Llamada a Procedimientos Remotos (RPC)

Móstoles, 10 de marzo de 2006

Contenidos del Tema

1. Introducción y Conceptos Básicos

2. Protocolos RPC

3. Computación RPC

4. Entornos RPC

Introducción y Conceptos Básicos

Remote Procedure Call (RPC)

- Idea: Enmascarar un sistema distribuido usando una abstracción “transparente” para el dersarrollador.

- Una aplicación puede invocar un procedimiento “situado” en otra máquina accesible desde una red.
  - Llamadas a procedimientos locales (llamantes y llamados se enlazan en tiempo de compilación)
  - Llamadas a procedimientos remotos (llamantes y llamados se enlazan en tiempo de ejecución)

- Desde el punto de vista del programador, ambas llamadas parecen idénticas.

- Un entorno RPC “esconde” los detalles de comunicación.

- Ambas llamadas parecen idénticas, pero ¿son realmente idénticas?

Un poco de historia

- Presentadas por Birrell y Nelson en los años 80 (Xerox Parc).
- Concepto muy atractivo pero no muy bien comprendido.
- Se produjeron intentos prematuros de estandarización...
- ... que terminaron mostrando problemas y carencias serios.
- En los años 90 aparecen entornos de desarrollo RPC con capacidades limitadas (p.e. DCE: Distributed Computing Environment)
- En la actualidad:
  - Énfasis en rendimiento.
  - Nuevos paradigmas orientados a objetos.
  - Preocupación por la “confiabilidad”.

RCPs: Terminología

- El término RCP se utiliza en contextos diferentes:

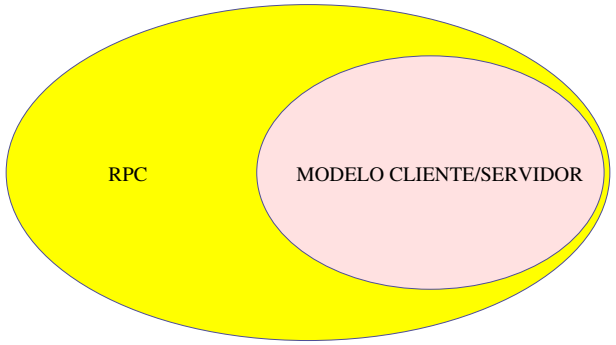
  Protocolos RPC: Protocolos de comunicaciones sobre los que se implementa la funcionalidad RPC.

  Computación RPC: Modelo de computación en el cual un proceso "llamante" ejecuta un procedimiento "llamado" en otro proceso remoto.

  Entorno RPC: Entorno de desarrollo que facilita al desarrollador un conjunto de herramientas que le permiten programar mecanismos RPC en sus aplicaciones de manera transparente (o casi).

RPC y el modelo de computación cliente/servidor

- Modelo cliente/servidor:
  - Los clientes solicitan un servicio.
  - El servidor presta un servicio.
  - Basado en un modelo consumidor/productor.
  - Normalmente un solo servidor para muchos clientes.
- Toda aplicación cliente/servidor puede implementarse a través de RPC
  - Cliente = llamante
  - Servidor = llamado
- RPC permite otras filosofías de computación (p.e. cooperativos)



