RPC - gRPC

Jaime A Riascos-Salas

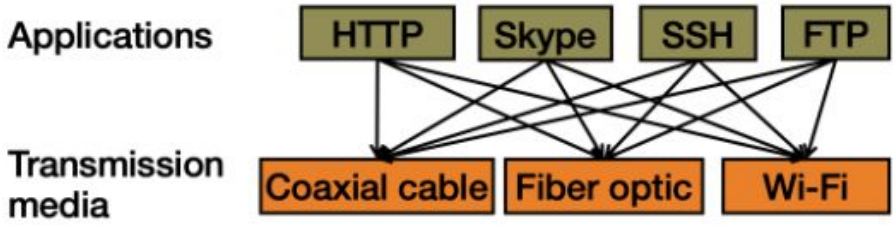
El problema de la comunicación

Para coordinarse, los nodos deben comunicarse.

– Problemas

– ¿Reimplementar cada aplicación para cada nuevo medio de transmisión subyacente?

– ¿Cambiar cada aplicación ante cualquier cambio en un medio de transmisión subyacente?

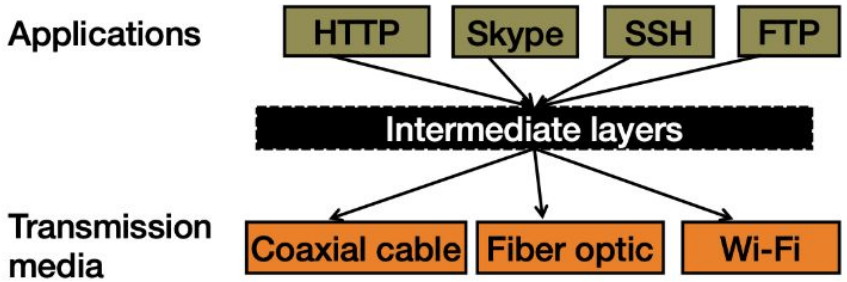


Solucion: layering

El poder de las capas

– Las capas intermedias proporcionan un conjunto de abstracciones para aplicaciones y medios.

– Las nuevas aplicaciones o medios solo necesitan implementarse para la interfaz de la capa intermedia.



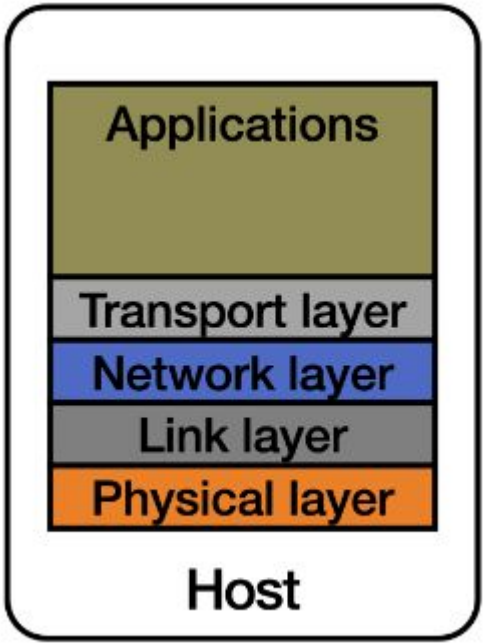
Layering en Internet

Transporte: Proporciona comunicación de extremo a extremo entre procesos en diferentes hosts.

– Red: Entrega paquetes a destinos en otras redes (heterogéneas).

– Enlace: Permite que los hosts finales intercambien mensajes atómicos entre sí.

– Físico: Transfiere bits entre dos hosts conectados por un enlace físico.



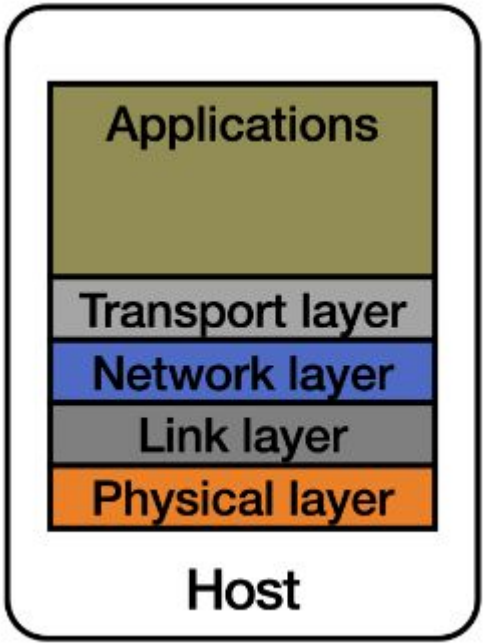
Layering en Internet

El estándar más aceptado para la comunicación en red es TCP/IP (Protocolo de Internet), que proporciona una entrega no fiable de paquetes individuales a hosts distantes de un solo salto.

IP fue diseñado para ocultarse tras otras capas de software:

- TCP (Protocolo de Control de Transporte) implementa un intercambio de mensajes conectado y fiable.
- UDP (Protocolo de Datagramas de Usuario) implementa intercambios de mensajes no fiables basados en datagramas.

TCP/IP y UDP/IP son visibles para las aplicaciones a través de sockets. El propósito de la interfaz de socket era proporcionar una abstracción similar a la de UNIX.



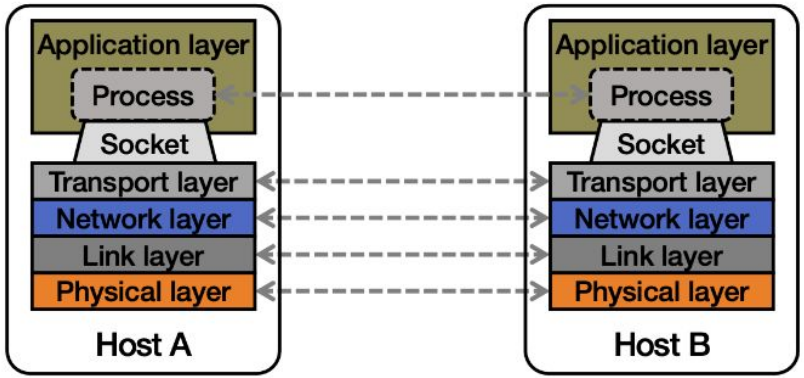
Comunicación basada en Sockets

Socket: La interfaz que el sistema operativo proporciona a la red.

– Facilita el intercambio explícito de mensajes entre procesos.

– Permite construir sistemas distribuidos sobre sockets: send(), recv().

– Por ejemplo: put(key,value) -> message.



Problemas con Sockets

La interfaz de sockets fuerza un mecanismo de lectura/escritura.

La programación suele ser más sencilla con una interfaz funcional.

Para que la computación distribuida se parezca más a la computación centralizada, la E/S (lectura/escritura) no es la solución.

Header embutido en el mensaje.

Lectura/escritura secuencial.

