Sistemas Distribuidos

Comunicación en Sistemas Distribuidos

Índice

- Introducción
- Paso de mensajes
  - Comunicación punto a punto
  - Comunicación de grupo
  - Sistemas de colas de mensajes (MOM)
- Llamadas a procedimientos remotos (RPC)
  - Sun/ONC RPC
- Invocación de métodos remotos (RMI)
  - Java RMI y CORBA

Introducción

- Sistema de comunicación: Espina dorsal del SD
- Paradigmas de comunicación
  - ¿Qué API de comunicación ofrece SD a las aplicaciones?
- Memoria compartida (¿en SD?) vs. Paso de mensajes
- Adecuación del paradigma a las distintas arquitecturas
  - cliente/servidor; editor/subscriptor; P2P
- Grado de acoplamiento del paradigma: espacial y temporal
- Comunicación persistente
  - SD almacena información hasta que destinatario(s) lo obtenga(n)
  - Incluso aunque emisor ya no exista → desacoplamiento temporal
- Patrón de comunicación (cardinalidad):
  - unidifusión versus multidifusión

Paradigmas de comunicación

- Paso de mensajes
  - Comunicación punto a punto
    - Unidifusión de mensajes no persistentes (sockets, MPI,…)
  - Comunicación de grupo
    - Multidifusión de mensajes no persistentes (ISIS, JGroups,…)
  - Sistemas de colas de mensajes (*Message-oriented middleware*)
    - Paso de mensajes persistentes
- Llamadas a procedimientos remotos (RPC) (ONC, DCE,…)
- Invocación de métodos remotos (RMI)
  - Entornos distribuidos basados en objetos (Java RMI, CORBA,…)
- Servicios web → Sistemas orientados a servicios
- Memoria compartida (tema 6)
  - *Distributed Shared Memory*
  - Espacios de tuplas

Sistemas Distribuidos

Paso de mensajes

Índice

- Introducción
- Paso de mensajes
  - Comunicación punto a punto
  - Comunicación de grupo
  - Sistemas de colas de mensajes (MOM)
- Llamadas a procedimientos remotos (RPC)
  - Sun/ONC RPC
- Invocación de métodos remotos (RMI)
  - Java RMI y CORBA

Paradigma de paso de mensajes

- HW de paso de mensajes → API de paso de mensajes
- Sist. de paso de mensajes: Capa sobre protocolo de transporte
- Alternativas de diseño en aspectos como:
  - API de comunicación ofrecido
  - Direccionamiento: ¿cómo especifica origen/destino de comunicación?
  - Especificación del mensaje
  - Optimización de transferencias (*zero-copy*)
  - Integridad de los mensajes
  - Representación externa de datos
  - Grado de sincronía (y *buffering*)
- Ejemplos con distintos niveles de funcionalidad (sockets, MPI)

API de paso de mensajes

- MPI

  int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

  int MPI_Recv(void *buf, int count, MPI_Datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
- Sockets datagrama

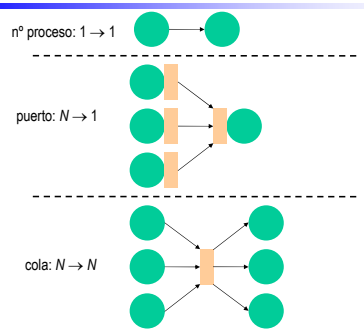
  ssize_t sendto(int socket, const void *buffer, size_t length, int flags, const struct sockaddr *dest_addr, socklen_t dest_len);

  ssize_t recvfrom(int socket, void * buffer, size_t length, int flags, struct sockaddr *address, socklen_t * address_len);
- Esquemas con conexión
  - Existen además primitivas para conectar y desconectar
  - Operaciones de envío y recepción no incluyen direcciones

Esquemas de direccionamiento

- Usando número de proceso (MPI):
  - En envío: nº proceso destinatario
  - En recepción: nº proceso origen; sólo interacción 1 → 1
    - O cualquiera (*MPI_ANY_SOURCE*): interacción *N* → 1
  - Difícil asignar nº proceso único en entorno de propósito general
    - Pero no en aplicación ejecutada en entorno de computación paralela
- Usando puertos: buzón asociado a una máquina (sockets)
  - Comunicación entre puertos
  - Proceso reserva uso de un puerto de su máquina (*bind* de sockets)
  - Envío: desde puerto origen local a puerto destino especificados
  - Recepción: de puerto local; interacción *N* → 1
  - Sockets INET: ID puerto = dir. IP + nº puerto + protocolo (TCP|UDP)
- Usando colas: buzón de carácter global; interacción *N* → *N*
  - Sistemas de colas de mensajes; desacoplamiento espacial+temporal

Modos de interacción punto-a-punto



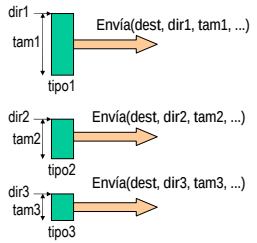
Especificación del mensaje

- Con o sin información de tipos (MPI vs. sockets)
  - Sin ella: aplicación debe gestionar heterogeneidad
  - Con ella: sistema de comunicaciones gestiona heterogeneidad
- Clases de mensajes (etiquetas)
  - Sistema de comunicación puede gestionar clases de mensajes
    - En envío: especifica clase de mensaje enviado
    - Recepción: especifica clase de mensaje que se quiere recibir
      - o usa comodín (*MPI_ANY_TAG*)
  - Múltiples canales sobre una misma comunicación
  - Diversas aplicaciones como por ejemplo:
    - Establecer prioridades entre mensajes
    - En cliente-servidor puede identificar operación a realizar
    - En editor-subscriptor basado en temas como identificador de tema
  - Disponible en MPI como parámetro de primitivas (*tag*)
  - No soportado en sockets, aunque sí mensajes urgentes (OOB)

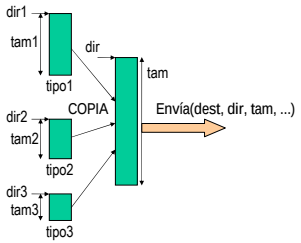
Zero-Copy

- Reducir al mínimo (≈ a cero) copias entre zonas de memoria
- *Escenario 1*: envío de *N* datos *dispersos* de emisor a receptor
  - N envíos: sobrecarga de llamada de as + fragmentación de mensajes
  - Reserva de *buffer* y 1 envío: sobrecarga de copias
  - Funciones *scatter/gather*: minimizar copias y llamadas
    - UNIX: *readv*, *writev*, *sendmsg*, *recvmsg*
- *Escenario 2*: envío de un fichero
  - Uso de operaciones convencionales de lectura y envío
    - Dos copias de memoria: de *buffer* de sistema a de usuario y viceversa
  - Uso de proyección de ficheros (*mmap*) y envío
    - Una copia de memoria a *buffer* de sistema en envío
  - Uso de operaciones de transferencia directa entre descriptores
    - No requiere copias entres *buffers*; reduce nº llamadas al sistema
    - Linux: *sendfile*, *splice*

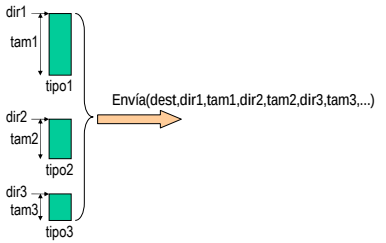
Datos dispersos: Envío múltiple



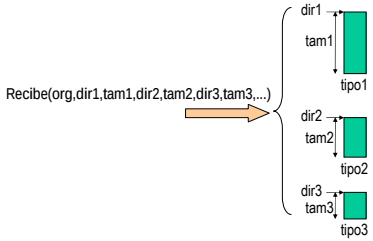
Datos dispersos: Envío con copia



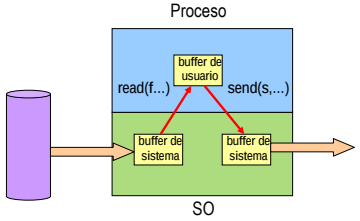
Datos dispersos: Envío *gather*



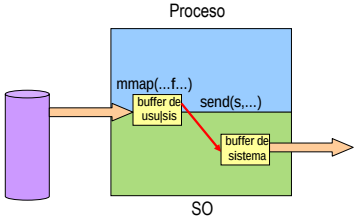
Datos dispersos: Recepción *scatter*



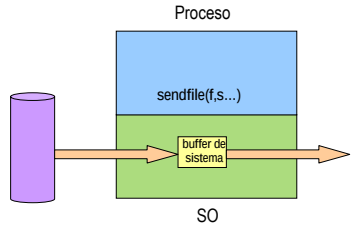
Envío convencional de fichero



Envío con proyección de fichero



Envío *zero-copy* de fichero



Integridad de los mensajes

- En recepción debe especificarse buffer de tamaño ≥ mensaje
  - Si menor: pérdida de resto del mensaje (MPI y sockets datagrama)
  - Se mantiene integridad de los mensajes
  - Nunca se entregan restos de mensajes ni dos mensajes juntos
- Excepto en comunicación como flujo de bytes (sockets *stream*)
  - Datos de mensaje no leídos se obtienen en próxima recepción
  - Recepción puede devolver datos de fragmentos de mensajes
  - Recepción puede devolver dos mensajes junto
  - Si se requiere no mezclar mensajes de tamaño variable se puede:
    - Enviar longitud
    - Usar un separador
    - Usar 1 conexión/mensaje y hacer *shutdown* de socket de envío