Protocolos RPC

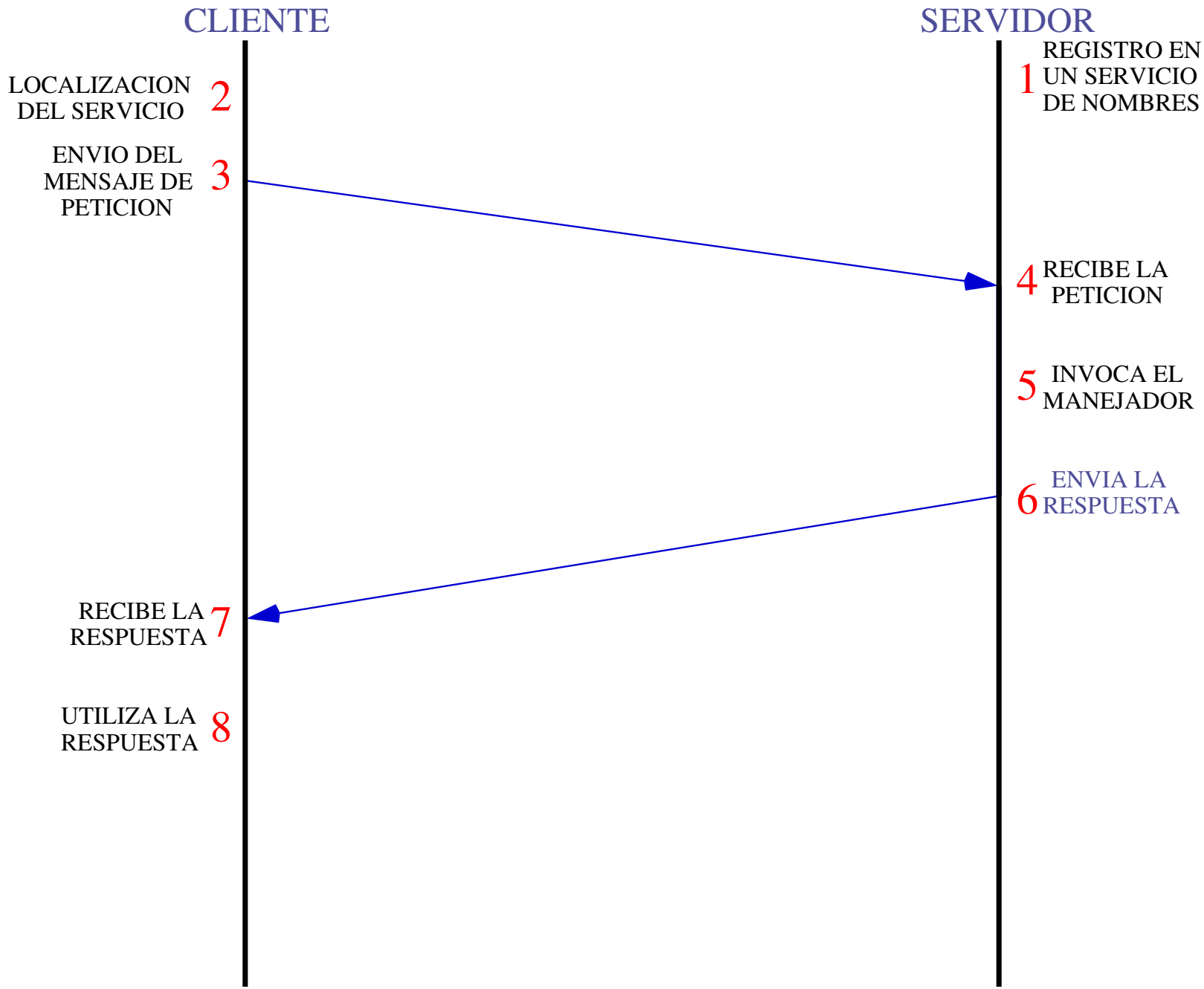
El protocolo RPC: mecanismo básico

- Describe los mecanismos por los que el proceso "cliente" envía una petición al proceso "servidor" y el proceso "servidor" envía la respuesta al "cliente".

- Trata de emular el comportamiento de LPC (*Local Procedure Calls*).
  Semántica **"exactamente una vez"**.

- Se basa en el intercambio de mensajes.

  1. Mensaje cliente ⟶ servidor
     ⟶ Ejecuta tal procedimiento con tales parámetros
  2. Mensaje cliente ⟵ servidor
     ⟵ El resultado de la ejecución es este


CLIENTE
SERVIDOR
REGISTRO EN
1 UN SERVICIO
DE NOMBRES
LOCALIZACION
DEL SERVICIO 2
ENVIO DEL
MENSAJE DE
PETICION 3
4 RECIBE LA
PETICION
5 INVOCA EL
MANEJADOR
ENVIA LA
6 RESPUESTA
RECIBE LA
RESPUESTA 7
UTILIZA LA
RESPUESTA 8

El protocolo RPC: ¿y si algo falla?

- Tipos de fallo
  - Fallos en la comunicación
  - Fallos en los procesos (harware, software, etc.)
- Efectos externos de los fallos
  - Peticiones o respuestas sin sentido
  - Peticiones o respuestas duplicadas
  - Respuestas de error
  - Ninguna respuesta

RPC con fallos en la comunicación

- Asumimos un modelo en el que SOLO hay fallos en la comunicación de mensajes

- Tipos de fallos

  - Errores estructurales (bits erróneos)
    ▶ Se solucionan fácilmente con sumas de comprobación (checksums) ⇒ Error de omisión.
  - Errores de reordenación (los paquetes se desordenan)
    ▶ Se solucionan fácilmente con números de secuencia (como TCP).
  - Errores de omisión (los paquetes se pierden)
    ▶ Temporizadores (timeouts)
    ▶ Asentimientos (ACKs)

- ¿Podemos garantizar semántica “exactamente una vez” con errores de omisión?

Medidas para protegerse frente a la pérdida de mensajes RPC

- **Medida 1:**

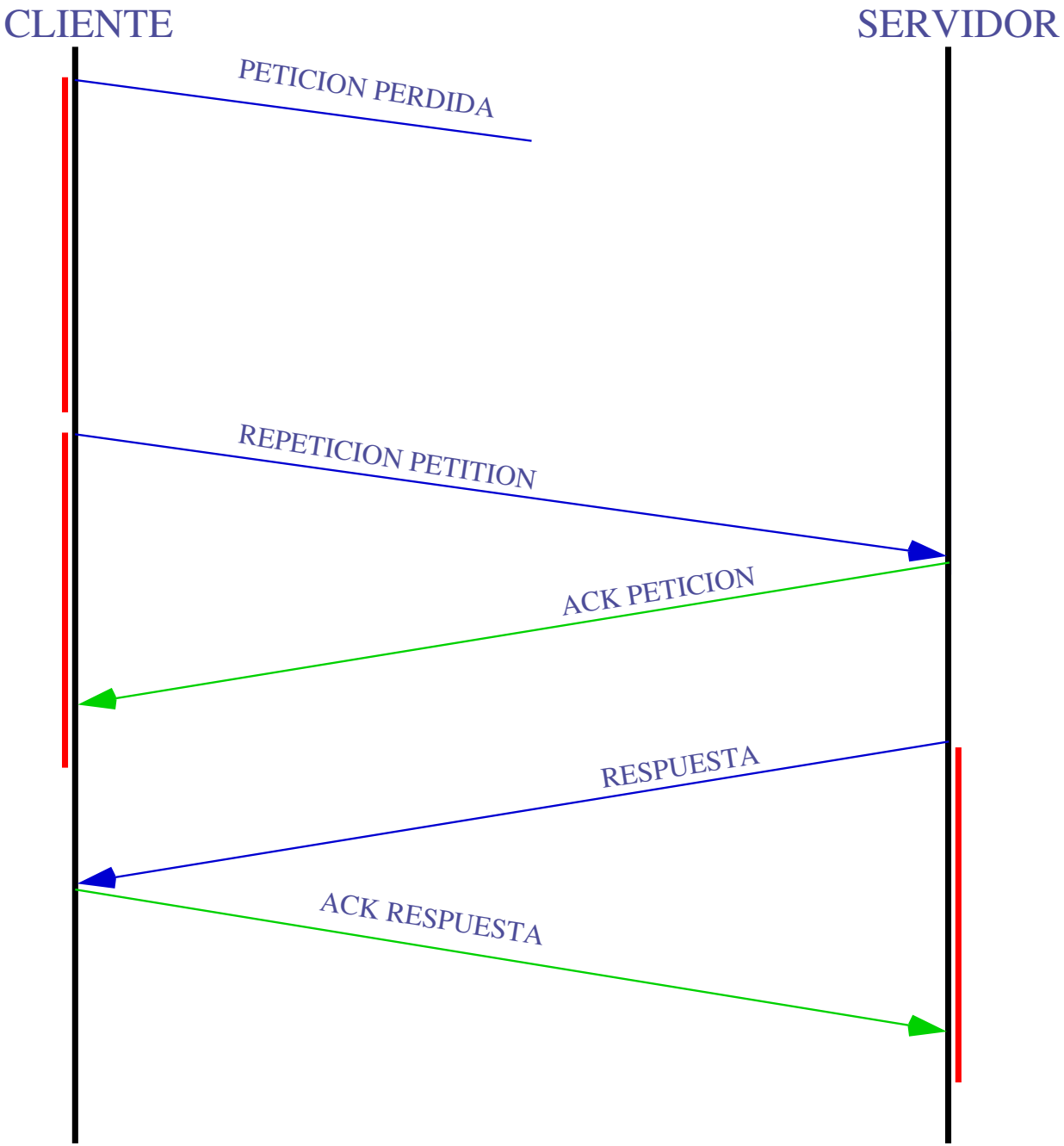
  - Por cada mensaje que se envía se lanza un temporizador