Ejemplo SMTP (Simple Mail Transfer Protocol)

SMTP = Simple Mail Transfer Protocol

```
$ telnet porthos.rutgers.edu 25
Trying 128.6.25.90...
Connected to porthos.rutgers.edu.
Escape character is '^]'.
220 porthos.cs.rutgers.edu ESMTP Postfix
HELO cs.rutgers.edu
250 porthos.cs.rutgers.edu
MAIL FROM: <pxk@cs.rutgers.edu>
250 2.1.0 Ok
RCPT TO: <p@pk.org>
250 2.1.5 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
From: Paul Krzyzanowski <pxk@cs.rutgers.edu>
Subject: test message
Date: Mon, 17 Feb 2020 17:00:16 -0500
To: Whomever <testuser@pk.org>

Hi,
This is a test
.
250 2.0.0 Ok: queued as AC37C3000175
quit
221 2.0.0 Bye
Connection closed by foreign host.
```

This is the message body.
Headers may define the structure of the message but are ignored for delivery.

Ejemplo HTTP = Hypertext Transfer Protocol

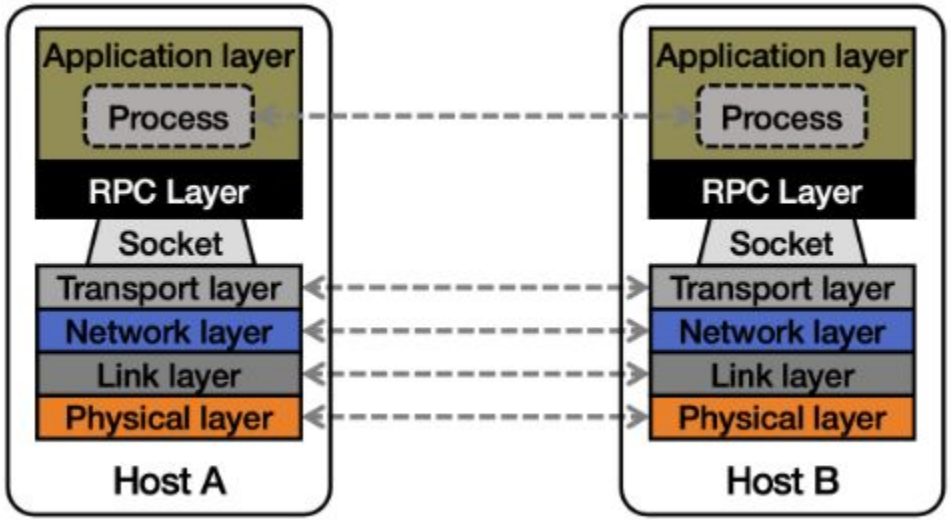
```
$ telnet www.google.com 80
Trying 172.217.12.164...
Connected to www.google.com.
Escape character is '^]'.
GET /index.html HTTP/1.1
HOST: www.google.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
User-Agent: Mozilla/4.0

HTTP/1.1 200 OK
Date: Mon, 03 Feb 2020 18:31:59 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
…
Transfer-Encoding: chunked

73c5
<!doctype html><html itemscope=""
itemtype="http://schema.org/WebPage" lang="en"><head>
        …
        …
```

HTTP = Hypertext Transfer Protocol

First part of the response – HTTP headers

Second part of the response – HTTP content

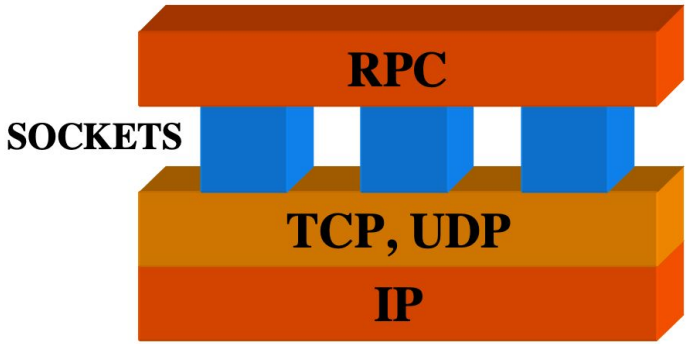
SOLUTION: ANOTHER LAYER!



RPC - Remote Procedure Call

Sin embargo, los sockets son de bajo nivel para muchas aplicaciones, por lo que RPC (Llamada a Procedimiento Remoto) surgió como una forma de ocultar los detalles de la comunicación tras una llamada a procedimiento en entornos de puente heterogéneos.

RPC es el estándar para la informática distribuida (cliente-servidor).



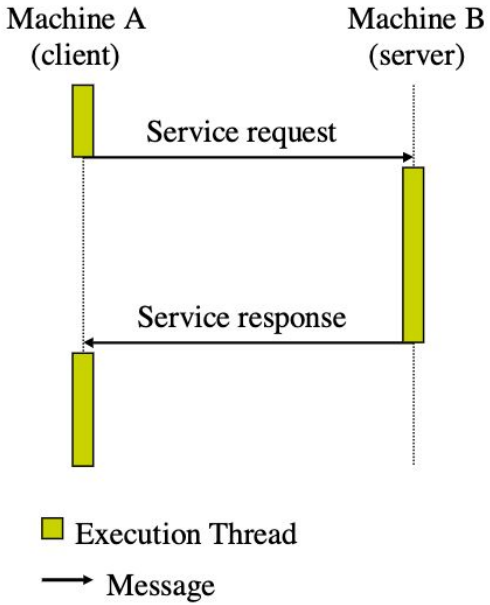
Básica comunicación cliente/servidor

Imaginemos que tenemos un programa (un servidor) que implementa ciertos servicios. Imaginemos que tenemos **otros** programas (clientes) que desean invocar esos servicios.

Para hacer el problema más interesante, supongamos también que:

- el cliente y el servidor pueden residir en diferentes computadoras y ejecutarse en diferentes sistemas operativos.
- la única forma de comunicación es mediante el envío de mensajes (sin memoria compartida, sin discos compartidos, etc.).
- se deben proporcionar algunas garantías mínimas (gestión de fallos, semántica de llamadas, etc.).
- Queremos una solución genérica y no un truco de un solo uso.

Idealmente, queremos que los programas se comporten así (¿suena simple? Bueno, esta idea solo tiene 20 años):



Problemas a resolver

Cómo integrar la invocación del servicio en el lenguaje de forma más o menos transparente.

- No olvide este aspecto importante: independientemente de lo que diseñe, otros tendrán que programarlo y usarlo.

Cómo intercambiar datos entre máquinas que podrían usar diferentes representaciones para distintos tipos de datos. Esto implica dos aspectos:

- formatos de tipos de datos (p. ej., orden de bytes en diferentes arquitecturas)
- estructuras de datos (que deben simplificarse y reconstruirse)

Cómo encontrar el servicio que realmente se desea entre una colección potencialmente grande de servicios y servidores.

- El objetivo es que el cliente no necesite saber necesariamente dónde reside el servidor ni qué servidor proporciona el servicio.

Cómo gestionar los errores en la invocación del servicio de forma más o menos elegante:

- servidor caído,
- comunicación interrumpida,
- servidor ocupado,
- solicitudes duplicadas...

Interoperabilidad

Al intercambiar datos entre clientes y servidores que residen en diferentes entornos (hardware o software), se debe garantizar que los datos tengan el formato adecuado:

- Orden de bytes: diferencias entre las arquitecturas little endian y big endian (bytes de orden superior al principio o al final en los tipos de datos básicos).
- Estructuras de datos: como árboles, tablas hash, matrices multidimensionales o registros, que deben aplanarse (convertirse en una cadena, por así decirlo) antes de enviarse.

