Comunicación entre procesos

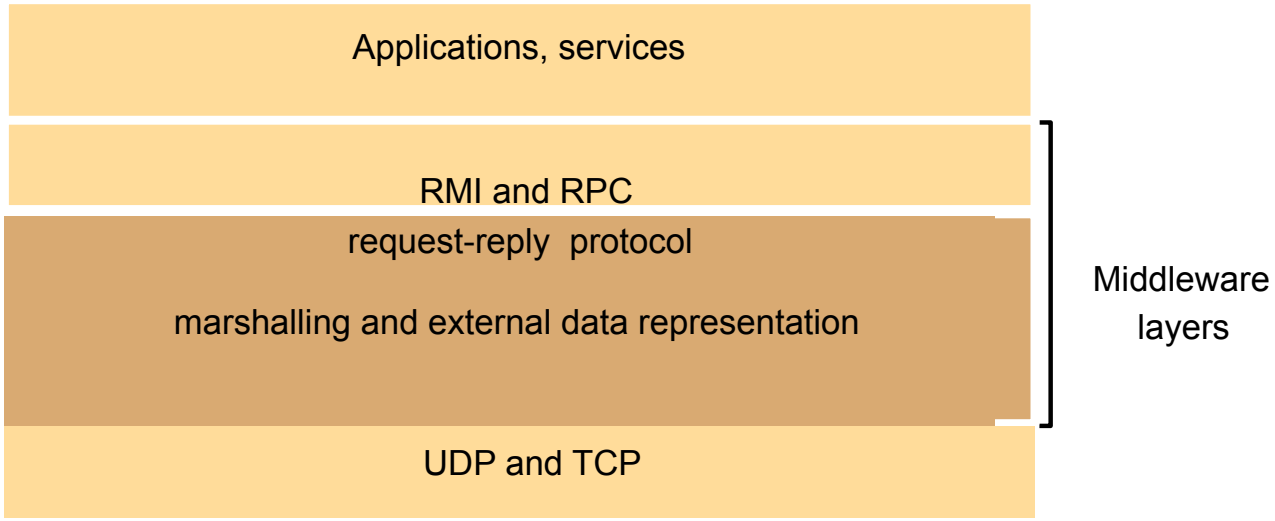
UNIVERSIDAD NEBRIJA

Índice

- Comunicación entre procesos
- Sockets
- Representación de datos
- Comunicación entre procesos: cliente-servidor (síncrona)
- Comunicación entre procesos: cliente-servidor (isócrona)
- Comunicación entre procesos: migración de código

Comunicación entre procesos

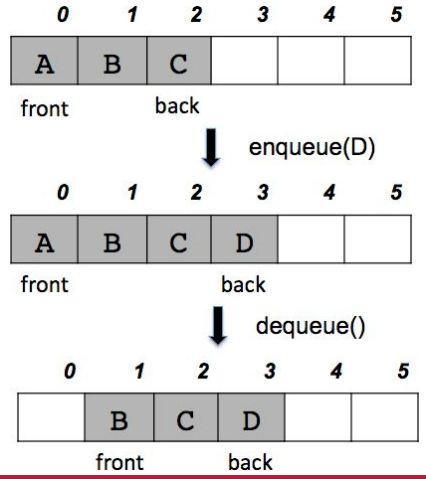
- Pila para la comunicación entre procesos:



(*) RMI: Remote Method Invocation
(*) RPC: Remote Procedure Call

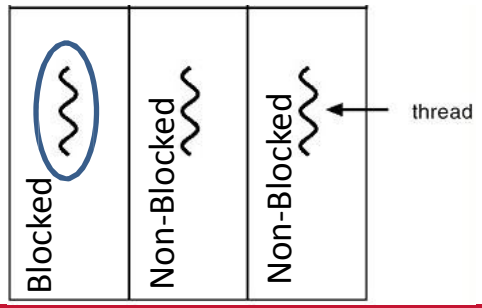
Comunicación entre procesos

- Características de la comunicación entre procesos:
  - Comunicación síncrona y asíncrona:
    - Emisores generan mensajes que se almacenan en colas remotas
    - Receptores desencolan mensajes



Comunicación entre procesos

- Características de la comunicación entre procesos:
  - Comunicación síncrona y asíncrona:
    - Síncrona: procesos de envío y recepción bloqueantes para sincronizar
    - Asíncrona: proceso de envío no bloqueante, proceso de recepción puede ser o no bloqueante. El no bloqueante gestiona el búfer en segundo plano
  - Los procesos bloqueantes son los más comunes en los sistemas actuales, ya que realmente con múltiples hilos no paraliza el procesado y reduce la necesidad de control de flujo