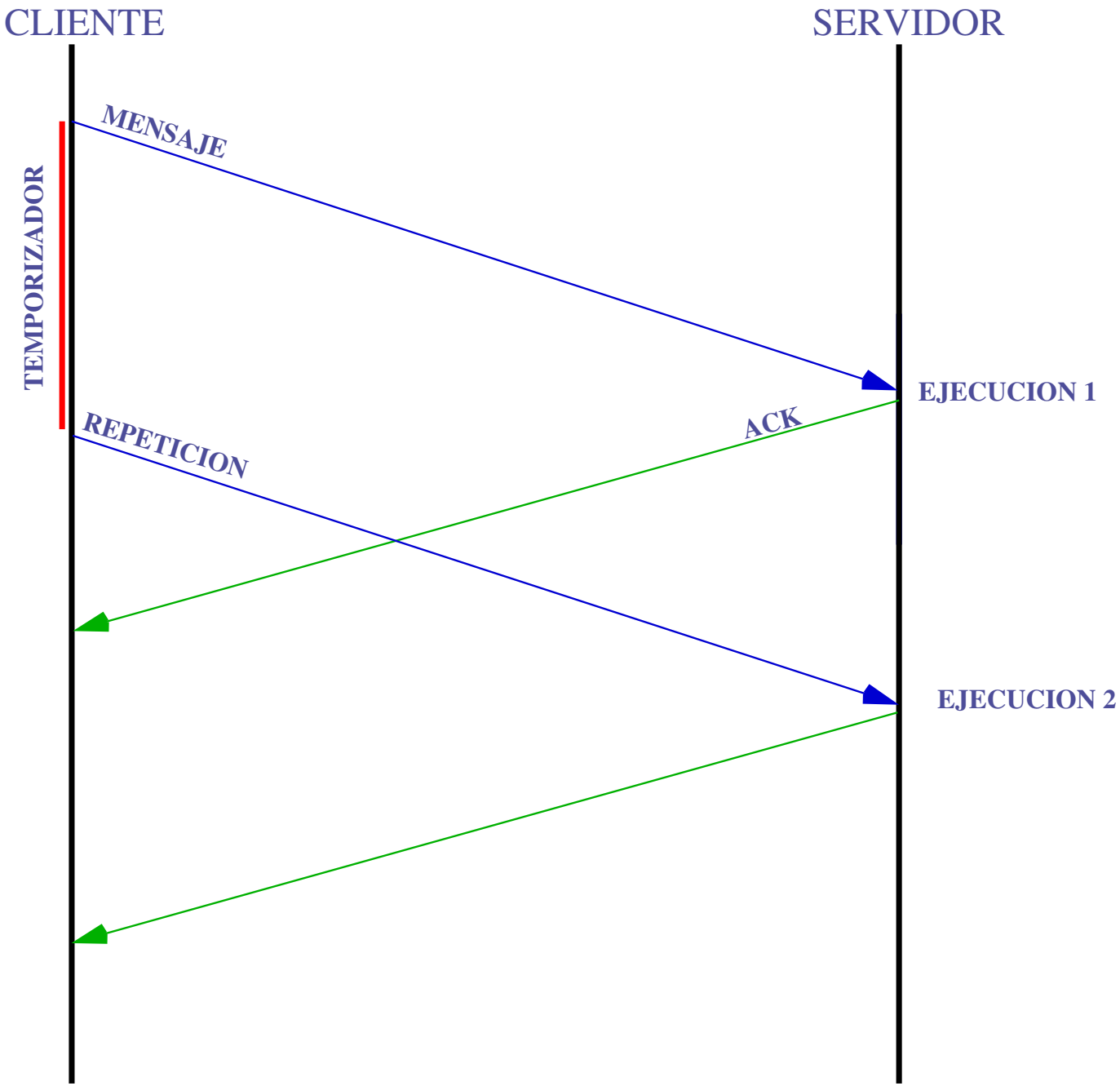
  - Por cada mensaje que se recibe se envía un ACK

  • Si recibimos ACK ⇒ Detenemos temporizador

  • Si temporizador se agota ⇒ Repetimos mensaje

  ► ¿Qué pasa si un mensaje es especialmente lento?
  ⇒ No se garantiza semántica “exactamente una vez”