Esto se logra mejor utilizando un formato de representación intermedio. El concepto de transformar los datos que se envían a un formato de representación intermedio y viceversa tiene diferentes nombres (equivalentes):

- Marshalling/Un-marshalling
- Serialización/Deserialización

El formato de representación intermedio suele depender del sistema. Por ejemplo:

- SUN RPC: XDR (Representación de Datos Externos)

Contar con un formato de representación intermedio simplifica el diseño; de lo contrario, un nodo deberá poder transformar los datos a cualquier formato posible.

Binding (vinculacion)

Un servicio es proporcionado por un servidor ubicado en una dirección IP específica y que escucha en un puerto determinado.

La vinculación es el proceso de asignar el nombre de un servicio a una dirección y un puerto que pueden utilizarse para fines de comunicación.

La vinculación puede realizarse:

- Localmente: el cliente debe conocer el nombre (dirección) del host del servidor.
- Distribuida: existe un servicio independiente (ubicación del servicio, nombre y servicios de directorio, etc.) encargado de asignar nombres y direcciones. Estos servicios deben ser accesibles para todos los participantes.

Con un enlazador distribuido, se pueden realizar varias operaciones generales:

- REGISTRO (Exportación de una interfaz): Un servidor puede registrar los nombres de los servicios y el puerto correspondiente.
- RETIRO: Un servidor puede retirar un servicio.
- BÚSQUEDA (Importación de una interfaz): Un cliente puede solicitar al enlazador la dirección y el puerto de un servicio determinado.

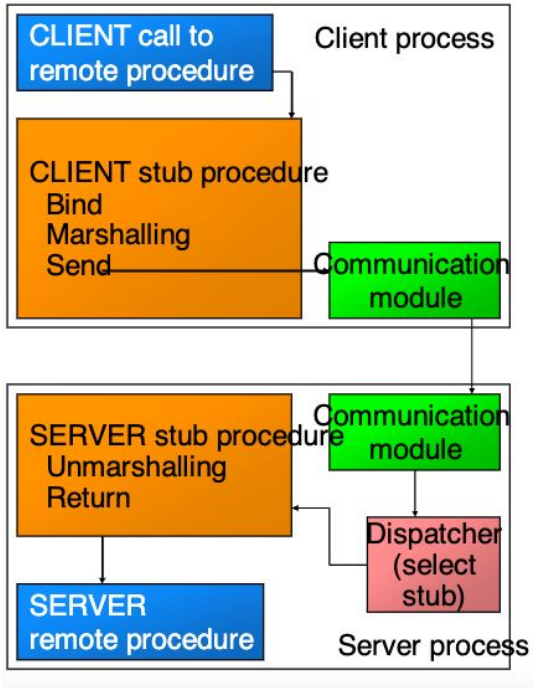
También debe existir una forma de localizar el enlazador (ubicación predefinida, variables de entorno, difusión a todos los nodos que lo buscan).

Como funciona en práctica

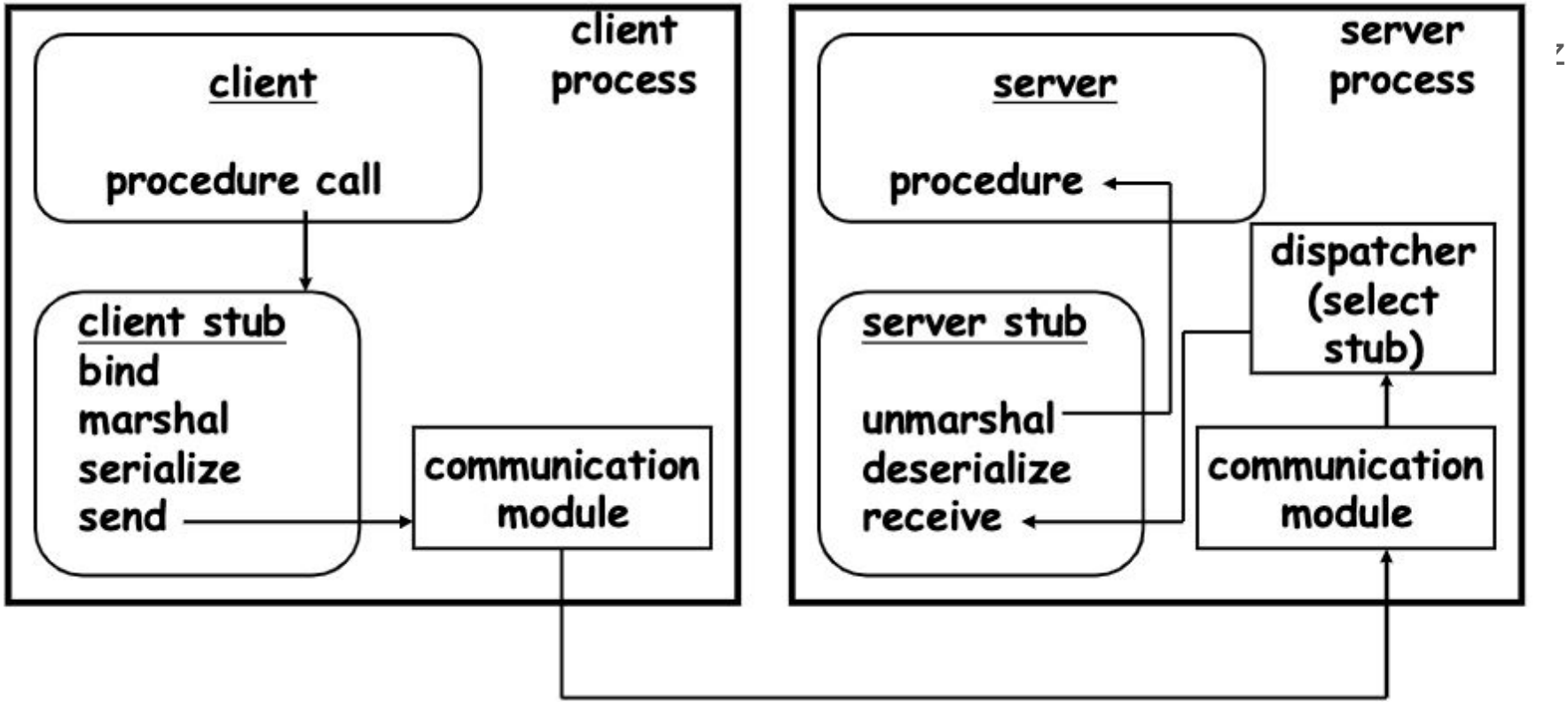
No se puede esperar que el programador implemente todos estos mecanismos cada vez que se desarrolla una aplicación distribuida. En cambio, estos son proporcionados por un sistema RPC (nuestro primer ejemplo de middleware de bajo nivel).

¿Qué hace un sistema RPC?