Representación externa de datos

- Emisor y receptor misma interpretación de información
  - Misma cuestión, y soluciones, para lector y escritor de un fichero
- Procesadores, lenguajes, compiladores difieren en:
  - Orden de bytes en tipos numéricos (*endian*)
  - Tamaño de datos numéricos (en C: ¿tamaño de int, long,…?)
  - *Strings* (con longitud vs. carácter terminador (¿qué carácter?))
  - Formatos de texto (ISO-8859-1, UTF-8,…)
  - Organización estructuras datos (compactación, alineamientos,…),…
- Se necesitan "serializar" los datos para enviar/almacenar
  - Asegurando misma interpretación en sistema heterogéneo
  - Eficientemente (en *serialización*, en uso de red/disco,…)
  - Facilitando la programación de la *serialización*
  - Admitiendo cambios incrementales en protocolo
    - P.e. protocolo con nuevo campo opcional pero cliente antiguo sigue OK

Marshalling

- Operación para *serializar* información en emisor
  - Y la operación inversa (*unmarshalling*) en receptor
- Con paso de mensajes puede ser:
  - Responsabilidad del programador (sockets)
    - Sockets tampoco ofrece funciones *serialización* (excepto para int: *htonl*…)
  - Automático (MPI)
- RPC/RMI lo realizan automáticamente
- Alternativas:
  - S. de comunicación en emisor convierte a formato de receptor
    - transformar a formato de cualquier receptor
  - S. de comunicación en receptor convierte a su formato
    - transformar desde formato de cualquier emisor
  - S. de comunicación en emisor convierte a formato externo
    - Sólo transformar de nativo a externo y viceversa
    - Ineficiente si formato de emisor = receptor pero ≠ de externo

Formato de *serialización* de datos

- Define cómo se transmiten/almacenan datos (*wire protocol*)
  - Secuencia de bits que representan cada dato
- Alternativas:
  - Formato propio vs. Estándar (mejor)
  - Texto vs. binario: menos compacto pero interpretable por usuarios
  - Información de tipos implícita o explícita:
    - Implícita: emisor y receptor conocen tipos de parámetros
      - no viaja info. de tipos con datos
    - Explícita: disponible información explícita de tipos
      - Viaja mezclada con datos o como referencia a un esquema
    - Explícita más flexible (permite reflexión) pero menos compacto
  - Información de nombres de campos implícita vs. explícita:
    - Explícita: viaja nombre de campo con datos
      - Puede facilitar cambios incrementales de un protocolo
  - Información explícita de campos y tipos
    - Función de *deserialización* genérica

Componentes de un s. de *serialización*

- No sólo define un *wire protocol*, además puede incluir:
  - IDL (*Interface Definition Language*) y API para la *serialización*
- Lenguaje IDL para especificar datos a transmitir/almacenar
  - Permite definir datos con independencia de la plataforma
    - Usando, habitualmente, lenguaje específico (véase sección sobre RPC)
  - *Compilador IDL*: a partir de especificación genera tipos/clases nativos
  - Aplicación usa tipos nativos generados y API para (de)serializar
- API hipotético para *serialización*
  - *Encode(dato, tipo)* → *buffer*
  - *Decode(buffer, tipo)* → *dato*
    - Si información de tipos/campos explícita: *Decode(buffer)*
- Puede estar integrado en entorno de RPC/RMI
  - ¡Buenas noticias!: Programador no realiza *serialización*
  - Código generado usa automáticamente API de *serialización*

Ejemplos de formatos de *serialización*

- XDR (RFC 1832): binario, info. implícita campos y tipos
- CDR de CORBA: binario, info. implícita campos y tipos
- Soluciones basadas XML: texto, inf. explícita campos y tipos
  - Info de tipos mediante referencia a XML Schema
- JSON: texto, info. explícita campos y tipos
- *Protocol Buffers* (Google): binario, no explícita campos y tipos
  - Pero sí viaja ID único y longitud de cada campo con datos
    - Facilita cambios incrementales en protocolo
- *Java Serialization*: binario, info. explícita campos y tipos
  - Info de campos y tipos mezclada con datos; no requiere IDL
- Muchos otros: ASN.1, Apache Thrift, Apache Avro, BSON,…
- *Wikipedia: Comparison data serialization formats*
- Ejemplos: http://laurel.datsi.fi.upm.es/~ssoo/SD.dir/serializacion

XDR

```
struct dato {int id;  string nombre<>; };  // especificación (fichero .x)
```

```
struct dato d;  XDR x;  char buf[TAM];
d.id=1; d.nombre="yo";
xdrmem_create(&x, buf, TAM, XDR_ENCODE);
xdr_dato (&x, &d);            // serializa
write(1, buf, xdr_getpos(&x));
```

```
XDR x; int tam;  char buf[TAM];
struct dato d = {0, NULL};
tam=read(0, buf, TAM);
xdrmem_create(&x, buf, tam, XDR_DECODE);
xdr_dato (&x, &d);            // deserializa
printf("id %d nombre %s\n", d.id, d.nombre);
```

Contenido = 12 bytes: 00 00 00 01 00 00 00 02 'y' 'o' 00 00

JSON (wikipedia)

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    },
    "phoneNumber": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "fax",
            "number": "646 555-4567"
        }
    ]
}
```


JSON (con Javascript)

```
<!DOCTYPE html><html><body><script>
var pers = new Object();
pers.nombre="yo";
pers.tfno=666;
var buf = JSON.stringify(pers);     // Serializa a {"nombre":"yo","tfno":666}
alert(buf);

var p = JSON.parse(buf);       // Deserialización genérica
alert(p.nombre + " " + p.tfno);
</script></body></html>
```


Protocol Buffers (con C++)

```
message Person {
  required int32 id = 1;
  required string name = 2;
  optional string email = 3;
}
```

Especificación (fichero .proto)

```
Person person;
person.set_id(123);
person.set_name("Bob");
person.set_email("bob@example.com");

fstream out("person.pb", ios::out | ios::binary | ios::trunc);
person.SerializeToOstream(&out);
out.close();
```

Serialización a un fichero (C++)

```
Person person;
fstream in("person.pb", ios::in | ios::binary);
if (!person.ParseFromIstream(&in)) {
  cerr << "Failed to parse person.pb." << endl;
  exit(1);
}

cout << "ID: " << person.id() << endl;
cout << "name: " << person.name() << endl;
if (person.has_email()) {
  cout << "e-mail: " << person.email() << endl;
}
```

Deserialización desde un fichero (C++)

Java Serialization

```
public class Dato implements Serializable {
    int id; String nombre; public Dato(int i, String n) {id = i; nombre = n; }}
```

```
class Encode {
    static public void main (String args[]) {
      Dato d = new Dato(1, "yo");
      try {  ObjectOutputStream o = new ObjectOutputStream(System.out);
           /* serialización */ o.writeObject(d); o.close(); }
      catch (java.io.IOException e) { System.err.println("Error serializando");}}}
```