CLIENTE
SERVIDOR
PETICION PERDIDA
REPETICION PETITION
ACK PETICION
RESPUESTA
ACK RESPUESTA


CLIENTE
SERVIDOR
MENSAJE
TEMPORIZADOR
EJECUCION 1
ACK
REPETICION
EJECUCION 2

Filtrando mensajes duplicados I

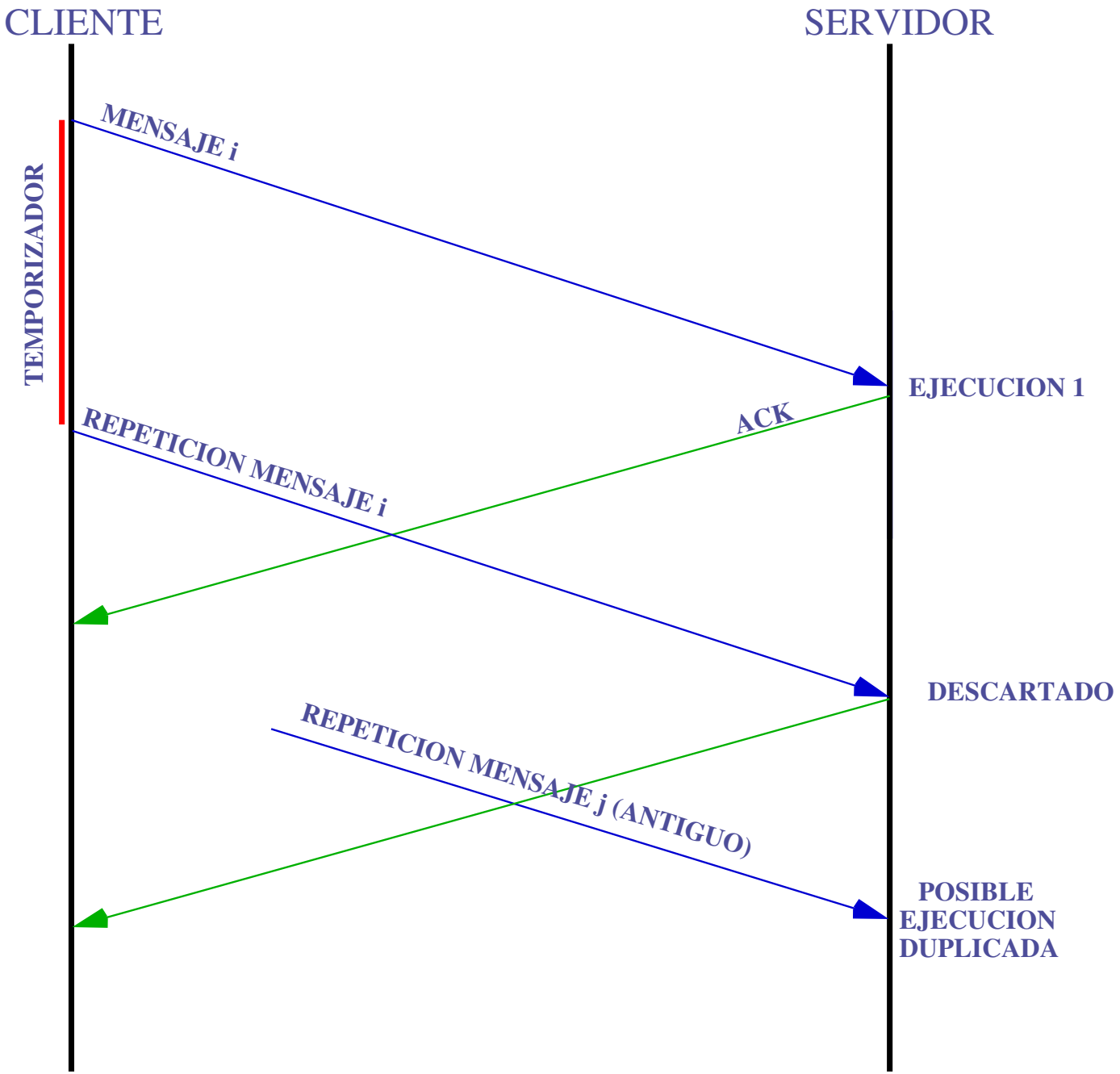
- **Medida 2:**

  - El cliente marca con un identificador único todo mensaje enviado a un proceso
  - El servidor mantiene una lista de identificadores de los mensajes recibidos
  - El servidor mantiene una lista de las respuestas a los mensajes recibidos de las que no se ha recibido ACK
  - Los identificadores y las respuestas muy antiguas se borran del servidor

  • Si el identificador no está en la lista ⇒ Se ejecuta el procedimiento

  • Si el identificador está en la lista ⇒
  Se descarta el mensaje (y se repite la respuesta - ACK)

  ► ¿Qué pasa si recibimos un mensaje repetido muy antiguo?
  ⇒ No se garantiza semántica “exactamente una vez”


CLIENTE
SERVIDOR
TEMPORIZADOR
MENSAJE i
EJECUCION 1
ACK
REPETICION MENSAJE i
DESCARTADO
REPETICION MENSAJE j (ANTIGUO)
POSIBLE
EJECUCION
DUPLICADA