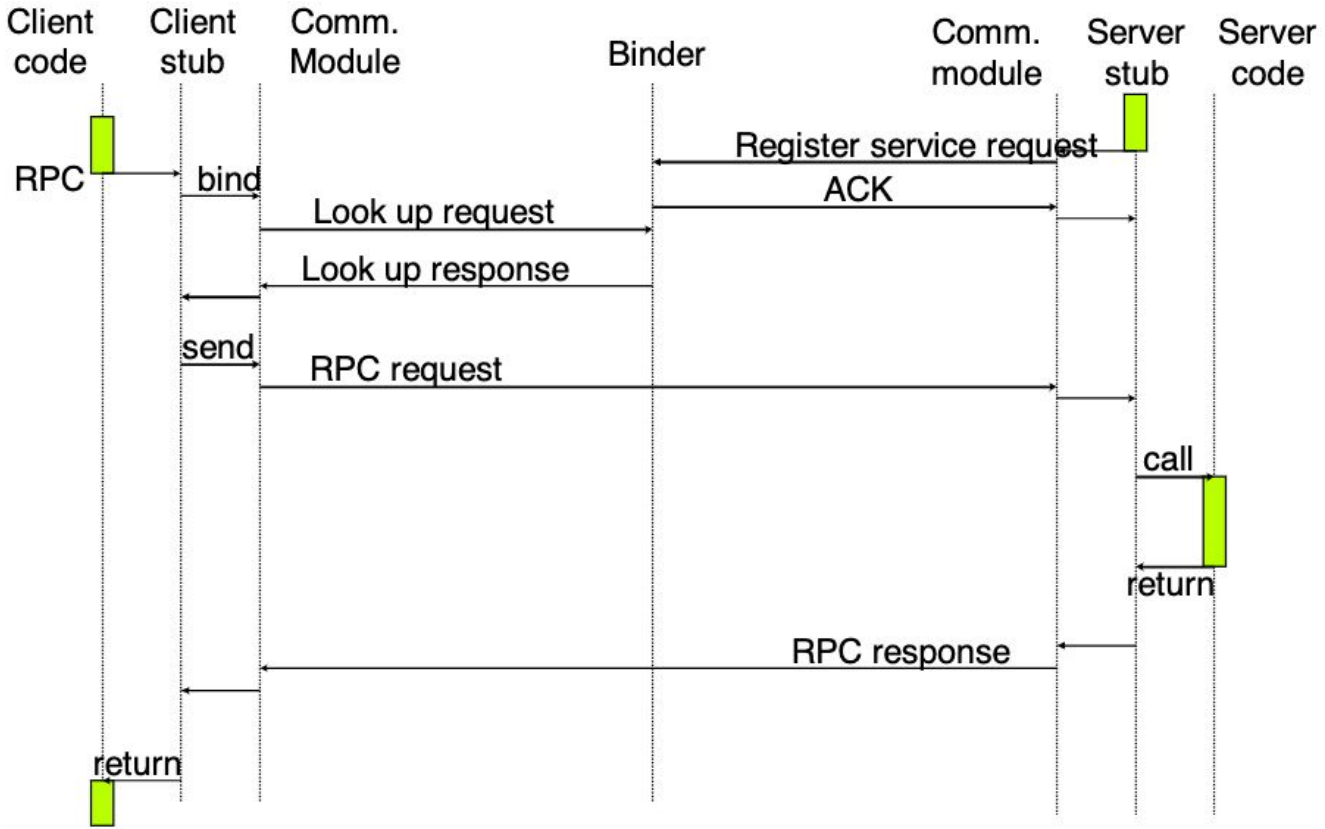
- Proporciona un lenguaje de definición de interfaz (IDL) para describir los servicios.
- Genera todo el código adicional necesario para realizar una llamada remota a un procedimiento y gestionar todos los aspectos de la comunicación.
- Proporciona un enlace en caso de que se cuente con un sistema de servicios de nombres y directorios distribuidos.



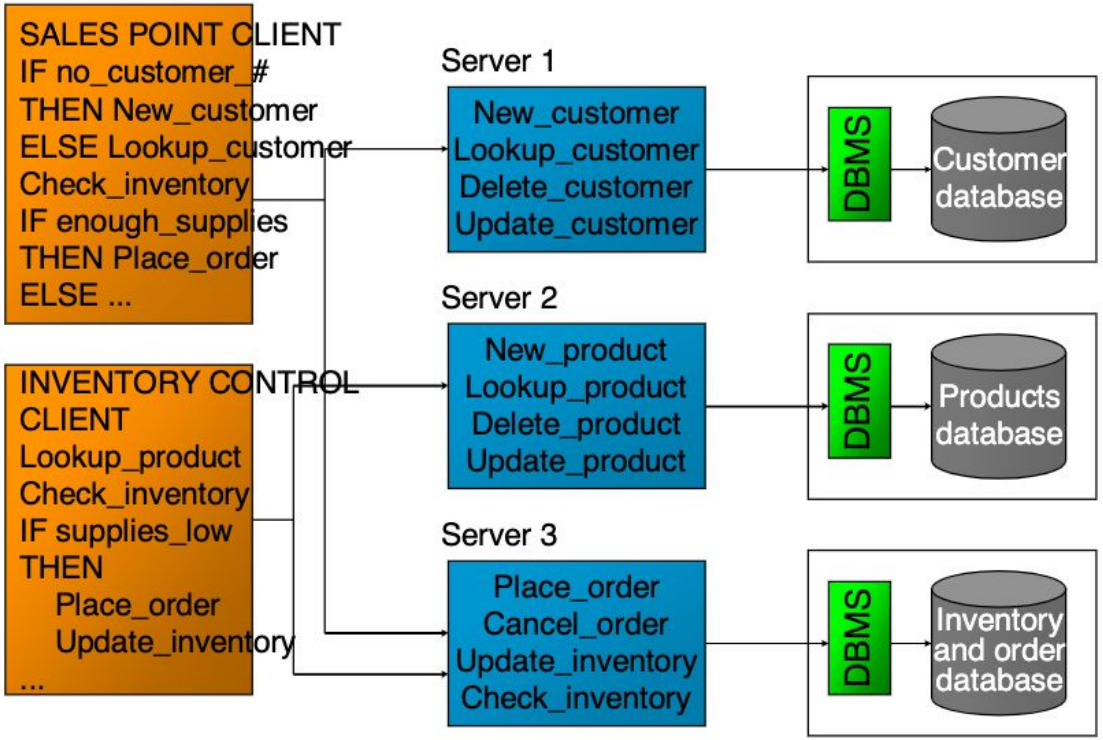
Como funciona en práctica



En mayor detalle



Aplicacion


SALES POINT CLIENT
IF no_customer_#
THEN New_customer
ELSE Lookup_customer
Check_inventory
IF enough_supplies
THEN Place_order
ELSE ...
INVENTORY CONTROL CLIENT
Lookup_product
Check_inventory
IF supplies_low
THEN
Place_order
Update_inventory
...
Server 1
New_customer
Lookup_customer
Delete_customer
Update_customer
Server 2
New_product
Lookup_product
Delete_product
Update_product
Server 3
Place_order
Cancel_order
Update_inventory
Check_inventory
DBMS
Customer database
DBMS
Products database
DBMS
Inventory and order database

Programando RPC

Se debe modificar el código del cliente

– Inicializar las opciones relacionadas con RPC

• Identificar el tipo de transporte

• Localizar el servidor/servicio

– Gestionar fallos en las llamadas a procedimientos remotos

• Funciones del servidor

– Generalmente requieren poca o ninguna modificación

Programando RPC

Compatibilidad con lenguajes

– Muchos lenguajes de programación no tienen un concepto a nivel de lenguaje de llamadas a procedimientos remotos (C, C++, Java <J2SE 5.0, ...)