```
class Decode {
    static public void main (String args[]) {
      try {  ObjectInputStream i = new ObjectInputStream(System.in);
           /* deserialización genérica */  Dato d = (Dato) i.readObject(); i.close();
           System.out.println(d.id + " " + d.nombre); }
      catch (Exception e) {  System.err.println("Error deserializando");} }}
```

Contenido = ¡69 B!; http://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html

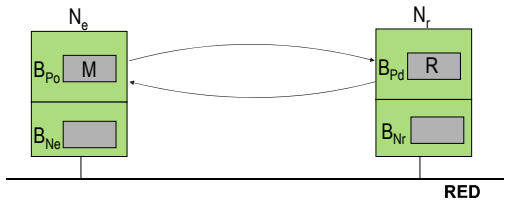
El precio de un entero (a=150)

Definición	Formato	Contenido
`message Test1 { required int32 a = 1; }`	`08 96 01`	https://developers.google.com/protocol-buffers/docs/encoding
struct dato {int a;};	XDR	00 00 96 00
var d=new Object(); d.a=150;	JSON	{"a":150}
public class D implements Serializable {int a;}	Java	30B

Grado de sincronía y *buffering*

- P_o envía M a P_d: copia entre *buffers* de procesos: $B_{Po} \rightarrow B_{Pd}$
  - Además puede haber *buffers* en nodo emisor B_{Ne} y/o receptor B_{Nr}
    - Minimizar copias entre *buffers* (ideal: *zero copy*)
- De menor a mayor grado de sincronía
  1. Envío devuelve control inmediatamente
     - No requiere B_{Ne} pero P_o no puede reutilizar B_{Po} hasta que sea seguro
       - Fin de operación o mensaje copiado en algún *buffer* (B_{Ne} o B_{Nr})
     - Requiere operación para comprobar si ya se puede reutilizar
  2. Envío devuelve control después de $B_{Po} \rightarrow B_{Ne}$
     - P_o puede reutilizar B_{Po}, pero posible bloqueo si B_{Ne} lleno
  3. Envío devuelve control cuando M llega a nodo receptor (B_{Nr})
     - No requiere B_{Ne}; ACK de N_r a N_e
  4. Envío devuelve control cuando M llega a P_d (B_{Pd})
     - No requiere B_{Ne} ni B_{Nr}; ACK de N_r a N_e
  5. Envío devuelve control cuando P_d tiene respuesta
     - No requiere B_{Ne} ni B_{Nr}: $B_{Po} \leftrightarrow B_{Pd}$; respuesta sirve de ACK

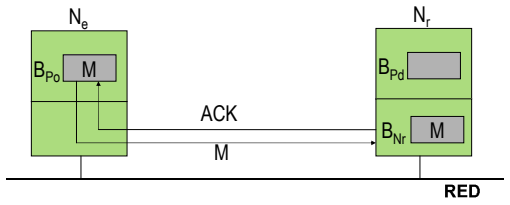
Posibles *buffers* en comunicación



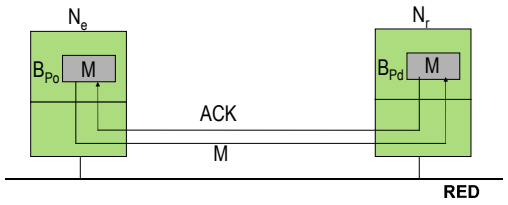
Retorno inmediato

Retorno después de copia local

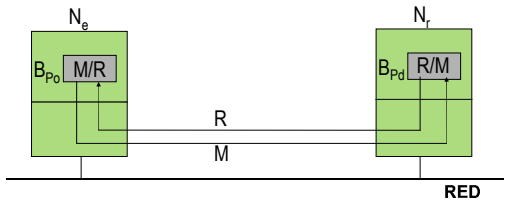
Retorno después de llegada



Retorno después de recepción



Retorno después de respuesta



Modo de operación en recepción

- Recepción generalmente bloqueante
- Opción no bloqueante: retorna si no hay datos
- Opción asíncrona:
  - Especifica *buffer* donde se almacenará el mensaje y
  - Retorna inmediatamente
  - S. comunicaciones realiza recepción mientras proceso ejecuta
- Espera temporizada: se bloquea un tiempo máximo
- Espera múltiple: espera por varias fuentes de datos

Sockets: grado de sincronía y *buffering*

- Modo de operación de envío tipo 2
  - Retorno después de copia local con bloqueo si *buffer* local lleno
  - *Buffer* reservado por SO
- Si aplicación no quiere bloquearse en envío:
  - Usar modo no bloqueante en descriptor socket: error si *buffer* lleno
  - Usar *select/poll/epoll* para comprobar que envío no bloquea
  - Usar E/S asíncrona (*aio_write*): modo de envío tipo 1
- Modo de operación de recepción bloqueante
- Si aplicación no quiere bloquearse en recepción:
  - Usar modo no bloqueante en descriptor socket: error si *buffer* vacío
  - Usar *select/poll/epoll* para comprobar que hay datos que recibir
  - Usar E/S asíncrona (*aio_read*)
- Espera múltiple temporizada mediante *select/poll/epoll*

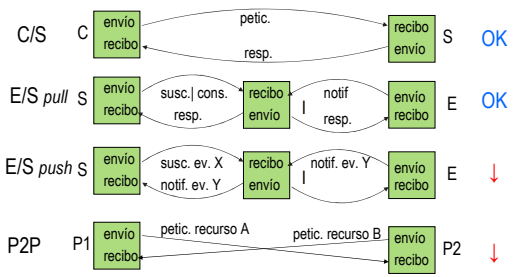
MPI: grado de sincronía y *buffering*

- `MPI_Send` Al retornar emisor puede usar su *buffer* (B_{Po})
  - Modo de operación dependiente de implementación: 2,3 o 4
- `MPI_Bsend` Modo 2 (retorno después de copia local)
  - Aplicación reserva y proporciona a sistema B_{Ne} de tamaño suficiente
- `MPI_Ssend` Modo 4 (retorno en recepción)
- `MPI_Rsend` Emisor sabe que receptor está listo para recibir
- `MPI_SendRecv` Modo 5 (retorno después de respuesta)
- `MPI_I...` Envío devuelve control inmediatamente (Modo 1)
  - Comprobar/esperar *buffer* se puede reutilizar (MPI_TEST|MPI_WAIT)
  - Varias primitivas dependiendo de cuándo se puede reutilizar:
    - `MPI_Isend MPI_Ibsend MPI_Issend MPI_Irsend`
- `MPI_Recv` Recepción bloqueante
- `MPI_Irecv` Recepción asíncrona

Adecuación a arquitecturas del SD

- Paso de mensajes adecuado para cualquier arquitectura
  - Pero cuidado con su asimetría: uno envía y otro recibe
- Cliente/servidor: su asimetría encaja con la del paso mensajes
  - Cliente: envía petición y recibe respuesta
  - Servidor: recibe petición y envía respuesta
- Editor/subscriptor: su asimetría no siempre encaja con p. mens.
  - Si *pull* con intermediario I: buen encaje
    - Su|Ed envían ops. a I y reciben respuestas de I → I siempre pasivo
  - Si *push* con intermediario I: encaje problemático
    - Su envía suscripción(evento X) y espera confirmación de I
    - Pero justo antes I envía notificación de evento Y a Su
    - Soluciones: uso de múltiples puertos y concurrencia en subscriptor
- P2P: arquitectura simétrica
  - ¿Quién envía y quién recibe?

Paso de mensajes y arquitecturas



Índice

- Introducción
- Paso de mensajes
  - Comunicación punto a punto
  - Comunicación de grupo
  - Sistemas de colas de mensajes (MOM)
- Llamadas a procedimientos remotos (RPC)
  - Sun RPC
- Invocación de métodos remotos (RMI)
  - Java RMI y CORBA

Multidifusión: comunicación de grupo

Destino de mensaje → grupo de procesos
- Envío/recepción especifican dirección de grupos de procesos
- Desacoplamiento espacial

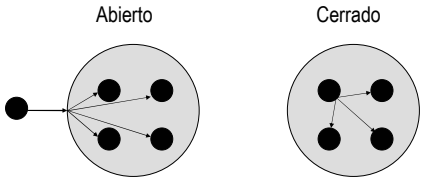
Trabajo seminal: ISIS (posteriores Horus, Ensemble, JGroups)

Adecuación a arquitectura del SD:
- Cliente-servidor replicado
  - Facilita actualizaciones múltiples
- Modelo editor/subscriptor
  - Envío de notificaciones (p.e. 1 grupo/tema)
- Arquitecturas para CD