Comunicación entre procesos

- Características de la comunicación entre procesos:
  - Comunicación síncrona y asíncrona:
    - Envío síncrono y recepción síncrona: La recepción causa la suspensión del proceso que recibe los datos. La emisión también suspende el proceso hasta que el proceso receptor envía una confirmación de recepción
      - **Se utiliza cuando ambos procesos necesitan los datos antes de continuar con la ejecución**
    - Envío asíncrono y recepción síncrona: La recepción causa la suspensión del proceso hasta que lleguen los datos. La emisión no se suspende (no bloqueante)
      - **Se utiliza cuando el proceso emisor puede seguir funcionando de manera autónoma sin necesidad de confirmar que el receptor posea los datos**

Comunicación entre procesos

- Características de la comunicación entre procesos:
  - Comunicación síncrona y asíncrona:
    - Envío síncrono y recepción asíncrona: Existen dos opciones:
      - Los datos llegan en el momento en el que se solicita la recepción, se envía confirmación y se desbloquea el proceso de envío
      - Los datos no llegan en el momento en el que se solicita la recepción, el proceso de envío se bloquea hasta que se realice una nueva recepción asíncrona. Se puede utilizar un *polling* en el receptor para invocar de manera repetitiva el proceso de recepción
    - Envío asíncrono y recepción asíncrona: Ambos procesos son no bloqueantes. Admite un *polling* en el receptor

Comunicación entre procesos

- Características de la comunicación entre procesos:
  - Comunicación síncrona y asíncrona:
    - Aunque el bloque puede servir como opción de sincronismo, en la mayoría de las aplicaciones es INACEPTABLE que un proceso se quede suspendido de manera indefinida
    - Buenas prácticas:
      - Utilizar temporizadores
      - Recurrir a procesos hijos o hilos donde ejecutar las operaciones bloqueantes

Comunicación entre procesos

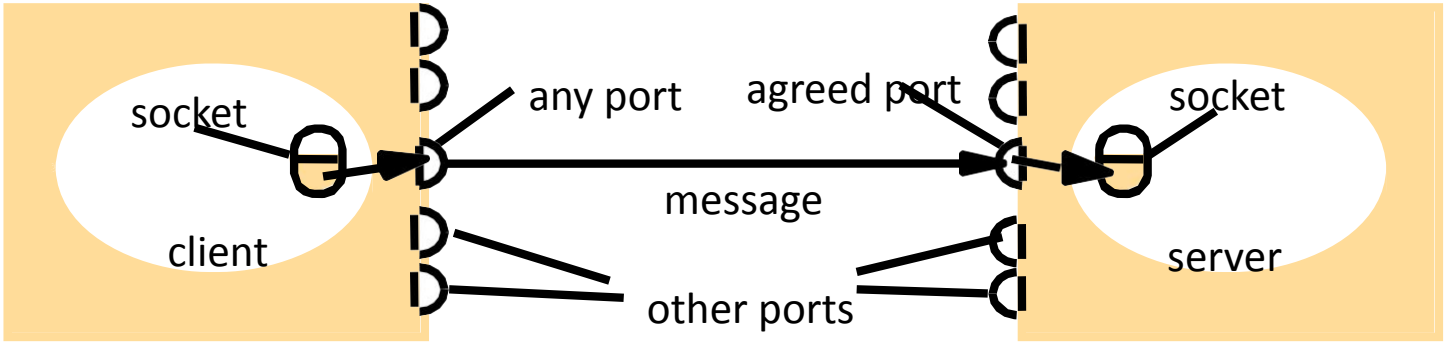
- Destino de los mensajes: Se recurre al uso de múltiples puertos para recibir
- mensajes. Un puerto tiene un receptor pero múltiples emisores.
  - Conociendo el número de puerto de puerto se puede enviar el mensaje
  - Las ventajas de utilizar puertos y no procesos es que proporcionan varios puntos de entrada al proceso receptor

  - Se suele utilizar un enlazador entre los servicios, los procesos y sus direcciones y puertos
  - Los sistemas operativos suelen proporcionar identificadores para procesos destino independientemente de su localización: FACILITAN LA MIGRACIÓN

- Fiabilidad: Se garantiza la recepción de un número razonable de mensajes. Los mensajes deben recibirse sin corromperse ni duplicarse

Sockets

- Los sockets son conectores que permiten la transmisión de un mensaje entre dos procesos
- El puerto en el proceso del recepción debe asociarse al propio proceso explícitamente
- Un proceso NO PUEDE COMPARTIR PUERTOS CON OTROS PROCESOS