Filtrando mensajes duplicados II

- **Medida 3:**

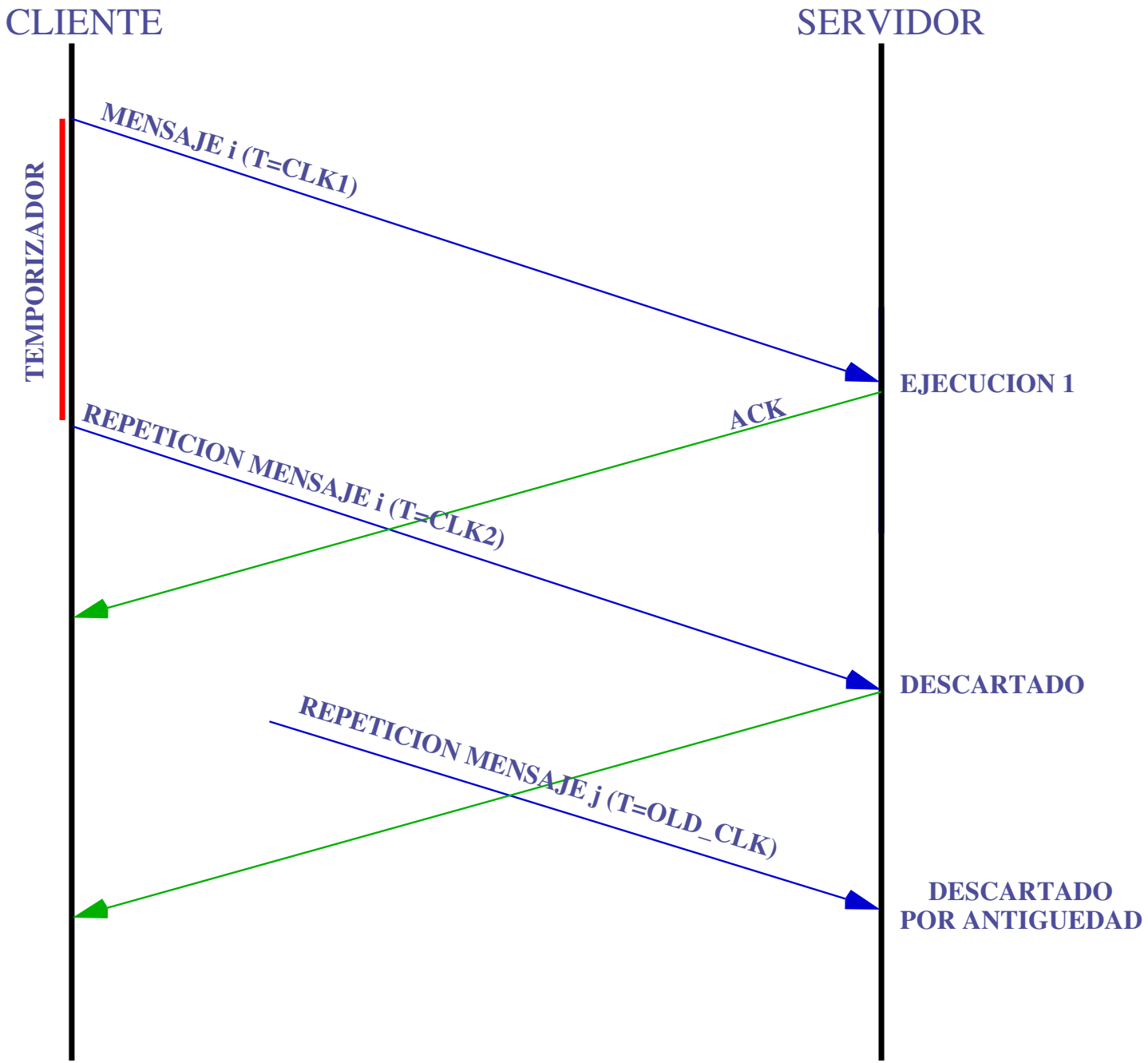
  - El cliente marca con un sello de tiempo todo mensaje enviado a un proceso

  - Los relojes del cliente y del servidor están (más o menos) sincronizados

  • Si el sello de tiempos no es muy antiguo ⇒ Se ejecuta la Medida 2

  • Si el sello de tiempos es muy antiguo ⇒ Se descarta el mensaje

  ► Con el modelo de fallos supuesto (sólo omisión) y en un sistema síncrono o parcialmente síncrono
  ⇒ Se garantiza la semántica "exactamente una vez" en tiempo infinito.


CLIENTE
SERVIDOR
TEMPORIZADOR
MENSAJE i (T=CLK1)
EJECUCION 1
ACK
REPETICION MENSAJE i (T=CLK2)
DESCARTADO
REPETICION MENSAJE j (T=OLD_CLK)
DESCARTADO
POR ANTIGUEDAD

RPC con otros modelos de fallo

Asumamos un modelo con fallos de comunicación y fallos de *crash*

- Imposible garantizar semántica "exactamente una vez" (el servidor puede fallar y no responder jamás).
- Cuando se dispara un temporizador, imposible distinguir entre:
  - Fallo de red (pérdida de paquete, partición, etc.)
  - Fallo de *crash*
  - Ambos fallos combinados
- Cuando se dispara un temporizador puede pasar:
  - La petición nunca fue recibida
  - La petición fue recibida, y ejecutada pero la respuesta se perdió
  - etc.

Garantías de la semántica RPC

- Todo depende del modelo de fallo que se utilice
- Si el fallo puede ser arbitrario (fallos bizantinos)
  $\Rightarrow$ No hay garantías posibles
- Para modelos de fallo más optimistas (omisión / crash)
  $\Rightarrow$ Puede haber ciertas garantías
- Definimos tres tipos de semántica para los protocolos RPC

  Pudiera ser: El método se puede ejecutar una vez o ninguna.
  No se aplican mecanismos de tolerancia a fallos.
  Al menos una vez: El método se ejecuta una o más veces.
  Se aplican mecanismos de tolerancia a fallos que evitan fallos de omisión.
  Como máximo una vez: El método se ejecuta una vez o ninguna. Se aplican mecanismos de tolerancia a fallos que evitan fallos de omisión y evitan la repetición de ejecuciones.

Garantías de la semántica RPC

Retransmisión Petición	Filtrado duplicados	Acción con duplicados	Modelo fallo	Semántica Garantizada
No	No procede	No procede	Cualquiera	Pudiera ser
Sí	No	Reejecutar	Omisión	Al menos una vez
Sí	Sí	Retransmitir respuesta	Omisión + Crash	Como máximo una vez

Garantías de la semántica RPC: Pudiera ser

- **Pudiera ser**: Se utiliza cuando pueden perderse invocaciones o cuando una invocación retrasada (repetida) es inútil.
  - En algunas aplicaciones con restricciones tiempo real en redes de elevada latencia, un mensaje repetido es inútil (p.e. Voz sobre IP).
  - Las semánticas *pudiera ser* se implementan sin ACKs.
  - CORBA permite este tipo de semántica.