  - Operaciones colectivas en proc. paralelo (pueden incluir cálculos)

Implementación depende de si red tiene *multicast* (*IP-multicast*)
- Si no, se implementa enviando *N* mensajes

Un proceso puede pertenecer a varios grupos (grupos solapados)

Grupo abierto *versus* cerrado



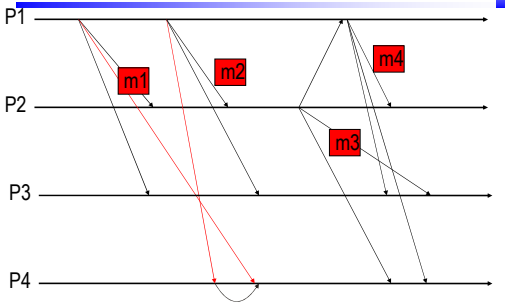
Proceso externo puede mandar mensaje a grupo
Suele usarse para datos o servicios replicados

Sólo procesos del grupo pueden mandar mensajes
Suele usarse en procesamiento paralelo

Aspectos de diseño de com. de grupo

- Atomicidad: o reciben el mensaje o ninguno
  - Con unidifusión fiable (TCP): en medio, se puede caer emisor
  - Con multicast IP: pérdida de mensajes
- Orden de recepción de los mensajes
  - FIFO: mensajes de misma fuente llegan en orden de envío
    - No garantía sobre mensajes de distintos emisores
  - Causal: entrega respeta relación "causa-efecto"
    - Si no hay relación, no garantiza ningún orden de entrega
  - Total: Todos los mensajes recibidos en mismo orden por todos
- El grupo suele tener carácter dinámico
  - Se pueden incorporar y retirar procesos del grupo
  - Gestión de pertenencia debe coordinarse con la comunicación
    - Propiedad denominada "*Virtual Synchrony*"

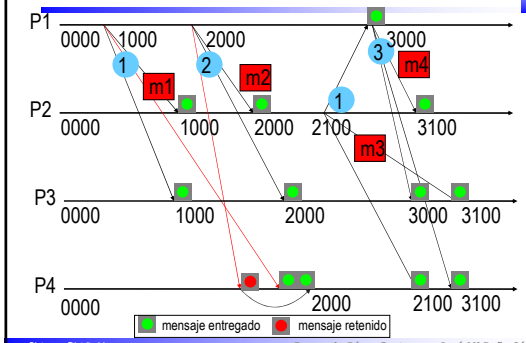
Orden FIFO



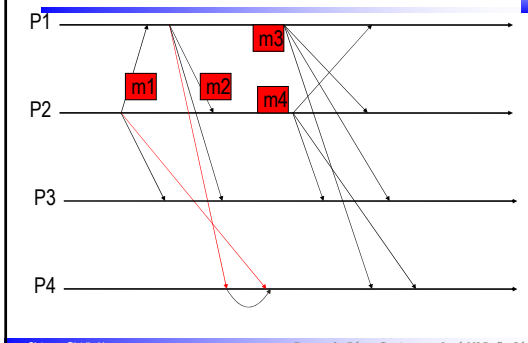
Mantenimiento de orden FIFO

- Nodo Ni almacena vector de contadores Vi (1 posición/nodo)
  - Vi[i] : nº último mensaje enviado por Ni
  - Vi[j] (j≠i): nº último mensaje de Nj recibido por Ni
- Ni envía mensaje M que incluye contador Cm:
  - Vi[i]++; Cm=Vi[i]
- Nj recibe mensaje M de Ni
  - No se entrega M si Cm > Vj[i] + 1
    - No han llegado todavía mensajes previos de Ni
    - Retenido hasta que lleguen mensajes de Ni que faltan [Vj[i] + 1, Cm)
  - En entrega: Vj[i]=Cm

Mantenimiento de orden FIFO



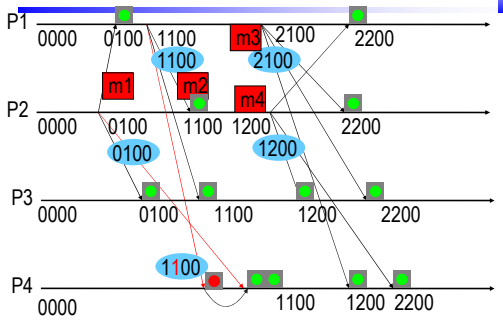
Orden causal



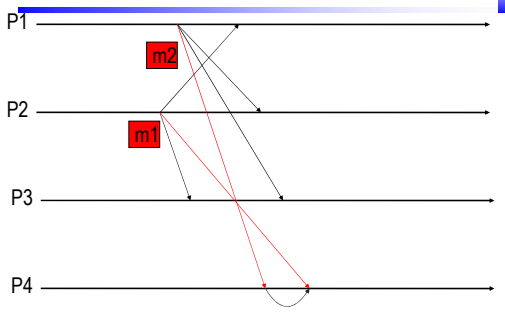
Mantenimiento de orden causal

- Nodo Ni almacena vector de contadores Vi (1 posición/nodo)
  - Vi[i] : nº último mensaje enviado por Ni
  - Vi[j] (j≠i): nº último mensaje de Nj recibido por Ni
- Ni envía mensaje M que incluye vector Vm:
  - Vi[i]++; Vm = Vi
- Nj recibe mensaje M de Ni: No se entrega M a Nj si
  - O bien Vm[i] > Vj[i] + 1
    - No han llegado todavía mensajes previos de Ni (FIFO ⊂ causal)
    - Retenido hasta que lleguen mensajes de Ni que faltan [Vj[i] + 1, Vm[i])
  - O bien ∃k (k≠i) tal que Vm[k] > Vj[k]
    - No han llegado todavía mensajes de Nk que ya ha recibido Ni
    - Retenido hasta que lleguen mensajes de Nk que faltan [Vj[k] +1 , Vm[k]]
  - En entrega: Vj[i]=Vm[i]
- Basado en vectores de relojes lógicos (tema “sincronización”)

Mantenimiento de orden causal



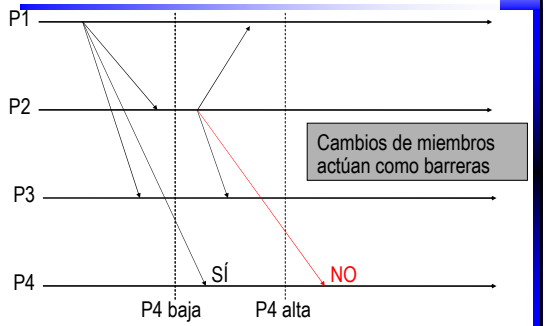
Orden total



Mantenimiento de orden total

- Por simplicidad, sólo solución basada en secuenciador S
  - S "cuello de botella" y punto único de fallo
- Proceso S en el sistema asigna número único a cada mensaje
  - S gestiona contador creciente Cs para cada grupo
- Ni envía mensaje de datos M: a todos miembros grupo G + S
  - Mensaje de datos M no incluye contador; sólo algún tipo de ID de M
- Recepción de M en S:
  - Cs++; asigna Cs a M
  - Envía a miembros G mensaje especial Ms = {ID de M, Cs}
- Nodo Ni incluye Ci: nº próximo mensaje Ms que espera recibir
- Recepción de M en Ni: siempre se retiene
- Recepción de Ms = {ID de M, Cs} en Ni:
  - Ms retenido hasta que recibido M y Cs == Ci → entrega de M

Sincronía virtual



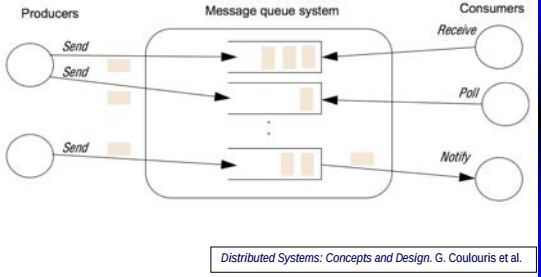
Índice

- Introducción
- Paso de mensajes
  - Comunicación punto a punto
  - Comunicación de grupo
  - Sistemas de colas de mensajes (MOM)
- Llamadas a procedimientos remotos (RPC)
  - Sun RPC
- Invocación de métodos remotos (RMI)
  - Java RMI y CORBA

MOM – Sistemas de colas de mensajes

- *Message-oriented middleware*: *WebSphere MQ*, MSMQ, *JMS*
  - *AMQP* protocolo estándar para MOM
- Envío/recepción mensajes a colas con comunic. "persistente":
  - Comunicación "convencional"
    - Destinatario debe estar presente cuando se recibe mensaje
  - Comunicación "persistente"
    - No es necesario que proceso receptor esté presente
    - Sistema de comunicación (p.e. red de intermediarios) guarda mensaje
- Comunicación desacoplada en espacio y tiempo
- API típico:
  - SEND: envía mensaje a cola
  - RECEIVE: recibe mensaje de cola (bloqueante)
  - POLL: recibe mensaje de cola (no bloqueante)
  - NOTIFY: proceso pide ser notificado cuando llegue mensaje a cola

Sistema de colas de mensajes

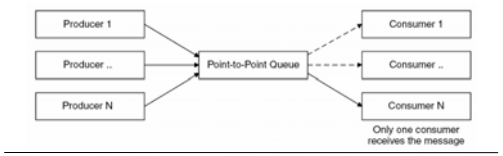


Distributed Systems: Concepts and Design. G. Coulouris et al.

MOM – Sistemas de colas de mensajes

- 2 modelos de comunicación habituales:
  - Basado en colas punto-a-punto
  - Basado en temas editor/subscriptor
- Adecuación a arquitecturas de SD
  - C/S: punto-a-punto → multi-servidor con reparto automático de carga
  - Ed/Su: modelo basado en temas; NOTIFY permite esquema *push*
- Características avanzadas habituales:
  - Filtrado de mensajes: receptor selecciona en cuáles está interesado
    - por propiedades, por contenido,...
    - Puede usarse como filtro por contenido en arquit. Ed./Su.
  - Mensajería con transacciones
  - Transformaciones de mensajes
- Apropiado para integración de aplicaciones de empresa (EAI)

Punto-a-punto vs. editor/subscriptor



Message-Oriented Middleware. Edward Curry

Sistemas Distribuidos

RPC: Llamada a procedimiento remoto

Índice

- Introducción
- Paso de mensajes
  - Comunicación punto a punto
  - Comunicación de grupo
  - Sistemas de cola de mensajes (MOM)
- Llamadas a procedimientos remotos (RPC)
- Invocación de métodos remotos (RMI)
  - Java RMI y CORBA

Provisión de servicios en S. no Dist.

Aplicación