• Estos compiladores no generarán automáticamente stubs de cliente y servidor.

– Algunos lenguajes tienen compatibilidad que permite RPC (Java, Python, Haskell, Go, Erlang).

• Sin embargo, es posible que debamos trabajar con entornos heterogéneos (p. ej., Java comunicándose con un servicio de Python).

Solución común

– Lenguaje de definición de interfaz (IDL): describe procedimientos remotos.

– Compilador independiente que genera stubs (precompilador).

Interface Definition Language (IDL)

Permite al programador especificar interfaces de procedimientos remotos (nombres, parámetros, valores de retorno).

• El precompilador puede usar esto para generar stubs de cliente y servidor.

– Código de serialización.

– Código de desmarshaling.

– Rutinas de transporte de red.

– Conformidad con la interfaz definida.

• Un IDL se parece a los prototipos de función.

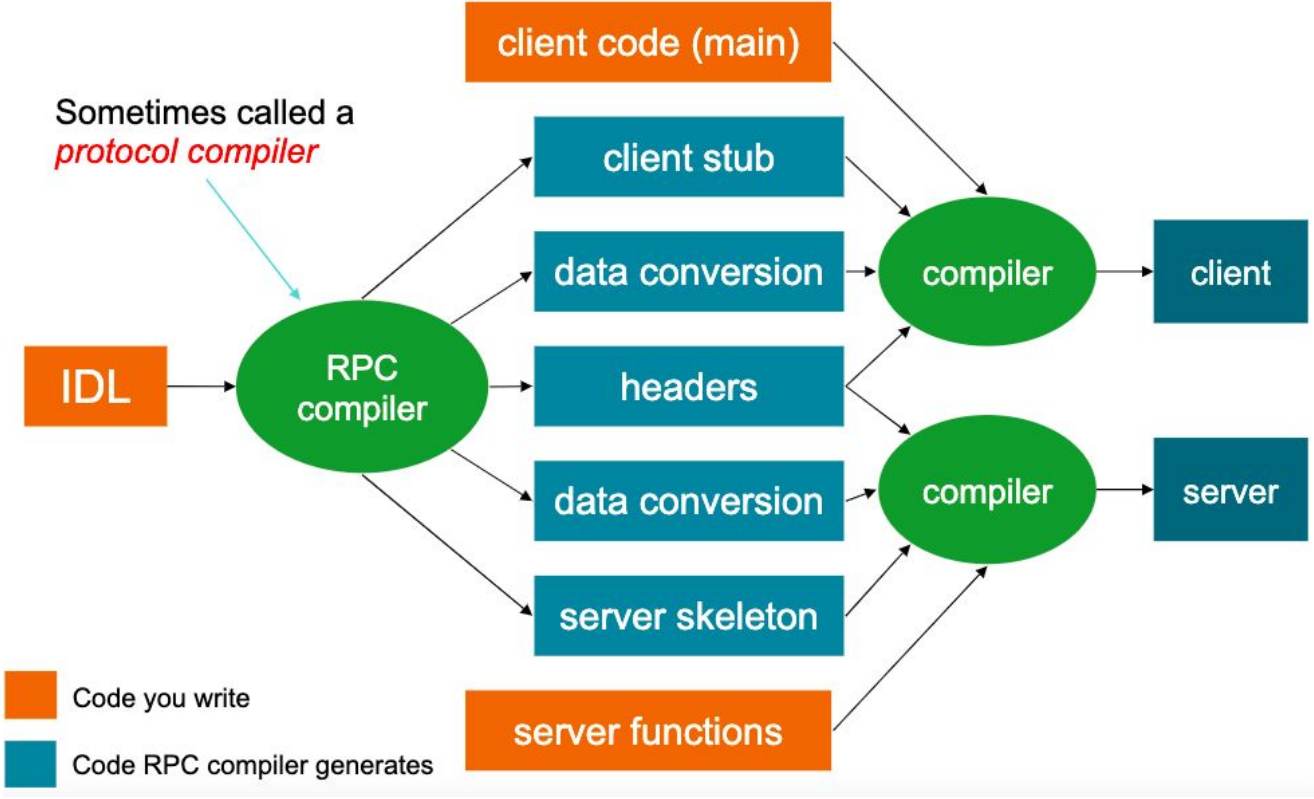
IDL

Se utiliza un compilador de interfaz para generar los stubs para clientes y servidores (rpcgen en SUN RPC). También puede generar encabezados de procedimiento que el programador puede utilizar para completar los detalles de la implementación.

Dada una especificación IDL, el compilador de interfaz realiza diversas tareas:

- Genera el procedimiento stub del cliente para cada firma de procedimiento en la interfaz. El stub se compila y enlaza con el código del cliente.
- Genera un stub del servidor. También puede crear un servidor principal, con el stub y el despachador compilados y enlazados a él. El diseñador puede extender este código escribiendo la implementación de los procedimientos. Podría generar un archivo *.h para importar la interfaz y todas las constantes y tipos necesarios.

Interface Definition Language (IDL)



Un proxy de cliente se parece a una función remota.

El stub de cliente tiene la misma interfaz que la función remota. Para el programador, se ve y se comporta como una función remota. Pero su función es:

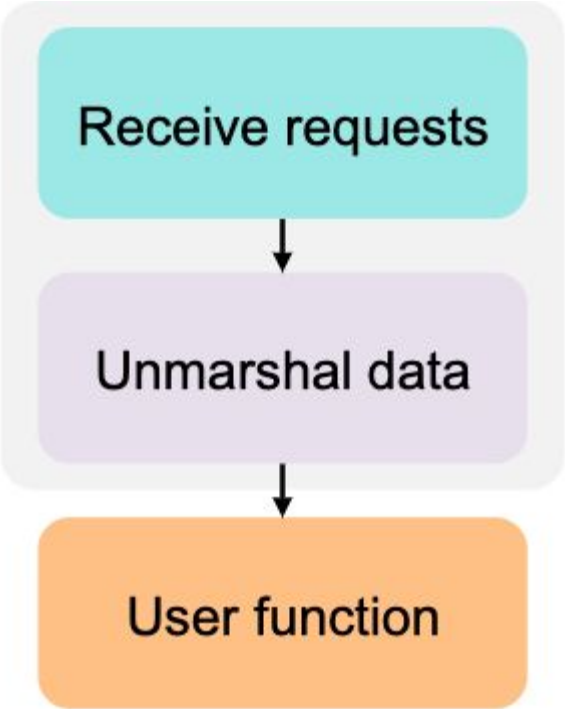
- Ordenar parámetros.
- Enviar el mensaje.
- Esperar una respuesta del servidor.
- Desenlazar la respuesta y devolver los datos apropiados.
- Generar excepciones si surgen problemas.