Garantías de la semántica RPC: Al menos una vez

- **Al menos una vez**: Se utiliza cuando una invocación puede repetirse varias veces sin afectar el funcionamiento de la aplicación.
  - Decimos que una operación es **idempotente** cuando puede realizarse repetidas veces con el mismo efecto que si hubiera sido realizada una sola vez.
  - Cuando una aplicación distribuida puede realizarse exclusivamente a través de invocaciones idempotentes, se puede utilizar esta semántica para su implementación.
  - Hay operaciones que pueden realizarse de forma idempotente o no idempotente dependiendo del diseño.
    No idempotente: (RPC) Incrementa en 10 el saldo.
    Idempotente: (RPC) Lee el saldo + (Local) incrementa en 10 el saldo + (RPC) escribe el nuevo saldo.
  - Sun RPC proporciona esta semántica

Garantías de la semántica RPC: Como máximo una vez

- **Como máximo una vez**: El invocante recibe un resultado, en cuyo caso sabe que el método se ejecutó exactamente una vez, o una excepción que le informa de que no se recibió el resultado, en cuyo caso el método pudo ejecutarse o no.
  - La mayoría de los entornos RPC implementan este tipo de semántica.
  - Es la que se aproxima más a la semántica LPC.
  - No requiere operaciones idempotentes.
  - Java RMI, CORBA y el DSA de Ada implementan esta semántica.

Optimizando el protocolo RPC

- Hemos descrito el mecanismo básico del protocolo. Existen situaciones en las que puede optimizarse.
  - Los ACKs son costosos, podemos retrasarlos y enviarlos con la respuesta
  - La retransmisión es costosa. Hay que ajustar los temporizadores al retardo de la comunicación y a su varianza. Problema complejo.
  - Para mensajes grandes, enviar paquetes en ráfagas y asentir con un solo ACK. ACKs negativos, etc.

Computación RPC

Computación RPC: Introducción

- ¿Qué necesitamos para poder disponer de un sistema de Computación RPC?
- Objetivo: Transparencia (Invocaciones RPC = Invocaciones LPC).
  - Sistema de comunicación de mensajes (protocolo RPC).
  - Mecanismos de representación de interfaces.
  - Mecanismos de representación de la información.
  - Servicios de localización (nombrado) de recursos.
- ¿Es realmente posible la transparencia RPC?

Computación RCP: codificación y compilación

Llamadas a procedimientos locales	Llamadas a procedimientos remotos
Codificación: Declaración "estándar" *package Cuenta_Corriente is* *procedure Suma(Amount: Integer);*	**Codificación:** Declaración "modificada" *package Cuenta_Corriente is* *pragma Remote_Call_Interface;* *procedure Suma(Amount: Integer);* ...
Compilación: Creación del objeto	**Compilación:** ¿Sólo creación del objeto?
Enlazado: En tiempo de compilación.	**Enlazado:** ¿En tiempo de compilación?

Computación RPC: compilación

- Notación: servidor = llamado, cliente = llamante.

- El "servidor" define y exporta un fichero en el que se definen las interfaces que soporta y los argumentos que se esperan
  A veces se utiliza un lenguaje especial para definir estas interfaces: IDL (*Interface Definition Language*)
  Gracias al IDL el "cliente" y el "servidor" pueden estar codificados en lenguajes diferentes.

- El cliente debe conocer la información sobre las interfaces que utiliza.

- Al compilar se generan dos elementos
  - El *stub* (suplente, delegado) en el lado del cliente.
  - Los *skeleton* (esqueleto, esquema) en el lado del servidor.

Stubs

- Notación: En algunos libros, en vez de stub utilizan el término *proxy*.
- Su papel es el de hacer que la invocación RPC sea transparente para el cliente.
- El stub y el cliente se enlazan en tiempo de compilación.
- Se comporta como un "objeto local" llamado por el cliente, pero en lugar de ejecutar la invocación, la dirige al objeto remoto.
- Oculta los detalles de referencia al objeto remoto.
- Se ocupa de la representación de los datos en los mensajes (aplanamiento, desaplanamiento).
- Hay un stub por cada procedimiento remoto que el cliente pueda invocar.
- El stub comunica con el skeleton.