```
int main(...) {
   ....
   r=op1(p, q, ...);
   .........
}
```

Biblioteca

```
t1 op1(ta a, tb b, ...) {
   .....
   return r1;
}
t2 op2(tx x, ty y, ...) {
   .....
   return r2;
}
...........
```


Provisión de servicios (C/S) en S. Dist.

Cliente

```
int main(...) {
  Mensaje msj,resp;
  ..........
  dir_srv=busca(IDservicio);
  msj.op=OP1;
  msj.arg1=p;
  msj.arg2=q;
  ...........
  envío(msj,dir_srv);
  recepción(&resp, NULL);
  r=resp.r;
  .........
}
```

Servidor

```
int main(...) {
  Mensaje msj,resp;
  .........
  alta(IDservicio,dir_srv);
  while (TRUE) {
    recepción(&msj,&dir_clie);
    switch(msj.op) {
    case OP1:
      resp.r=op1(msj.arg1,...);
    case OP2:
      resp.r=op2(msj.arg1,...);
    .........
    envío(resp,dir_clie);
  }
}
t1 op1(ta a, tb b, ...) {
}
...........
```


Cliente-servidor con paso de mensajes

- Provisión de servicios en SD tiene mayor complejidad
- Programador tiene que ocuparse de aspectos como:
  - Operaciones de mensajería
  - *Marshalling/Unmarshalling*
    - Con conversión de formato si sistema de comunicación no lo hace
  - Realizar *binding*
  - Gestión de esquema de concurrencia elegido
    - *Threads*/procesos dinámicos, estáticos o con umbrales
  - Asegurar semántica de comportamiento ante fallos
  - Gestión de conexiones (si se usa esquema con conexión)
    - 1 conexión/petición vs. *N* peticiones de cliente usan misma conexión
  - Si comunicación no fiable, garantizar envío correcto de mensajes
  - Si limitación en tamaño de mensajes, fragmentación y compactación

Fundamento de las RPC

- Código añadido a provisión de servicios en SD
  - Es independiente de la implementación del cliente y del servidor
  - Sólo depende de la interfaz de servicio
  - Puede generarse automáticamente a partir de la misma
- Objetivo de las RPC
  - Provisión de servicios igual que en sistema no distribuido
  - Sólo hay que programar bibliotecas de servicio y aplicaciones
  - Código restante generado automáticamente
  - Lograr semántica convencional de llamadas a procedimiento en SD

Provisión de servicios en SD con RPC

Aplicación

```
int main(...) {
   ....
   r=op1(p, q, ...);
   .........
}
```

```
init() {
  dir_srv=busca(IDservicio);
}
t1 op1(ta a, tb b, ...) {
  Mensaje msj,resp;
  msj.op=OP1;
  msj.arg1=a;
  msj.arg2=b;
  envío(msj,dir_srv);
  recepción(&resp, NULL);
  return resp.r;
}
```

Biblioteca

```
int main(...) {
  Mensaje msj,resp;
  .........
  alta(IDservicio,dir_srv);
  while (TRUE) {
    recepción(&msj,&dir_clie);
    switch(msj.op) {
    case OP1:
      resp.r=op1(msj.arg1,...);
    case OP2:
      resp.r=op2(msj.arg1,...);
    .........
    envío(resp,dir_clie);
  }
}
```

```
t1 op1(ta a, tb b, ...) {
}
..........
```

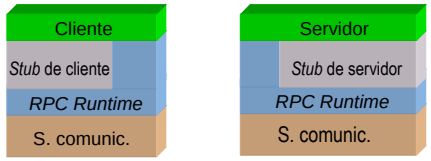

Historia y evolución de las RPC

- Primeras ideas: White 1975; White se enrola en Xerox.
- Desarrollo en Xerox: primer sistema de RPC *Courier* (1981)
- Artículo clásico de Birrel y Nelson en 1984.
- Sun: *Open Network Computing* (1985)
  - RPC de Sun/ONC es la base para servicios (NFS o NIS)
- Apollo (competidora de Sun) crea NCA: RPC de NCA
  - HP compra Apollo; HP miembro de Open Group
- Open Group: DCE (*Distributed Computing Environment* 1990)
  - RPC de DCE basada en NCA
  - RPC de Microsoft (sustento de DCOM) basada en RPC de DCE
- RPC de Sun/ONC vs. RPC de DCE
  - RPC de Sun/ONC menos sofisticada pero más extendida

Componentes de un sistema de RPC

- En tiempo de construcción del programa:
  - Generador automático de código: Compilador de IDL
    - Definición de interfaz de servicio → Resguardos
    - Heterogeneidad: disponibles para múltiples lenguajes
- En tiempo de ejecución del programa:
  - Resguardos (*stubs*)
    - Módulos que se incluyen en cliente y en servidor
    - Ocultan comunicación dando abstracción de llamada a procedimiento
  - *Runtime* de RPC
    - Capa que proporciona servicios que dan soporte a RPC
      - *Binding*, conversión datos, autenticación, comportamiento ante fallos, ...
    - Uso normalmente por resguardos pero también por aplicación/biblioteca
  - *Binder*: Localización de servicios; alternativas ya presentadas
    - Ámbito no global: ID servicio + dir. servidor (Sun/ONC RPC)
    - Ámbito global: ID servicio
      - Uso sólo de *binder* global vs. *binder* global + *binders* locales (DCE RPC)

Pila de comunicación en RPC



Resguardo del cliente (aka *proxy*)

- Conjunto de funciones enlazadas con la aplicación cliente
  - Ofrece la misma API que el servicio
  - Usa servicios de *runtime* de RPC
- Tareas realizadas por una función del resguardo de cliente:
  - Localiza al servidor usando *binder* (identificador del servicio)
    - Si se usa BG + BL → doble consulta
  - Control de conexiones, si se usa comunicación que las requiera
  - Empaqueta/convierte id. función y parámetros (*marshalling*)
  - Envía el mensaje al servidor
  - Espera la recepción del mensaje
  - Extrae/convierte resultados (*unmarshalling*) y los devuelve
    - Si mensaje info. explícita de tipos, conversión independiente del servicio

Resguardo del servidor (aka *skeleton*)

- Programa que se enlaza en servidor con biblioteca de servicio
  - Usa servicios de *runtime* de RPC
- Tareas realizadas por resguardo de servidor:
  - Registra el servicio en *binder*
    - Si se usa BG + BL → doble registro
  - Si servidor concurrente, gestiona procesos/*threads* de servicio
  - Si comunicación con conexión, control/gestión de las conexiones
  - Ejecuta bucle de espera de mensajes
  - Recibe petición
  - Desempaqueta/convierte mensaje (*unmarshalling*)
    - Si mensaje info. explícita de tipos, conversión independiente del servicio
  - Determina qué función invocar
  - Invoca función con argumentos
  - Empaqueta resultados (*marshalling*)
  - Envía mensaje a resguardo del cliente

Características de las RPC

- Modo de operación con bloqueo hasta respuesta
  - También asíncrono
    - P.ej. Extensión de Microsoft a RPC de DCE
- Normalmente comunicación desde cliente a servidor
  - En algunos sistemas *Callback* RPC: llamadas de servidor a cliente
- Normalmente implican comunicación uno a uno
  - En algunos sistemas *Multicast* y *Broadcast* RPC (p. ej. RPC de Sun)
- Comportamiento ante fallos garantizado por RPC
  - P.ej. RPC de DCE asegura por defecto como mucho una vez
    - Además permite marcar una función de servicio como idempotente
- Si servidor concurrente
  - Funciones de servicio pueden requerir mecanismos de sincronización

Adecuación a las arquitecturas del SD

- Orientadas a modelo cliente-servidor
- Arquitectura editor/subscriptor
  - Adecuadas si modelo *pull* con intermediario I
    - Ed y Su invocan RPCs de intermediario
      - Ed y Su resguardo de cliente; I resguardo de servidor
  - Modelo *push* más problemático
    - Su invoca RPC del servicio de I para suscripción:
      - Su (y Ed) resguardo de cliente; I resguardo de servidor
    - Pero I invoca RPC del servicio de Su para notificación de eventos
      - I también resguardo de cliente; Su también resguardo de servidor
- Solución habitual *Callback* RPC:
  - Intermediario proporciona función de registro a subscriptor
    - Incluye parámetro que identifica servicio de notificación en subscriptor
  - Intermediario almacena lista de ID de servicio de suscriptores
  - Cuando I recibe evento de Ed, invoca *callback* RPC de subscriptores

Lenguajes de definición de interfaces

- Los resguardos se generan a partir de la interfaz de servicio
  - ¿Cómo se define esta interfaz?
- Deben de poder definirse entidades tales como:
  - Un tipo interfaz de servicio con un número de versión asociado
  - Prototipos de funciones, constantes, tipos de datos, …
    - Tipo de parámetros: de entrada, de salida o de entrada/salida
  - Directivas específicas (por ejemplo, si una función es idempotente)
  - Sólo definiciones; nunca implementación
- Alternativa: Uso de lenguaje específico vs. uno convencional
  - Permite definición neutral de interfaces
  - Supera limitaciones en expresividad de lenguajes convencionales
  - Pero hay que aprenderlo...
- Procesamiento: Compilador IDL (por ejemplo para C)
  - fichero IDL→ fichero .h + resguardo cliente + resguardo servidor

Transparencia de las RPC

- ¿Una invocación remota se comporta igual que una local?
- Existen varias diferencias:
  - Puede producir un error si no puede comunicarse con servidor.
    - ¿Cómo notificarlo a la aplicación?
    - Generalmente, usando excepciones
  - No puede haber variables globales en el servicio accesibles al cliente
  - Los parámetros no pueden pasarse por referencia
    - Uso de copia y restauración: tratamiento de parámetro de entrada/salida
      - Resguardo de cliente envía copia al resguardo servidor
      - Resguardo servidor lo pasa por referencia a función real, que lo modifica
      - Resguardo de servidor envía a resguardo cliente nuevo valor
      - Resguardo cliente lo establece como nuevo valor
  - Problema: parámetro de tipo estructura con punteros internos (listas)
    - Algunos sistemas las prohíben
    - Otros, como RPC de Sun, las permiten
      - el resguardo del emisor las "aplana" y el del receptor las reconstruye

RPC de Sun/ONC

- Estándar (RFC 1831) y disponible en muchas plataformas
- Junto con XDR (*External Data Representation*, RFC 1832)
  - Define IDL y formato de representación externo (binario e implícito)
- Independiente de nivel de transporte subyacente (TCP o UDP)
- Distintos tipos de autenticación de cliente
  - Sin autenticación
  - Modelo UNIX (uid:gid)
  - Basada en cifrado (DES o Kerberos)
    - Secure RPC (base de Secure NFS)
- Comportamiento ante fallos:
  - Si se implementa sobre TCP: como mucho una vez
  - Sobre UDP, *runtime* de RPC retransmite petición si no respuesta
    - Permite activar una caché de respuestas en servidor

Consultar guía de programación RPC en página de la asignatura

Direccionamiento en RPC de Sun/ONC

- Servicio identificado por nº de programa (32 bits) + versión
  - De 0 a 1FFFFFFF reservados por Sun
- ID de servicio no tiene ámbito global (ID + dir. máquina)
  - 1 *binder* (*portmap*|*rpcbind*) por máquina en puerto conocido (111)
- Modo de operación:
  - Resguardo de servidor se da de alta en *binder* local:
    - nº de programa + versión + protocolo + puerto (efímero)
  - Cliente especifica: máquina + nº de programa + versión + protocolo
  - Resguardo de cliente contacta con *binder* de máquina
    - nº de programa + versión + protocolo → puerto
- Herramienta *rpcinfo* permite contactar con *binder* para:
  - obtener lista de servicios dados de alta, ejecutar el procedimiento nulo de un servicio para comprobar si servidor arrancado, etc.

IDL (XDR) de RPC de Sun/ONC

- Extensión de C. Permite definir:
  - Un programa (interfaz) y su nº de versión
  - Constantes, tipos y funciones (hay que asignarles un número)
    - Restricción: función con un 1 solo argumento y 1 solo valor de retorno
- Compilador *rpcgen*:
  - Además de .h y resguardos, genera fichero de *marshalling* (.xdr)
- Algunos tipos específicos de XDR (además de los de C):
  - Vectores de tamaño variable:
    - tipo_elem vector<>; tipo_elem vector<tam_max>;
  - *String*
  - *Union* distinta de la de C: registro variante