Un servidor STUB tiene dos partes:

1. Despachador: el receptor
    - Recibe las solicitudes del cliente
    - Identifica la función (método) apropiada
2. Esqueleto: el desmarshaller y el llamador
    - Desmarshalliza los parámetros
    - Llama al procedimiento del servidor local
    - Marshalliza la respuesta y la envía de vuelta al despachador

Todo esto es invisible para el programador

- El programador no se ocupa de nada de esto
- El despachador y el esqueleto pueden estar integrados

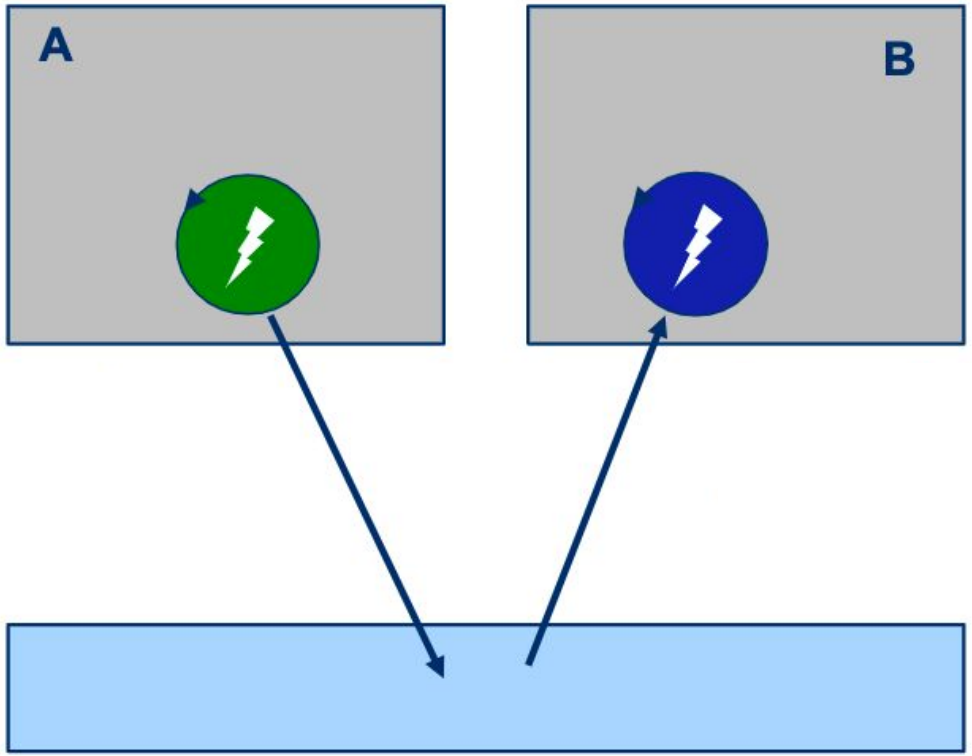


Cross-Domain calls

A: llamada al sistema para enviar un mensaje a B (p. ej., una cola de mensajes). Esperar respuesta.

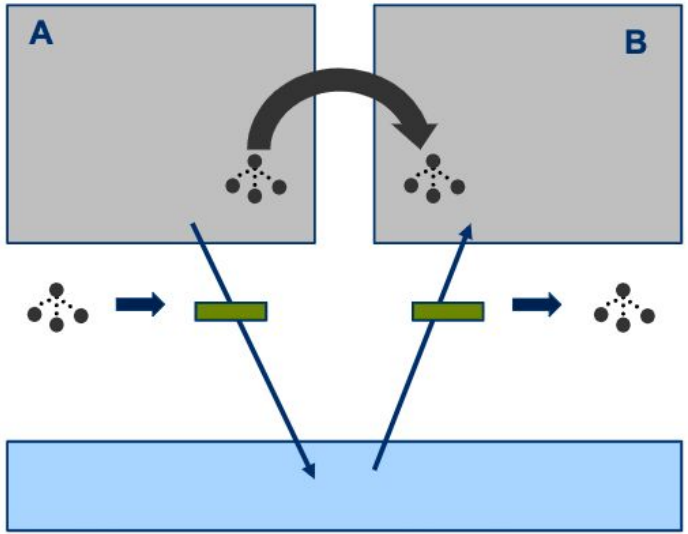
B: llamadas al sistema para recibir un mensaje entrante. Esperar solicitud.

Solicitud: bloque A, activación B.
Respuesta: bloque B, activación A.

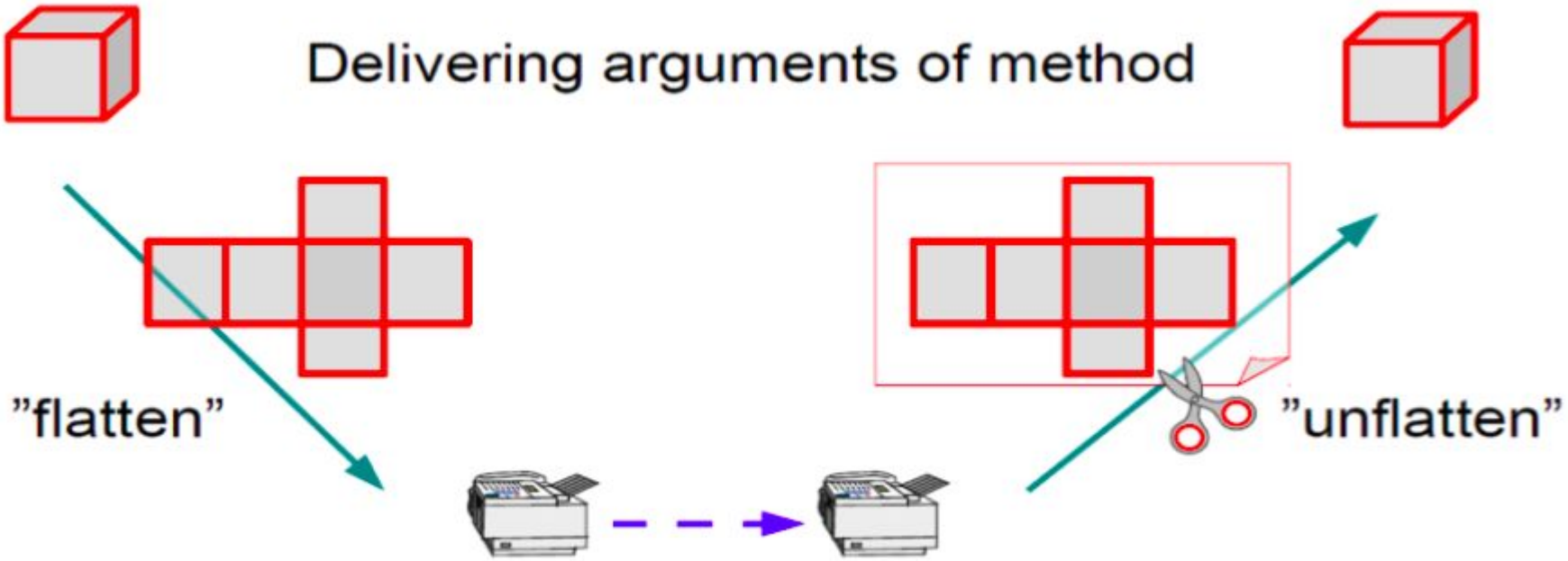


Marshalling (serializing)

¿Qué ocurre si los datos son una estructura enlazada compleja? Es necesario empaquetarlos como una secuencia de bytes en un mensaje y reconstruirlos en el otro lado.