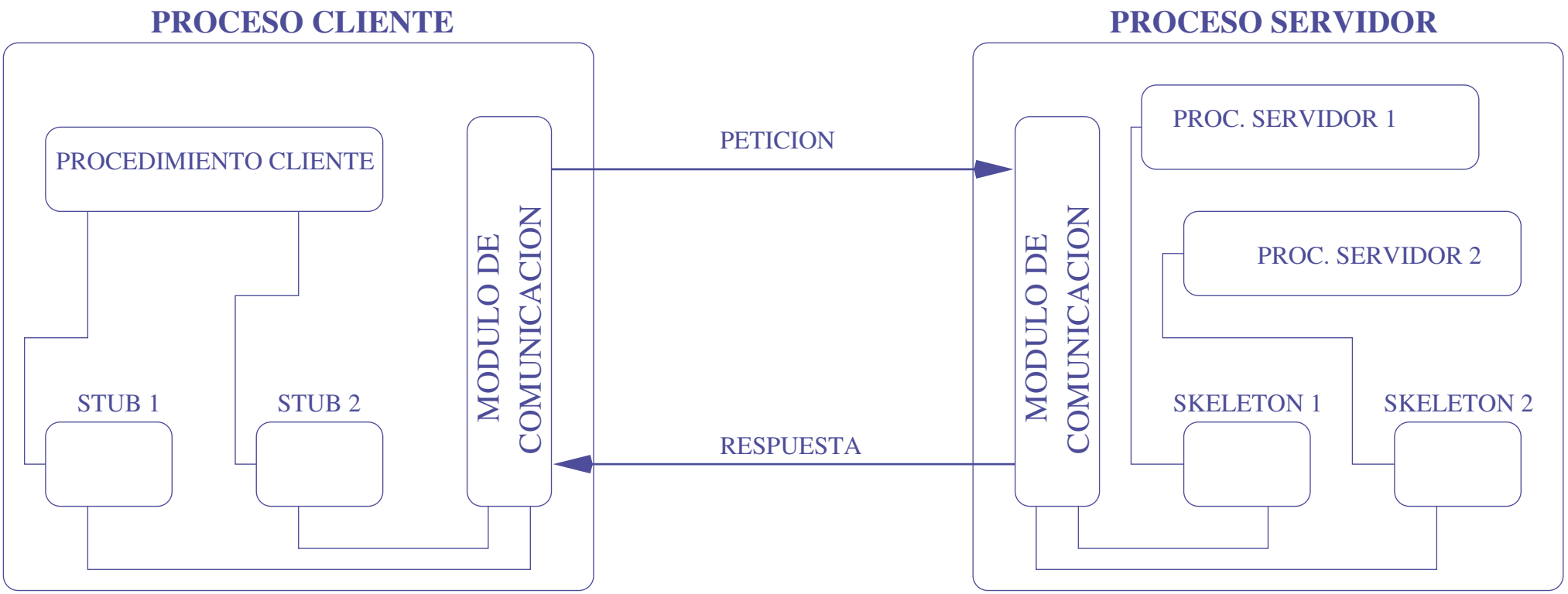
Skeletons

- Su papel es el de hacer que la invocación RPC sea transparente para el servidor.
- El skeleton y el servidor se enlazan en tiempo de compilación.
- Se comporta como un “objeto local” que llama al servidor.
- Se ocupa de la representación de los datos en los mensajes (aplanamiento, desaplanamiento)
- Hay un skeleton por cada procedimiento remoto que el servidor exporta.

Los datos en los mensajes

- Notación: aplanamiento = *marshalling*, desaplanamiento = *demarshalling*.
- Los datos se aplanan al "meterlos" en un mensaje y se desaplanan al "sacarlos" del mismo.
- El aplanamiento/desaplanamiento debe tratar con cuestiones de representación de la información:
  Orden de bits, representación de tipos (string, int, array, etc), alineación, etc.
- Debe contemplarse la posibilidad de que el cliente y el servidor ejecuten en máquinas diferentes (diferente hardware, diferente sistema operativo, etc.)
- Deben existir mecanismos de representación de datos complejos (estructuras, punteros, etc.)
- El aplanamiento/desaplanamiento es responsabilidad de los stubs y de los skeletons.

Esquema de funcionamiento



Servicio de localización

- En un sistema distribuido con invocaciones RPC, el “enlazado” entre el cliente y el servidor se realiza en tiempo de ejecución.
- El servidor y el cliente pueden estar en diferentes nodos de una red (que pueden cambiar)
- Es necesario disponer de un servicio que permita “localizar” al servidor.
- Procedimiento:
  - El servidor se registra en un servicio de directorio al arrancar.
  - El cliente busca en el directorio para localizar al servidor apropiado
  - La asociación cliente/servidor puede realizarse de varios modos
    - Cableada (no necesario directorio)
    - Proximidad
    - Balanceo
    - etc.

Transparencia en las invocaciones RPC: fallos

- Los creadores de RPC intentaban que las llamadas a procedimientos remotos fueran lo más parecido posible a las llamadas locales (transparencia)
- En la actualidad se sabe que el RPC no puede ser totalmente transparente.
- Las invocaciones remotas son más vulnerables a fallos que las locales.
- Cualquiera que sea la semántica de invocación empleada siempre es posible que no reciba resultado alguno.
- En ese caso, es imposible distinguir entre un fallo de la red o un fallo del proceso remoto.
- Cuanto más medidas de guarda tome el protocolo RPC, mayor será su complejidad.
- En la práctica, la mayor parte de los entornos RPC garantizan alguna forma de semántica *como mucho una vez*.

Transparencia de las invocaciones RPC: argumentos

- En LPC podemos pasar cualquier tipo de argumento a los procedimientos
- En RPC puede haber problemas con los argumentos
  - Limitación en el tamaño
    La mayoría de los entornos RPC limitan el tamaño de sus argumentos
  - Argumentos complejos: ¿qué pasa con los punteros?
    Dirección local no tiene sentido en proceso remoto
    Derreferenciación $\Rightarrow$ elevado coste
    La mayoría de los entornos RPC limitan el tipo de sus argumentos

Modelos orientados a objetos

- Recientemente, el modelo de programación orientado a objetos ha dido extendido permitiendo la comunicación entre objetos de procesos diferentes

- Principios de funcionamiento análogos al RPC, pero basados en un paradigma orientado a objetos
  - RPC (Remote Procedure Call): permite a programas cliente llamar a procedimientos en programas servidor.
  - RMI (Remote Method Invocation): Permite que un objeto que vive en un proceso invoque los métodos de un objeto que vive en otro proceso.

- Simplifica el paso de parámetros y produce un modelo más comprensible para el programador. Solo existen dos tipos de objetos
  - Objetos locales: Se pasan por valor
  - Objetos remotos: Se pasan por referencia

- Existen métodos *factoría* que se ocupan de crear un objeto cuando este no existe pero aparece en una invocación.