```
union nombre_tipo switch (int discriminante) {
    case valor1: declaración1;
    case valor2: declaración2;
    default:  declaraciónN; }
```


Sistemas Distribuidos

RMI: Invocación de método remoto

- **Java RMI**
- **CORBA**

Índice

- Introducción
- Paso de mensajes
  - Comunicación punto a punto
  - Comunicación de grupo
- Llamadas a procedimientos remotos (RPC)
- Invocación de métodos remotos (RMI)
  - Java RMI y CORBA

Modelo de objetos en sis. distribuidos

- Sistemas distribuidos.
  - Aplicaciones inherentemente distribuidas.
  - Se caracterizan por su complejidad.
- Sistemas orientados a objetos.
  - Más cercanos al lenguaje natural.
  - Facilitan el diseño y la programación.

Objetos-Distribuidos

Características:

- Uso de un *Middleware*: Nivel de abstracción para la comunicación de los objetos distribuidos. Oculta:
  - Localización de objetos.
  - Protocolos de comunicación.
  - Hardware de computadora.
  - Sistemas Operativos.
- Modelo de objetos distribuidos: Describe los aspectos del paradigma de objetos que es aceptado por la tecnología: Herencia, Interfaces, Excepciones, Polimorfismo, ...
- Recogida de basura (*Garbage Collection*): Determina los objetos que no están siendo usados para liberar recursos.

Ventajas respecto a paso de mensajes

- Paso de mensajes:
  - Procesos fuertemente acoplados
  - Paradigma orientado a datos: No adecuado para aplicaciones muy complejas que impliquen un gran número de peticiones y respuestas entremezcladas.
- Paradigma de objetos distribuidos
  - Mayor abstracción
  - Paradigma orientado a acciones:
    - Hace hincapié en la invocación de las operaciones
    - Los datos toman un papel secundario

Ventajas respecto a RPC

- Ventajas derivadas al uso de programación orientada a objetos:
  - Encapsulación
  - Reutilización
  - Modularidad
  - Dinamismo

Objetos distribuidos

- Minimizar las diferencias de programación entre las invocaciones de métodos remotos y las llamadas a métodos locales
- Ocultar las diferencias existentes:
  - Tratamiento del empaquetamiento de los datos (*marshalling*)
  - Sincronización de los eventos
  - Las diferencias deben quedar ocultas en la arquitectura

Sistemas Distribuidos

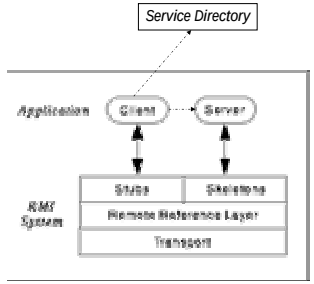
Java RMI

Java RMI

Java: *Write Once, Run Anywhere*

Java RMI extiende el modelo Java para la filosofía *"Run Everywhere"*

Arquitectura de Java RMI



Arquitectura de Java RMI

- *Nivel de resguardo/esqueleto* (*proxy/skeleton*) que se encarga del aplanamiento (serialización) de los parámetros
  - *proxy*: resguardo local. Cuando un cliente realiza una invocación remota, en realidad hace una invocación de un método del resguardo local.
  - Esqueleto (*skeleton*): recibe las peticiones de los clientes, realiza la invocación del método y devuelve los resultados.
- *Nivel de gestión de referencias remotas*: interpreta y gestiona las referencias a los objetos de servicio remoto
- *Nivel de transporte*: se encarga de las comunicaciones y de establecer las conexiones necesarias. Basada en TCP (orientada a conexión)

Servicio de directorios

- Se pueden utilizar diferentes servicios de directorios para registrar un objeto distribuido
  - Ejemplo: JNDI (Java Naming and Directory Interface)
- El registro RMI, *rmiregistry*, es un servicio de directorios sencillo proporcionado por SDK (Java Software Development Kit)

Java RMI

El soporte para RMI en Java está basado en las interfaces y clases definidas en los paquetes:

- *java.rmi*
- *java.rmi.server*

Características de Java RMI:

- Se basa en una interfaz remota Java (hereda de la clase Java *Remote*).
- Es necesario tratar mayor número de excepciones que en el caso de invocación de métodos locales
  - Errores en la comunicación entre los procesos (fallos de acceso, fallos de conexión)
  - Problemas asociados a la invocación de métodos remotos (no encontrar el objeto, el resguardo o el esqueleto)
  - Los métodos deben especificar la excepción *RemoteException*.

Ejemplo de interfaz remota Java

```
public interface InterfazEj extends
  java.rmi.Remote
{
  public String metodoEj1()
     throws java.rmi.RemoteException;
  public int metodoEj2(float param)
     throws java.rmi.RemoteException;
}
```


Java RMI

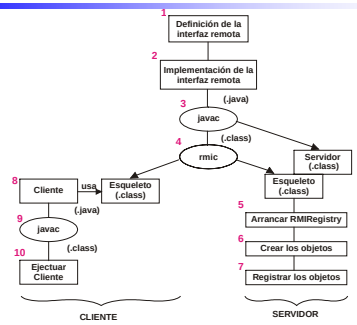
Localización de objetos remotos:

- Servidor de nombres: *java.rmi.Naming*

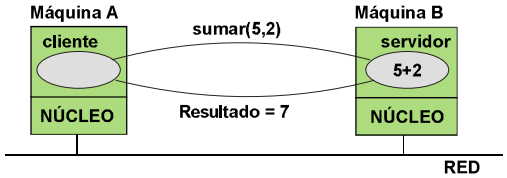
Ejemplo:

```
Cuenta cnt = new CuentaImpl();
String url = "rmi://java.Sun.COM/cuenta";
// enlazamos una url a un objeto remoto
java.rmi.Naming.bind(url, cnt);
     ....
// búsqueda de la cuenta
cnt=(Cuenta)java.rmi.Naming.lookup(url);
```


Desarrollo de Aplicaciones RMI



Ejemplo



Modelización de la interfaz remota (Sumador)