Marshalling (serializing)



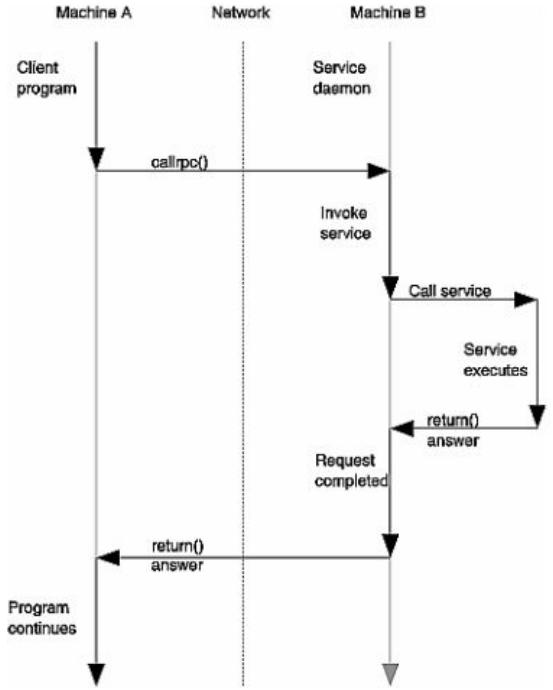
Ejecución

En general, RPC permite intercambios de solicitud/respuesta (por ejemplo, mediante mensajes a través de una red) que se asemejan a una llamada a procedimiento local.

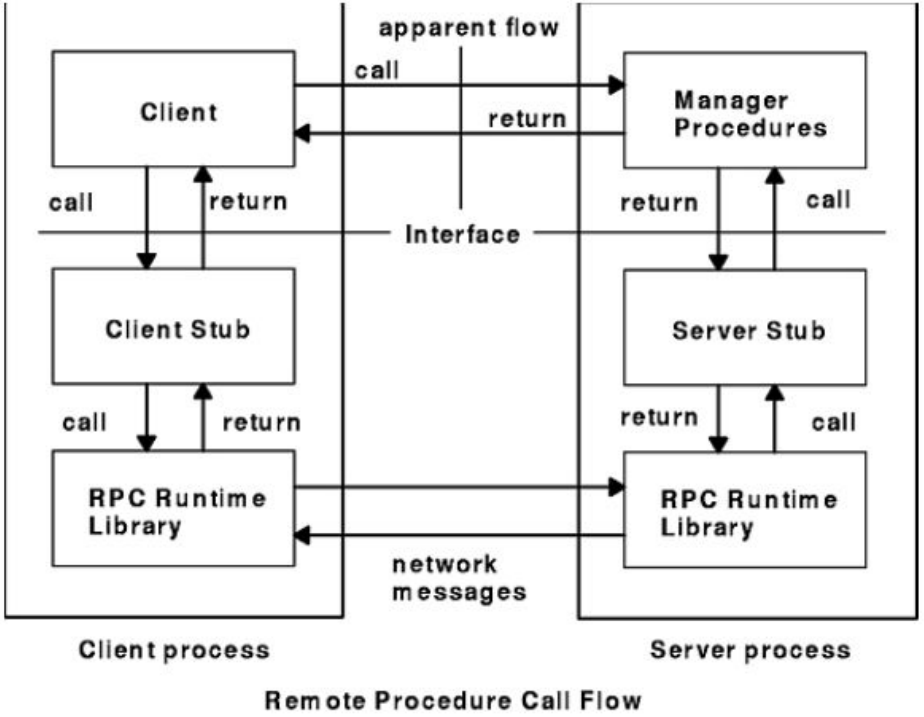
En Android, RPC permite una interacción flexible entre aplicaciones que se ejecutan en diferentes procesos, a través de los límites del kernel.

¿En qué se diferencia de una llamada a procedimiento local?

¿En qué se diferencia de una llamada al sistema?



RPC: Integración de lenguajes



Remote Procedure Call Flow

STUBS

Los stubs de RPC son procedimientos vinculados al cliente y al servidor.

– Los stubs de RPC son similares a los stubs de llamadas al sistema, pero su función va más allá de la simple captura al núcleo.

– Los stubs de RPC construyen/deconstruyen un mensaje transmitido a través de un sistema de mensajería.

– Binder es un ejemplo de este tipo de sistema de mensajería, implementado como un módulo de complemento del núcleo de Linux (un controlador) y algunas bibliotecas de espacio de usuario.

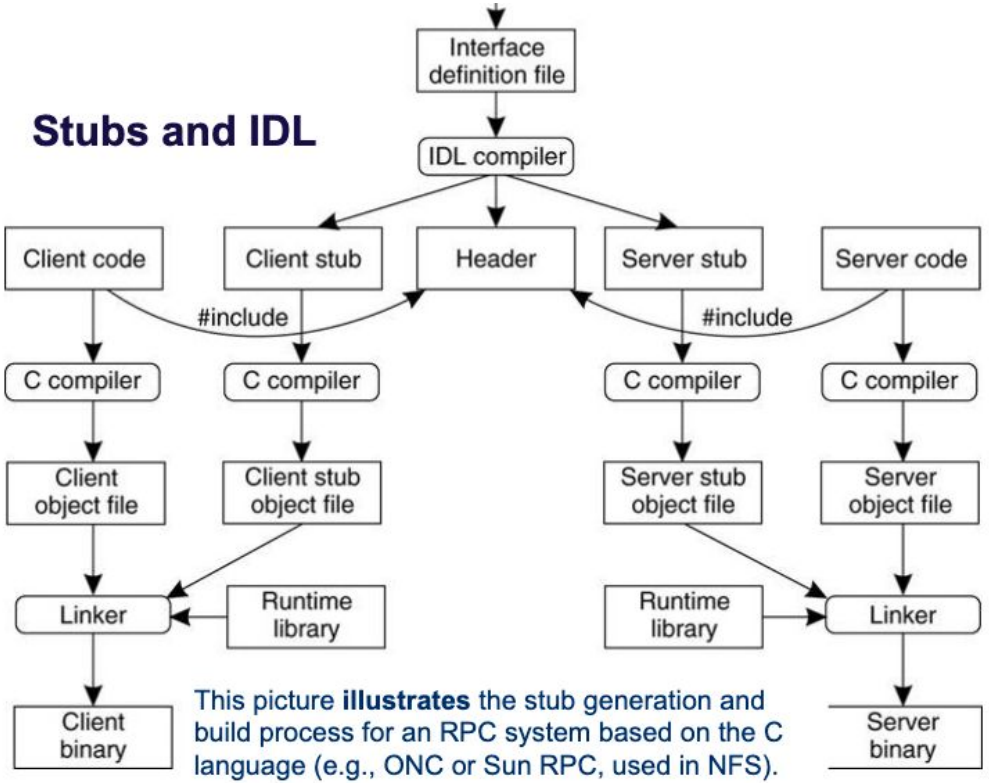
• Los stubs son generados por una herramienta que toma una descripción de la API de RPC de la aplicación escrita en un Lenguaje de Descripción de Interfaz (IDE).

– Se asemeja a cualquier definición de interfaz...

