **trabaja en paralelo**
  - Se facilita el **cálculo de peticiones paralelas** en el caso de varios servidores

# RPC Asíncrona





# RPC Asíncrona

- Optimizaciones:
  - Varias peticiones en una sola comunicación: Se almacenan mensajes hasta que:
    - a) Se cumple un **plazo de tiempo**
    - b) Se realiza una petición que **requiere respuesta**
  - El cliente puede proceder si no espera una respuesta que puede obtener más tarde

# Citas

- A veces nombradas como **citas extendidas** en entornos distribuidos
- Las invocaciones remotas se sirven mediante una **instrucción de aceptación** (par de instrucciones *call – in, se ve más adelante*)
- **RPC** es una **comunicación intermódulo**
- Las instrucciones de comunicación están **limitadas**:
  - *A menudo un proceso desea comunicarse con más de un proceso, quizás en puertos diferentes, y no se sabe el orden en el que los otros procesos desean comunicarse con él*

# Citas

- **No determinismo mediante instrucciones guardadas:**

*Var* a, b, c;

a, b, c := A, B, C;

*do*

a > b → a, b := b, a;

□ b > c → b, c := c, b;

*od*

# Citas

- **Comunicación no determinista** = instrucciones guardadas + instrucciones de comunicación



# Citas

- **Semántica** de la guarda:  $B ; C \rightarrow S$ 
  1. Tiene **éxito** si **B** es verdad y la ejecución de **C** no produce retardo
  2. **Falla** si **B** es falso
  3. **Bloquea** si **B** es verdad, pero **C** no se puede ejecutar sin producir retardo
- **B** no puede cambiar hasta ejecutar otras instrucciones de asignación, ya que no hay **variables globales**
- La guardas pueden incluir instrucciones de comunicación de entrada o salida

- Las instrucciones guardadas de comunicación pueden combinarse en construcciones:
  - Alternativas (*IF*):
    - Si al menos una guarda tiene éxito, una de ellas se escoge de forma no determinista ejecutando **C** y **S**
    - Si todas las guardas fallan, entonces *IF* falla o termina
    - Si no hay guardas con éxito y algunas están bloqueadas, la ejecución se retrasa hasta que la primera tenga éxito
  - Repetitivas (*DO*) igual que *IF* con ejecución iterativa hasta que todas las guardas fallen

# Citas

- Ejemplo anterior

**Var** a, b, c;

a, b, c := A, B, C;

**do**

a > b → a, b := b, a;

□ b > c → b, c := c, b;

**od**

# Citas

- **Ejemplo:** Servidor de ficheros
  - Hasta ***n*** ficheros abiertos a la vez por ***m*** clientes
  - El acceso a cada fichero se proporciona por un proceso servidor de fichero distinto
  - Paso de mensajes síncrono
- Operaciones sobre archivos: **abrir, leer, escribir, cerrar**
- Canales de comunicación:
  - `mailbox abrir (string nombre_fichero, int n_cliente);`
  - `chan respuesta_abrir [1..m] (int n_serv_fichero);`
  - `//tantos como clientes`
  - `chan leer [1:n] (...);`
  - `chan escribir [1:n] (...);`
  - `chan cerrar [1:n] (...);`
  - `chan respuesta [1..m] (...);`
- Modelar **un** proceso servidor de fichero



# Citas

- **Ejemplo:** Servidor de servicios de fichero. Hasta *n* ficheros abiertos a la vez. El acceso a cada fichero se proporciona por un proceso servidor de fichero distinto
- Operaciones sobre archivos: abrir, leer, escribir, cerrar

Ficheros [i:1..n]::

var nombref:string; args: otros tipos;

índice\_cliente:int; resultados:int;

fichero\_abierto: bool;

buffer\_local, caché, dirección\_de\_disco, ...;

do receive abrir (...)