Modelos basados en eventos

- Se utilizan normalmente en los modelos RMI basados en objetos
- Permiten a los objetos recibir notificaciones de los eventos que ocurren en otros objetos en los que se ha manifestado interés.
- El modelo ha sido extendido para permitir escribir programas distribuidos basados en eventos.
- Este modelo permite dos modos de programación
  - Modo síncrono: El que invoca espera por la respuesta
  - Modo asíncrono: El que invoca no espera, pero será notificado sobre cualquier evento que suceda en el objeto invocado.
- Los entornos RMI más populares (CORBA, Java RMI, etc.) utilizan modelos basados en eventos.

Entornos RPC

1. Introducción y Conceptos Básicos

2. Protocolos RPC

3. Computación RPC

4. Entornos RPC

Los entornos RPC como Middleware

- Middleware: Software “in the middle”

- Proporciona transparencia de ubicación e independencia de los detalles relativos a protocolos, sistemas operativos, hardware, etc.

Aplicaciones
RPC, RMI y eventos
Representación de datos
Protocolo de petición-respuesta
Sistema operativo

Componentes de un entorno RPC

- Estándares para la representación de datos
- Compiladores de interfaces
  - Generan los *stubs* y los *skeletons*
  - La descripción del interfaz puede realizarse:
    - De forma implícita con el lenguaje de programación (el entorno solo soporta ese lenguaje).
    - De forma explícita a través de un IDL (el entorno puede soportar varios lenguajes de programación).
- Servicios para gestionar
  - Servicio de directorio
  - La sincronización de relojes
  - Los eventos
- Herramientas para visualizar el estado del sistema y gestionar servidores de aplicaciones.

Ejemplos de entornos RPC

- **DCE:** Distributed Computing Environment. Desarrollado entre 1987 y 1989. Fue el primer entorno de calidad comercial en implementar las ideas relativas al RPC.
- **Sun RPC (ONC):** Propuesto por SUN Microsystems, usado en la arquitectura de NFS y en muchos servicios UNIX
- **CORBA:** Nueva generación de entornos orientados a objetos multiplataforma y multilenguaje.
- **Java RMI:** Entorno orientado a objetos del lenguaje de programación Java.
- **DSA:** Distributed System Annex. El entorno RPC para el lenguaje de programación Ada

Multithreading: Introducción

- Para aumentar el rendimiento es habitual utilizar multithreading.
- La idea es aprovechar el tiempo ocioso del servidor mientras espera por una E/S al atender a un cliente, lanzando varios threads concurrentes.
- Existen tres opciones:
  - Servidor mono-thread: Sólo hace una cosa a la vez, usa llamadas al sistema *send/recv* y se bloquea esperando.
  - Servidor multi-thread: Con concurrencia interna. Cada petición hace que se cree un nuevo thread para atenderla.
  - Upcalls: El bucle despachador de eventos hace una llamada a procedimiento para cada evento que se produce, como X11 o windows

Mono-thread: inconvenientes

- Las aplicaciones pueden llegar al interbloqueo si las peticiones forman ciclos.
- Mucho tiempo ocioso esperando respuestas a peticiones pendientes.
  - Tiempo desperdiciado en el cliente
  - Tiempo desperdiciado en el servidor
- Es difícil implementar el propio protocolo RPC con este modelo
  - ¿Cómo se gestionan los timers?
  - ¿Cómo se gestionan las repeticiones?
  - Diseño complejo de servidores que atiendan peticiones simultáneas
  - etc.

Multithreading

- La idea es soportar concurrencia interna, como si cada proceso fuera realmente varios procesos compartiendo un espacio de direcciones.
- El planificador de threads usa timers para emular este comportamiento
- Cada petición se atiende en un nuevo thread.
- Cuando un thread se bloquea en una operación (p.e. E/S) el resto de los threads aprovechan el tiempo de espera
- El diseñador debe implementar de manera adecuada los mecanismos necesarios para el control de concurrencia (semáforos, objetos protegidos, etc.)

Aspectos negativos del multithreading

- Los usuarios pueden tener poca experiencia con la concurrencia por lo que es frecuente que aparezcan errores de programación.
- Estos errores son difíciles de encontrar y depurar. En ocasiones son difíciles de reproducir.
- Cada thread necesita su propia pila, por lo que el consumo de memoria puede dispararse si hay demasiadas ejecuciones concurrentes.
- Si se genera un número excesivo de threads, el rendimiento puede caer abruptamente y los clientes “pensar” que el servidor ha fallado.

Modelo de Upcalls

- Es habitual en sistemas de ventanas.
- Los usuarios registran los procedimientos de atención a eventos.
- El bucle despachador de eventos llama a un procedimiento cuando se recibe un evento. Espera que termine la llamada, y despacha un nuevo evento.
- Se suele utilizar combinado con multithreading
  - Es, quizás, el mejor modelo de programación de RPC.
  - Cada manejador puede ser etiquetado indicando si se ejecutará con su propio thread o no.
  - Los desarrolladores deben ser cuidadosos sobre cómo y cuando utilizar threads.

Soporte del sistema operativo para RPC

- LRPC: *Lightweight* RPC. Para cuando el emisor y el receptor están en la misma máquina. Una sola llamada al sistema send_rcv/rcv_send.

- Fbufs: Herramienta para acelerar los protocolos a base de niveles apilados. Usa gestión de memoria con buffers en caché y compartición con protecciones

- Mensajes Activos: Mejora el rendimiento en máquinas paralelas. Supone que el emisor sabe todo sobre el destino, incluyendo disposición de la memoria, formato de datos, etc.

- U/Net: Comunicaciones de baja latencia y alto rendimiento para ATM en máquinas Unix. La aplicación y el controlador ATM comparten la misma región de memoria. La E/S se realiza añadiendo mensajes a una cola o leyendo de esa cola.

Referencias

[Birm 96] Kenneth P. Birman, Building Secure and Reliable Network Applications, M anning Publications Co., 1996. Disponible en PDF (con el permiso del autor).

[Colo 01] George Coulouris, Jean Dollimore, Tim Kindberg, Distributed Systems (3rd. edition), Addison Wesley., 2001.