Sockets

- Comunicación mediante datagramas (UDP): sin acuse de recibo ni reintentos
  - Envío no bloqueante. Recepción bloqueante (con posibilidad de *timeout*)
  - Modelo de fallo, aceptable para algunas aplicaciones:
    - Fallo por omisión, desecha mensajes de manera ocasional
    - Fallo de ordenación, no coincide el orden de emisión y de recepción
  - No genera sobrecargas al no garantizar la entrega de los mensajes, ya que:
    - No almacena información de estado en origen y en el destino
    - No transmite mensajes extra

Sockets

- Comunicación mediante streams TCP:
  - Los streams poseen un nivel mayor de abstracción:
    - No necesita especificar el tamaño de los datos. Los datos se proporcionan según se van solicitando
    - Utiliza acuse de recibo. Si el emisor no recibe el acuse de recibo dentro del *timeout* retransmite el mensaje automáticamente
    - Utiliza control de flujo: ajusta las velocidades de transmisor y receptor
    - Utiliza un identificador de paquetes que posibilita su ordenación y la detección de duplicidades
  - Establece la conexión antes de empezar a transmitir mediante *streams*.
  - Una vez establecida la conexión, la lectura y escritura no requiere direcciones ni puertos
  - La abstracción crea SOBRECARGAS EN MENSAJES CORTOS

Sockets

- Comunicación mediante streams TCP:
  - CONCORDANCIA entre el formato de los datos entre transmisor y receptor
  - La lectura BLOQUEA hasta que se disponga de datos en el *buffer* de entrada
  - Cada cliente debe atenderse en un hilo diferente para no bloquear al servidor
  - Modelo de fallo:
    - Número de identificación de paquetes (ordenación y no duplicación)
    - CRC y rechazo de paquetes corruptos
    - *Timeout* para declarar fallo (no se sabe si de red o de proceso)

Representación de datos

- ¿Es necesaria la conversión a un formato COMÚN entre transmisor y receptor? (acordado previamente)

- Alternativa: transmitir los datos junto con el formato utilizado (*marshalling*), para que la conversión se realice en el receptor (*unmarshalling*)
  - *Marshalling*: Aplanado de los campos de la estructura de datos y conversión de los datos a la representación de la red
  - *Unmarshalling*: Conversión de los datos a la representación interna y reconstrucción de la estructura de datos

- Serialización de objetos: codificación de objetos en un flujo de bytes y soporte para su reconstrucción desde el flujo de bytes

Representación de datos

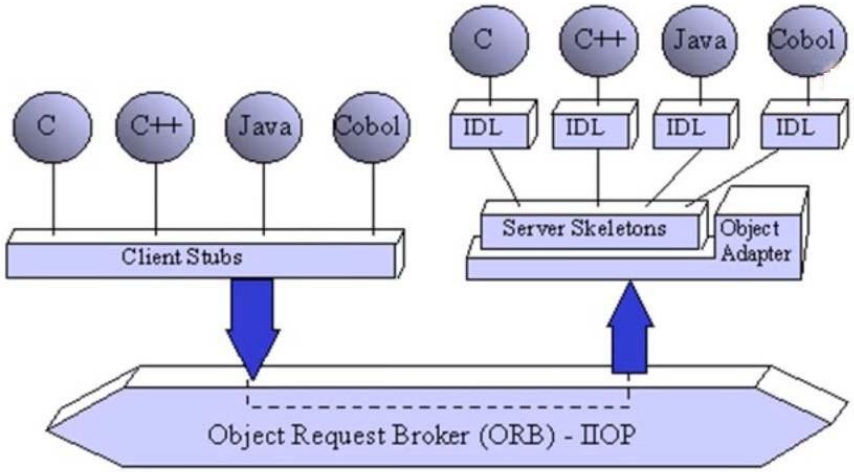
- CORBA: empaqueta objetos. Útil para diversos lenguajes de programación
  - Los datos en CORBA se transforman a representaciones binarias
  - Requiere de un fichero IDL para especificar los tipos de datos empaquetados

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h "	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on "	
24–27	1934	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

Representación de datos

- CORBA: empaqueta objetos. Útil para diversos lenguajes de programación
  - Los datos en CORBA se transforman a representaciones binarias
  - Requiere de un fichero IDL para especificar los tipos de datos empaquetados