```
public interface Sumador extends java.rmi.Remote
{

  public int sumar(int a, int b)
       throws java.rmi.RemoteException;

}
```


Clase que implementa la interfaz (SumadorImpl)

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class SumadorImpl extends UnicastRemoteObject
  implements Sumador {
   public SumadorImpl(String name) throws RemoteException {
       super();
      try {
       System.out.println("Rebind Object " + name);
           Naming.rebind(name, this);
      } catch (Exception e){
       System.out.println("Exception: " + e.getMessage());
                e.printStackTrace();
      }
  }
   public int sumar (int a, int b) throws RemoteException {
          return a + b;
   }
}
```


Código del servidor (SumadorServer)

```
import java.rmi.*;
import java.rmi.server.*;

public class SumadorServer  {
   public static void main (String args[]) {
      try {
            SumadorImpl misuma = new
                  SumadorImpl("rmi://localhost:1099"
                  +"/MiSumador");
            } catch(Exception e) {
            System.err.println("System exception" +
  e);
      }
   }
}
```


Registro del servicio

Antes de arrancar el cliente y el servidor, se debe arrancar el programa *rmiregistry* en el servidor para el servicio de nombres. El puerto que utiliza el *rmiregistry* por defecto es el 1099.

```
rmiregistry [port_number]
```

El método *rebind* es utilizado normalmente en lugar del método *bind*, porque garantiza que si un objeto remoto se registró previamente con dicho nombre, el nuevo objeto reemplazará al antiguo.

El método *rebind* almacena en el registro una referencia al objeto con un URL de la forma:

```
rmi://<nombre máquina>:<número puerto>/<nombre
referencia>
```


Código en el cliente (SumadorClient)

```
import java.rmi.registry.*;
import java.rmi.server.*;
import java.rmi.*;
public class SumadorClient {
   public static void main(String args[]){
  int res = 0;
  try {
       System.out.println("Buscando Objeto ");
       Sumador misuma = (Sumador)Naming.lookup("rmi://" +
  args[0] +                          "/" +"MiSumador");
        res = misuma.sumar(5, 2);
       System.out.println("5 + 2 = " + res);
   }
   catch(Exception e){
            System.err.println(" System exception");    }
  System.exit(0);
  }
}
```

Búsqueda

Cualquier programa que quiera instanciar un objeto remoto debe realizar una búsqueda de la siguiente forma:

```
Sumador misuma = (Sumador)Naming.lookup("rmi://" + args[0] +
"/"             +"MiSumador");
```

El método *lookup* devuelve una referencia remota a un objeto que implementa la interfaz remota.

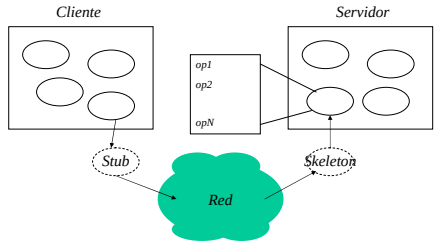
El método *lookup* interactúa con *rmiregistry*.

Pasos

Java RMI:

- Enlace a un nombre: bind(), rebind()
- Encontrar un objeto y obtener su referencia: lookup()
- refObj.nombre_met()

Cuadro general



¿Cómo se ejecuta?

Compilación

- javac Sumador.java
- javac SumadorImpl.java
- javac SumadorClient.java
- javac SumadorServer.java

Generación de los esqueletos

- rmic SumadorImpl

Ejecución del programa de registro de RMI

- rmiregistry <número puerto>
- *Por defecto, en número de puerto es el 1099*

Ejecución del servidor

- java SumadorServer

Ejecución del cliente

- java SumadorCliente <host-del-servidor>

Callback de cliente

Hay escenarios en los que los servidores deben notificar a los clientes la ocurrencia de algún evento. Ejemplo: chat.

- Problema: llamada a método remoto es unidireccional

Posibles soluciones:

- Sondeo (*polling*): Cada cliente realiza un sondeo al servidor, invocando repetidas veces un método, hasta que éste devuelva un valor *true*.
  - Problema: Técnica muy costosa (recursos del sistema)
- *Callback*: Cada cliente interesado en la ocurrencia de un evento se registra a sí mismo con el servidor, de modo que el servidor inicia una invocación de un método remoto del cliente cuando ocurra dicho evento.
  - Las invocaciones de los métodos remotos se convierten en bidireccionales

Extensión de la parte cliente para *callback* de cliente

El cliente debe proporcionar una interfaz remota: Interfaz remota de cliente

```
public interface CallbackClient extends java.rmi.Remote {
    public String notificame (String mensaje) throws
   java.rmi.RemoteException;
}
```

Es necesario implementar la interfaz remota de cliente, de forma análoga a la interfaz de servidor (CallbackClientImpl)

En la clase cliente se debe añadir código para que instancie un objeto de la implementación de la interfaz remota de cliente y que se registre para callback (método implementado por el servidor):

```
CallbackServer cs = (CallbackServer) Naming.lookup(URLregistro);
CallbackClient objCallback = new CallbackClientImpl();
cs.registrarCallback(objCallback);
```

Extensión de la parte servidora para *callback* de cliente

Añadir el método remoto para que el cliente se registre para *callback*

```
public void registrarCallback (CallbackClient
objCallbackClient) throws java.rmi.RemoteException;
```

Se puede proporcionar un método `eliminarRegistroCallback` para poder cancelar el registro

Ambos métodos modifican una estructura común (por ejemplo, un objeto Vector) que contiene referencias a los *callbacks* de clientes. Se utilizan métodos *synchronized* para acceder a la estructura en exclusión mutua.

Descarga dinámica de resguardo

Mecanismo que permite que los clientes obtengan dinámicamente los resguardos necesarios
- Elimina la necesidad de copia de la clase del resguardo en la máquina cliente
- Se transmite bajo demanda desde un servidor web a la máquina cliente (Similar a la descarga de los applets)

El servidor exporta un objeto a través del registro RMI (registro de una referencia remota al objeto mediante nombre simbólico) e indica el URL donde se almacena la clase resguardo.

La clase resguardo descargada no es persistente
- Se libera cuando la sesión del cliente finaliza

Comparativa RMI vs Sockets

Los sockets están más cercanos al sistema operativo, lo que implica una menor sobrecarga de ejecución.

RMI proporciona mayor abstracción, lo que facilita el desarrollo de software. RMI es un buen candidato para el desarrollo de prototipos.

Los sockets suelen ser independientes de plataforma y lenguaje.

Sistemas Distribuidos

Entornos de Objetos Distribuidos

- **CORBA**

OMG

(Object Management Group)

Conjunto de organizaciones que cooperan en la definición de estándares para la interoperabilidad en entornos heteregéneos.

Fundado en 1989, en la actualidad lo componen más de 700 empresas y otros organismos.

OMA

(Object Management Architecture)

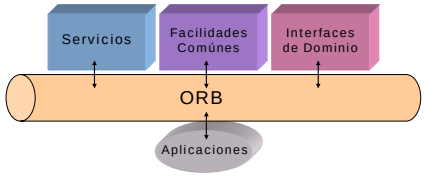
Arquitectura de referencia sobre cual se pueden definir aplicaciones distribuidas sobre un entorno heteregéneo.

CORBA es la tecnología asociada a esta arquitectura genérica.

Formalmente esta dividida en una serie de modelos:
- Modelo de Objetos
- Modelo de Interacción
- ...

OMA

Una aplicación definida sobre OMA está compuesta por una serie de objetos distribuidos que cooperan entre sí. Estos objetos se clasifican en los siguientes grupos:



OMA

Servicios:

Proporcionan funciones elementales necesarias para cualquier tipo de entorno distribuido, independientemente del entorno de aplicación.

Los tipos de funciones proporcionados son cuestiones tales como la resolución de nombres, la notificación asíncrona de eventos o la creación y migración de objetos.

Concede un valor añadido sobre otras tecnologías (por ejemplo RMI).

Están pensados para grandes sistemas.

OMA

Aplicaciones:

El resto de funciones requeridas por una aplicación en concreto. Es el único grupo de objetos que OMG no define, pues está compuesto por los objetos propios de cada caso concreto.

Estos son los objetos que un sistema concreto tiene que desarrollar. El resto (servicios, facilidades) pueden venir dentro del entorno de desarrollo.

OMA

ORB:

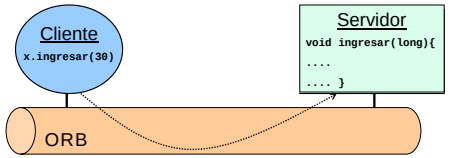
(*Object Request Broker*)

Es el elemento central de la arquitectura. Proporciona las funcionalidades de interconexión entre los objetos distribuidos (servicios, facilidades y objetos de aplicación) que forman una aplicación.

Representa un bus de comunicación entre objetos.