# Citas (*Citas extendidas*)

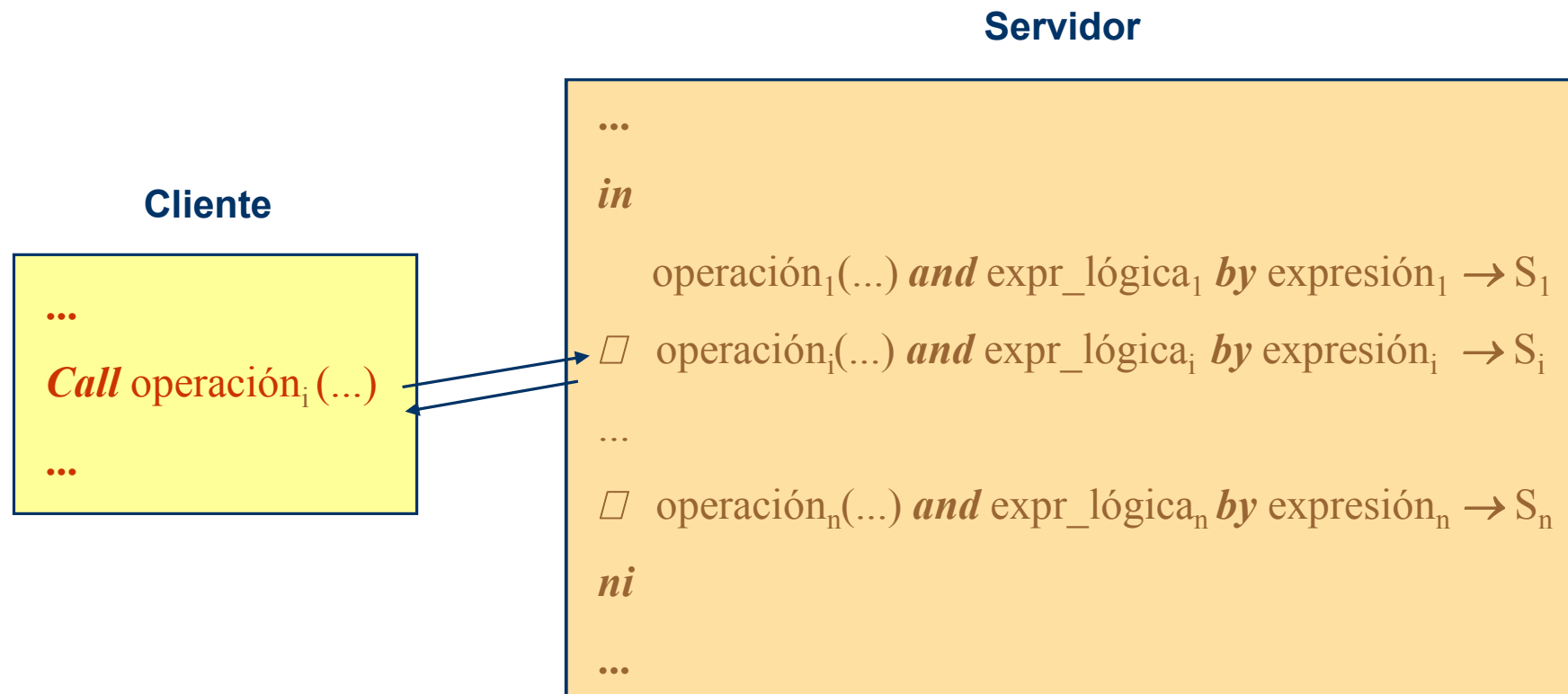
- Órdenes guardadas de comunicación:

- Un proceso **exporta operaciones** de forma similar a RPC
- Otro proceso invoca operaciones exportadas
  - **call** <nombre\_proceso>.<nombre\_operación>(<argumentos>);
- El proceso servidor atiende invocaciones en su contexto de ejecución mediante **instrucciones de aceptación** (**in** <nombre\_operación> (<parámetros\_formales>) → S ni)
- El **ámbito** de los parámetros formales es el de la operación guardada
- Una guarda de una operación **tiene éxito** cuando:
  - a) Se ha **invocado la operación**
  - b) La expresión lógica **se evalúa a verdad**
- La ejecución **se retrasa** hasta que una guarda tiene éxito → no determinismo cuando hay varias

guarda

# Citas extendidas

- Construcción alternativa con órdenes guardadas de comunicación:



# Citas extendidas

- **Características:**

- Operaciones en el **contexto del proceso** que especifican puntos de comunicación de **muchos a uno**
- Sin parámetros hay **sincronización** y no comunicación
- El servidor puede definir distintas guardas par la invocación de una misma operación exportada, y por tanto, producir **efectos diferentes** ante la invocación de un mismo servicio
- Las invocaciones se sirven en los **instantes que desee el servidor**
- A diferencia de RPC, el **servidor** es un **proceso activo** que se está ejecutando antes y después de servir una invocación remota

# Citas extendidas. Ejemplo buffer acotado en el Problema del Productor-Consumidor

Bufferacotado::

```
  op poner (dato:T);
    tomar (var resultado:T);
var buffer[1:n]:T
    cabecera:int:=1; cola:int:=1; contador:int:=0;
do true ->
  in poner (dato) and (contador < n) ->
    buffer[cola]:=dato;
    cola:=(cola mod n)+1;
    contador:=contador+1;
  □ tomar (resultado) and (contador > 0) ->
    resultado:=buffer[cabecera];
    cabecera:=(cabecera mod n)+1;
    contador:=contador-1;
  ni;
od;
```

# Citas extendidas. Ejemplo Ada (1 elemento)

```
task Buffer1 is
  entry Escribir (Elem: TElemento);
  entry Leer (Elem: out TElemento);
end Buffer1;
task body Buffer1 is
  ElemLocal: TElemento;
  tamBuffer : constant := 1;
begin
  loop
    select when (nElementos < tamBuffer)
      accept Escribir (Elem: TElemento) do
        ElemLocal:= Elem;  -- Guarda el elemento.
        nElementos:=nElementos+1; Ada.Text_IO.Put_Line("Elemento escrito");
      end Escribir;
    or when (nElementos > 0)
      accept Leer (Elem: out TElemento) do
        Elem := ElemLocal;  -- Devuelve el elemento.
        nElementos:=nElementos-1;Ada.Text_IO.Put_Line("Elemento leído");
      end Escribir;
    end select;
  end loop;
end Buffer1;
...
```

-- Ejemplos de llamadas

Buffer1.Escribir(elemento), Buffer1.Leer(elemento)...

Fuente: [wikibooks]

# Citas extendidas. Definición arrays en Ada

```
Tam_máx : constant := 100;
subtype T_Rango is Positive range 1 .. Tam_máx;
cabecera, cola : T_Rango := 1;
Buffer : array (T_Rango) of TElemento;

-- Alternativamente
Buffer : array (0 .. Tam_máx) of TElemento;
Buffer : array (0 .. 100) of TElemento;...

-- Otros ejemplos de declaraciones para variables (no necesario)
nElementos : Natural range 0 .. Tam_máx := 0;

-- Ejemplos de acceso a elementos en posiciones de un array
Buffer(3) := otroElemento;
for i in Buffer'Range loop
    Buffer(i) := unElemento;
end loop;
```