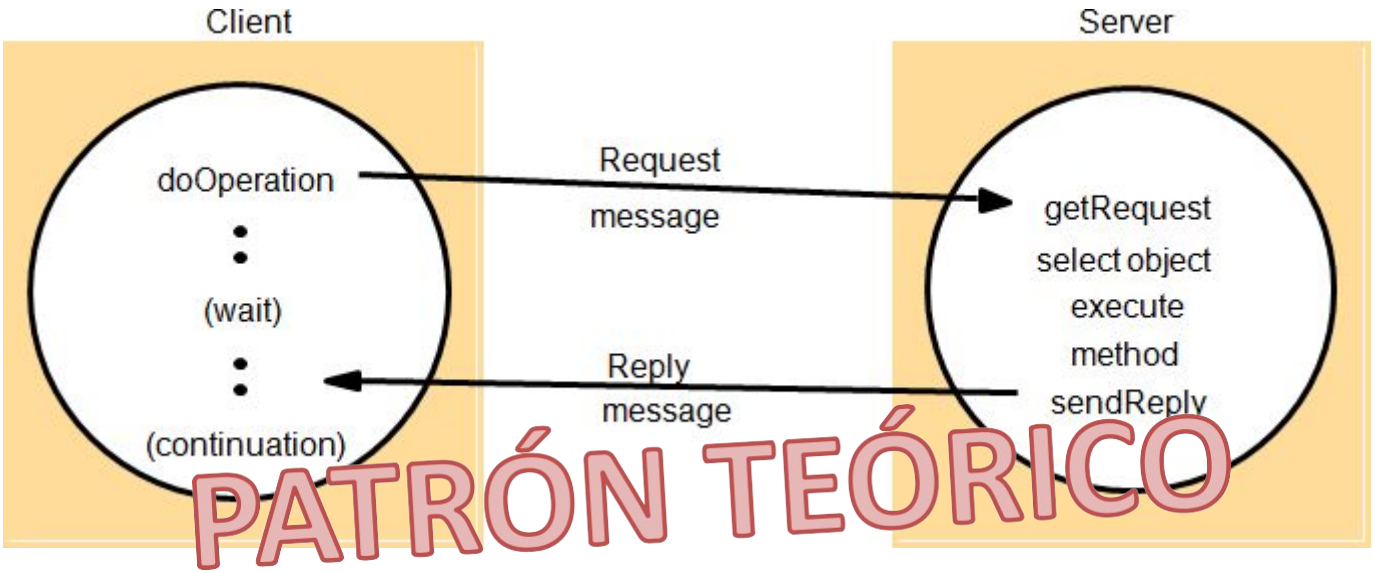


Representación de datos

- Serialización Java: Utiliza la propiedad de reflexión de Java (habilidad de preguntar a una clase sus propiedades, nombres, métodos, campos, etc. EN TIEMPO DE EJECUCIÓN)
  - Implementar una clase con la interfaz Serializable (java.io) permite que sus instancias sean serializables
  - Los objetos serializables incluyen nombre y número de versión de la clase
  - Al serializar un objeto, TODOS LOS OBJETOS A LOS QUE REFERENCIA SE SERIALIZAN CON ÉL, para asegurar que no se pierde nada en el destino
  - Se recurre al uso de apuntadores (handlers) para las referencias
  - Una clase NO se serializa más de una vez. Si existen dos objetos de la misma clase se recurre al apuntador
  - Se utilizan funciones de writeObject y readObject como si de un fichero se tratara
- IMPORTANTE: Borrar la referencia para evitar que el objeto pueda ser reutilizado

Comunicación entre procesos: cliente-servidor (síncrona)

- Comunicación síncrona: el cliente está bloqueado hasta que responde el servidor
  - Fiabilidad: la respuesta/confirmación del servidor



Comunicación entre procesos: cliente-servidor (síncrona)

- Diseño de servidores
  - Servidor iterativo: el propio servidor manipula la petición y devuelve una respuesta a la petición del cliente
  - Servidor concurrente: no manipula por sí mismo la petición, la pasa a un hilo separado o a otro proceso, después de lo cual de inmediato queda en espera de la siguiente petición entrante (TEMA 7)
  - Servidor sin estado: no mantiene información con respecto al estado de sus clientes, y puede modificar su propio estado sin necesidad de informar a ningún cliente
  - Servidor con estados: por lo general mantiene información persistente acerca de sus clientes
    - Estado temporal
    - Estado permanente