ORB

Para posibilitar la comunicación entre dos objetos cualesquiera de una aplicación se realiza por medio del ORB. El escenario de aplicación elemental es:



IDL de CORBA

(*Interface Definition Language*)

Es el lenguaje mediante el cual se describen los métodos que un determinado objeto del entorno proporciona al resto de elementos del mismo.

```
interface Cuenta
{
  void ingresar(in long cantidad);
  void retirar(in long cantidad);
  long balance();
};
```

IDL de CORBA

Language Mappings:

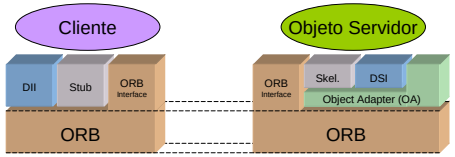
Traducen la definición IDL a un lenguaje de programación como:

- C++,
- Ada,
- COBOL,
- SmallTalk,
- Java,
- ...

Este proceso genera dos fragmentos de código denominados *Stub* y *Skeleton* que representan el código de cliente y servidor respectivamente.

Componentes de un ORB

Arquitectura completa de comunicaciones de CORBA:



Componentes de un ORB

Stub:

Código cliente asociado al objeto remoto con el que se desea interactuar. Simula para el cliente los métodos del objeto remoto, asociando a cada uno de los métodos una serie de funciones que realizan la comunicación con el objeto servidor.

Skeleton:

Código servidor asociado al objeto. Representa el elemento análogo al stub del cliente. Se encarga de simular la petición remota del cliente como una petición local a la implementación real del objeto.

Componentes de un ORB

DII:

(*Dynamic Invocation Interface*)

Alternativa al uso de stubs estáticos que permite que un cliente solicite peticiones a servidores cuyos interfaces se desconocían en fase de compilación.

DSI:

(*Dynamic Skeleton Interface*)

Alternativa dinámica al uso de skeletons estáticos definidos en tiempo de compilación del objeto. Es usado por servidores que durante su ejecución pueden arrancar diferentes objetos que pueden ser desconocidos cuando se compiló el servidor.

Componentes de un ORB

ORB/Interface ORB:

Elemento encargado de (entre otras) las tareas asociadas a la interconexión entre la computadora cliente y servidor, de forma independiente de las arquitecturas hardware y SSOO.

Debido a que tanto clientes como servidores pueden requerir de ciertas funcionalidades del ORB, ambos son capaces de acceder a las mismas por medio de una interfaz.

Las dos principales responsabilidades del ORB son:

Localización de objetos: El cliente desconoce la computadora donde se encuentra el objeto remoto.

Comunicación entre cliente y servidor: De forma independiente de protocolos de comunicación o características de implementación (lenguaje, sistema operativo, ...)

Componentes de un ORB

Adaptador de Objetos:

En este elemento se registran todos los objetos que sirven en un determinado nodo. Es el encargado de mantener todas las referencias de los objetos que sirven en una determinada computadora de forma que cuando llega una petición a un método es capaz de redirigirla al código del skeleton adecuado.

Existen dos tipos de Adaptadores de Objetos especificados por OMG:

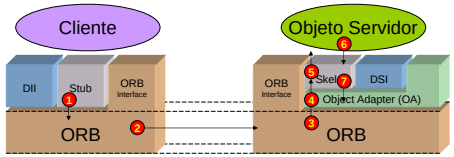
BOA: (Basic Object Adapter).

POA: (Portable Object Adapter).

Comunicación vía CORBA

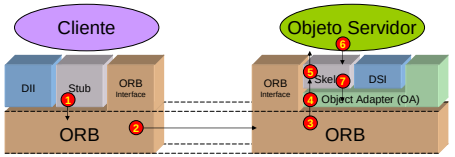
Pasos de una comunicación:

1- El cliente invoca el método asociado en el stub que realiza el proceso de marshalling. (Stub cliente)
2- El stub solicita del ORB la transmisión de la petición hacia el objeto servidor. (ORB cliente)
3- El ORB del servidor toma la petición y la transmite el Adaptador de Objetos asociado, por lo general sólo hay uno. (ORB servidor)



Comunicación vía CORBA

4- El Adaptador de Objetos resuelve cuál es el objeto invocado, y dentro de dicho objeto cuál es el método solicitado (Adaptador de Objetos)
5- El skeleton del servidor realiza el proceso de de-marshalling de los argumentos e invoca a la implementación del objeto. (Skeleton servidor)
6- La implementación del objeto se ejecuta y los resultados y/o parámetros de salida se retornan al skeleton. (Implementación del objeto)
7- El proceso de retorno de los resultados es análogo.



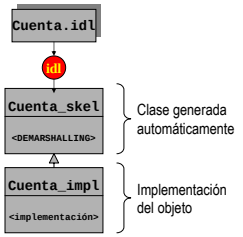
Localización de Objetos

Los objetos de servicio de una aplicación CORBA se encuentran identificados por medio de una referencia única (Identificador de Objeto).

Esta referencia es generada al activar un objeto en el Adaptador de Objetos.

Por medio de esta referencia el ORB es capaz de localizar la computadora y el Adaptador de Objetos donde se encuentra, y éste último es capaz de identificar el objeto concreto dentro del Adaptador.

Implementación del Servidor



La implementación del objeto se diseña como una subclase de la clase generada por el compilador (el *skeleton*) de IDL en base a la definición.

Si el lenguaje usado para la implementación no soporta objetos el mecanismo es diferente.

Servicios CORBA

Conjunto de objetos o grupos de objetos, que proporcionan una serie de funcionalidades elementales. Estos objetos están definidos de forma estándar (interfaces IDL concretas).

Sus especificaciones se encuentran recogidas en los documentos COSS (Common Object Services Specifications).

Los servicios definidos son:

- Servicio de Nombres
- Servicio de Eventos
- Servicio de Ciclo de Vida
- Servicio de Objetos Persistentes
- Servicio de Transacciones
- Servicio de Control de Concurrencia
- Servicio de Relación
- Servicio de Externalización
- Servicio de Consulta
- Servicio de Licencias
- Servicio de Propiedad
- Servicio de Tiempo
- Servicio de Seguridad
- Servicio de Negociación
- Servicio de Colección de Objetos

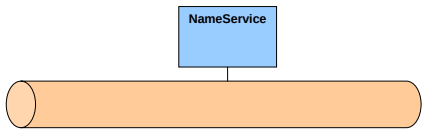
Uso del Servicio de Nombres

Permite asociar un nombre a una referencia de objeto. De esta forma los objetos al activarse pueden darse de alta en el servidor, permitiendo que otros objetos los obtengan su referencia en base a dicho nombre.

Los nombres de los objetos se encuentran organizados en una jerarquía de *contextos de nombre*.

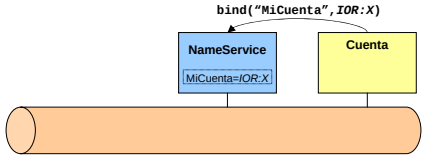
Servicio de Nombres

El Servidos de Nombres, al igual que todo objeto del sistema se encuentra previamente activo.



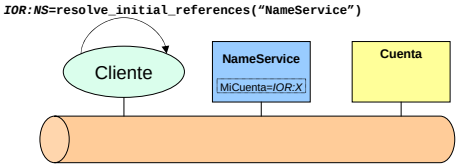
Servicio de Nombres

Un nuevo objeto se arranca y se registra en el servidor de nombres:



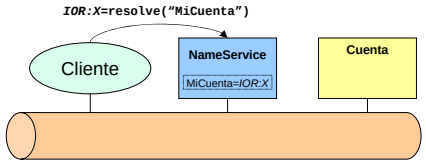
Servicio de Nombres

Un cliente localiza al servidor de nombres. Suele existir una función interna del ORB para localizar al servidor de nombres (**`resolve_initial_references`**) .



Servicio de Nombres

El cliente le pide al servidor de nombres que resuelva el nombre. Así obtiene la referencia al objeto buscado.



Servicio de Negociación

Este servicio también permite obtener referencias a objetos usando otra información:

- Los objetos se exportan en el servidor con una serie de características del servicio que proporcionan.
- Los clientes consultan con el servidor cuáles son los objetos ofertados con una serie de características.

Un cliente que buscase un servicio de impresión podría construir una consulta del tipo:

```
Service=Printer AND
PrinterType=HP AND
OS!=WinNT
```

- A lo cual el servidor le indicará los objetos que existen con dichas características.

Comparativa CORBA vs DCOM

CORBA es un estándar abierto y no propietario.
CORBA proporciona soporte para diversos SO.
CORBA es más completo y flexible.
CORBA da una salida a los *legacy systems*

DCOM esta integrado con la tecnología *MicroSoft*.
DCOM ha tenido una fuerte penetración en el mercado.

Comparativa RMI vs CORBA

CORBA permite una mayor heterogeneidad en el desarrollo de aplicaciones (RMI sólo se puede desarrollar con Java).

CORBA ademas de las funcionalidades de comunicación, proporciona servicios.

RMI funciona sobre CORBA (IIOP).

RMI es mucho más sencillo y cómodo de usar.

RMI permite un desarrollo rápido de prototipos.