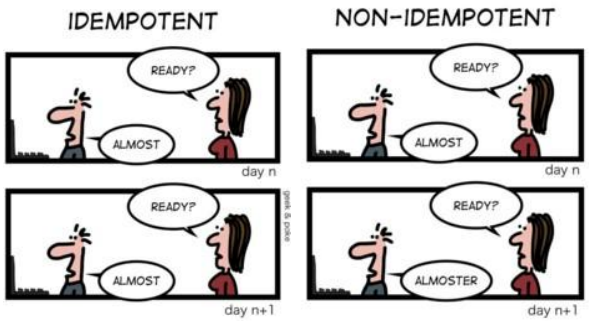
Comunicación entre procesos: cliente-servidor (síncrona)

- Añadir nuevos reconocimientos (ACKs) es redundante ya que la respuesta del servidor actúa como tal
- Si utilizamos establecimiento de conexión (como vimos en TCP) se añaden dos pares de mensajes extra

```
s = new Socket("localhost", serverPort);
DataInputStream in = new DataInputStream( s.getInputStream());
DataOutputStream out =new DataOutputStream( s.getOutputStream());
out.writeUTF(args[0]);
```

- Para invocar operaciones remotas se utiliza el comando doOperation
- El comando doOperation implica *mashalling* y *unmarshalling* de objetos
- El cliente que invoca doOperation se bloquea hasta que el objeto remoto envía el resultado solicitado mediante el comando sendReply
- El comando getRequest se ejecuta en el servidor para atender las peticiones de los clientes

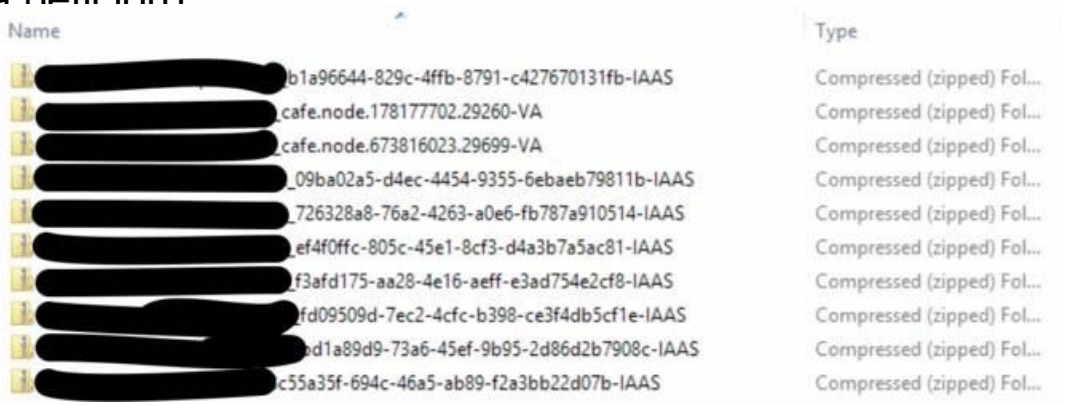
Comunicación entre procesos: cliente-servidor (síncrona)



- Modelo de fallos del protocolo:
  - Timeout límite: el comando doOperation puede desbloquearse indicando que ha fallado, puede solicitar reenvío durante un tiempo o número de reintentos fijo o indicar mediante una excepción que no ha recibido ninguna respuesta
  - Eliminación de mensajes de petición duplicados: es necesario eliminar los duplicados implementando algún tipo de control con el fin de NO atender DOS VECES a la misma petición
  - Pérdida de mensajes respuesta: no es un problema para servidores con operaciones idempotentes. No obstante, los servidores con operaciones no idempotentes deberán implementar medidas especiales para evitar errores indeseados.
- RECUERDA: Operación IDEMPOTENTE es aquella que puede ser llevada a cabo múltiples veces obteniendo siempre el mismo resultado

Comunicación entre procesos: cliente-servidor (síncrona)

- Modelo de fallos del protocolo:
  - Historial: contiene un identificador de petición, un mensaje y el identificador del cliente
    - El servidor podrá retransmitir si los clientes lo solicitan.
    - Consume almacenamiento.
    - Si el volumen de peticiones es un problema: borrado del historial con el tiempo o con la confirmación por parte del cliente (bien sea mediante ACK o con una nueva petición)



Comunicación entre procesos: cliente-servidor (isócrona)

- Sistemas isócronos: la transferencia de datos está sujeta a un retraso máximo y
- mínimo fin a fin, también conocido como inestabilidad limitada (retraso)
  - Desempeña un papel muy importante en la representación de audio y video (streaming / flujo)
    - Flujo simple: una sola secuencia de datos
    - Flujo complejo: consta de varios flujos simples relacionados temporalmente, llamados subflujos
      - Los subflujos están continuamente sincronizados