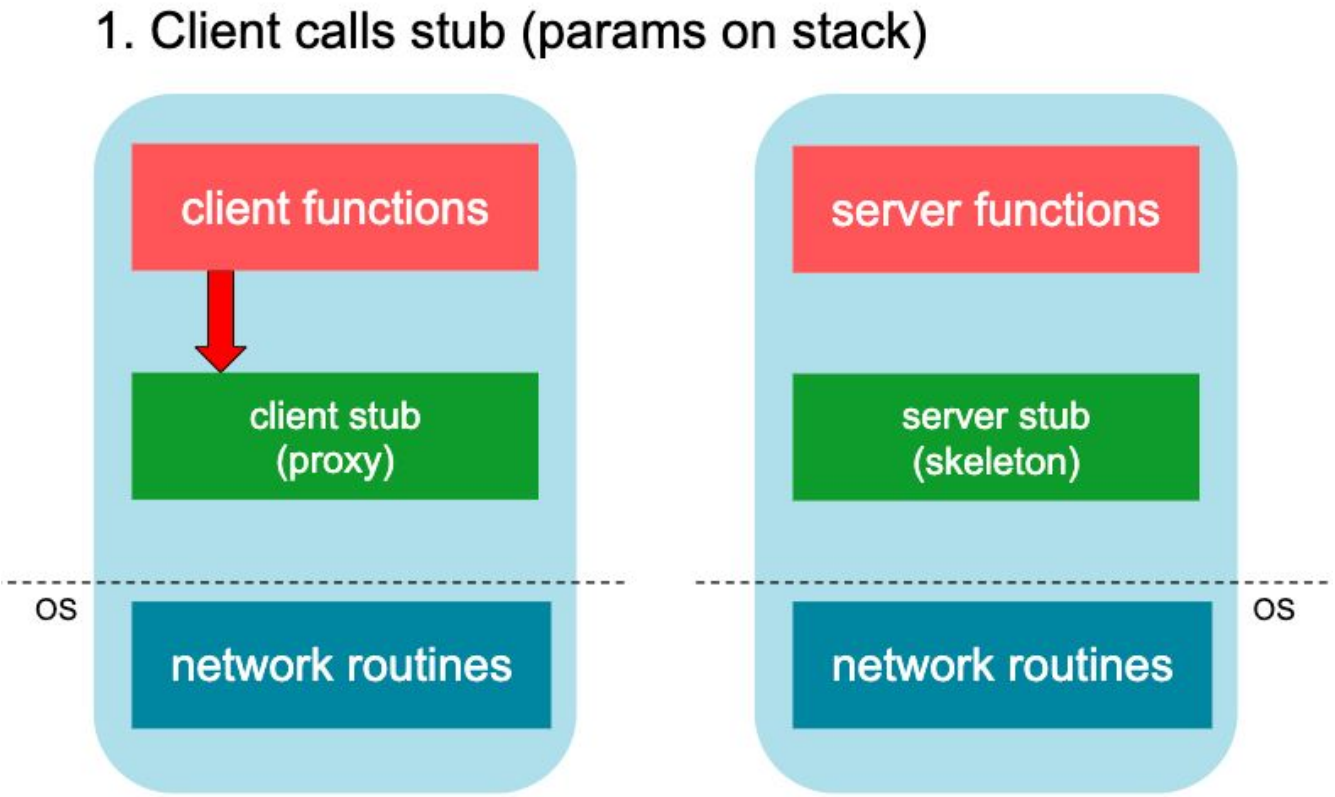
– Lista de nombres de métodos, tipos de argumentos/resultados y firmas.

– El código del stub ordena los argumentos en un mensaje de solicitud y los resultados en un mensaje de respuesta.

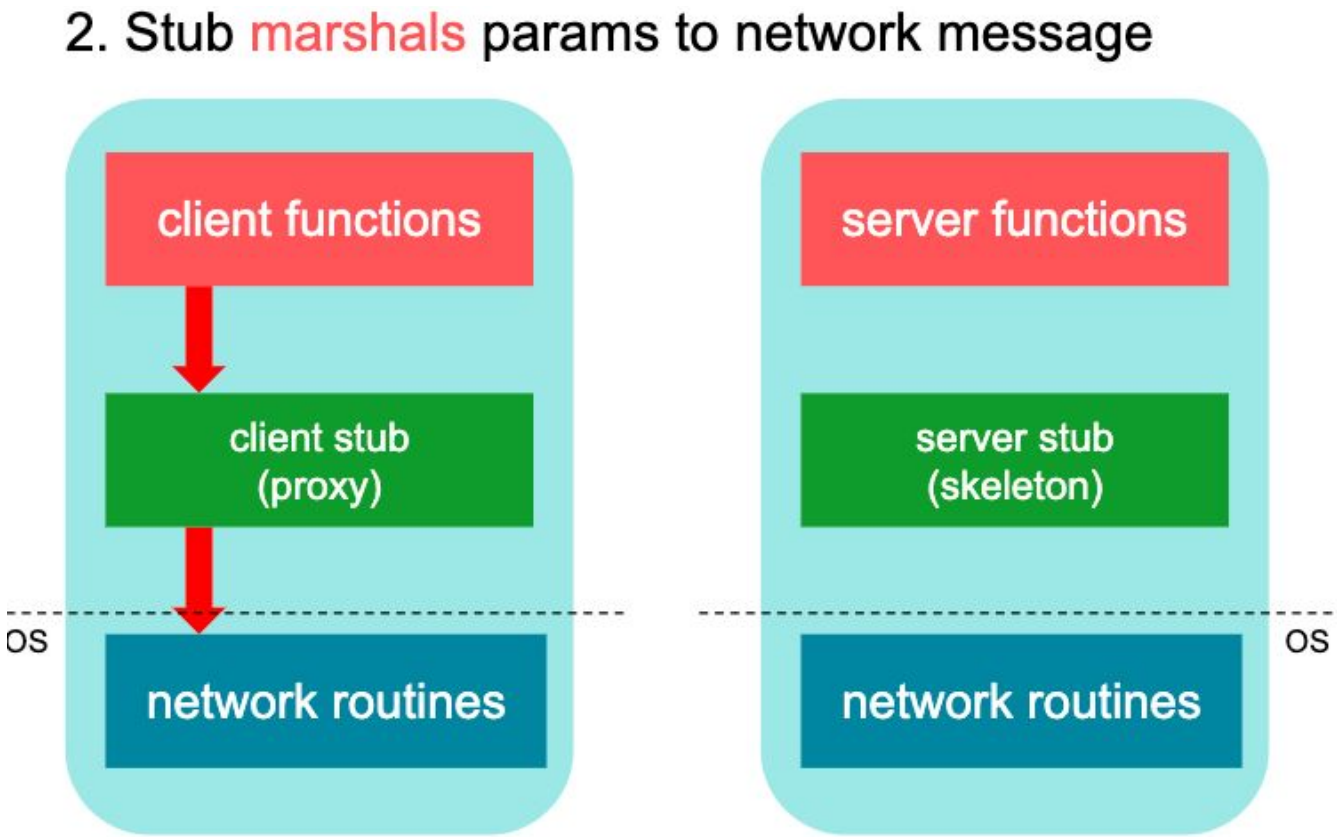
STUBS e IDL


Stubs and IDL
Interface definition file
IDL compiler
Client code
Client stub
Header
Server stub
Server code
#include
#include
C compiler
C compiler
C compiler
C compiler
Client object file
Client stub object file
Server stub object file
Server object file
Linker
Runtime library
Runtime library
Linker
Client binary
Server binary
This picture illustrates the stub generation and build process for an RPC system based on the C language (e.g., ONC or Sun RPC, used in NFS).

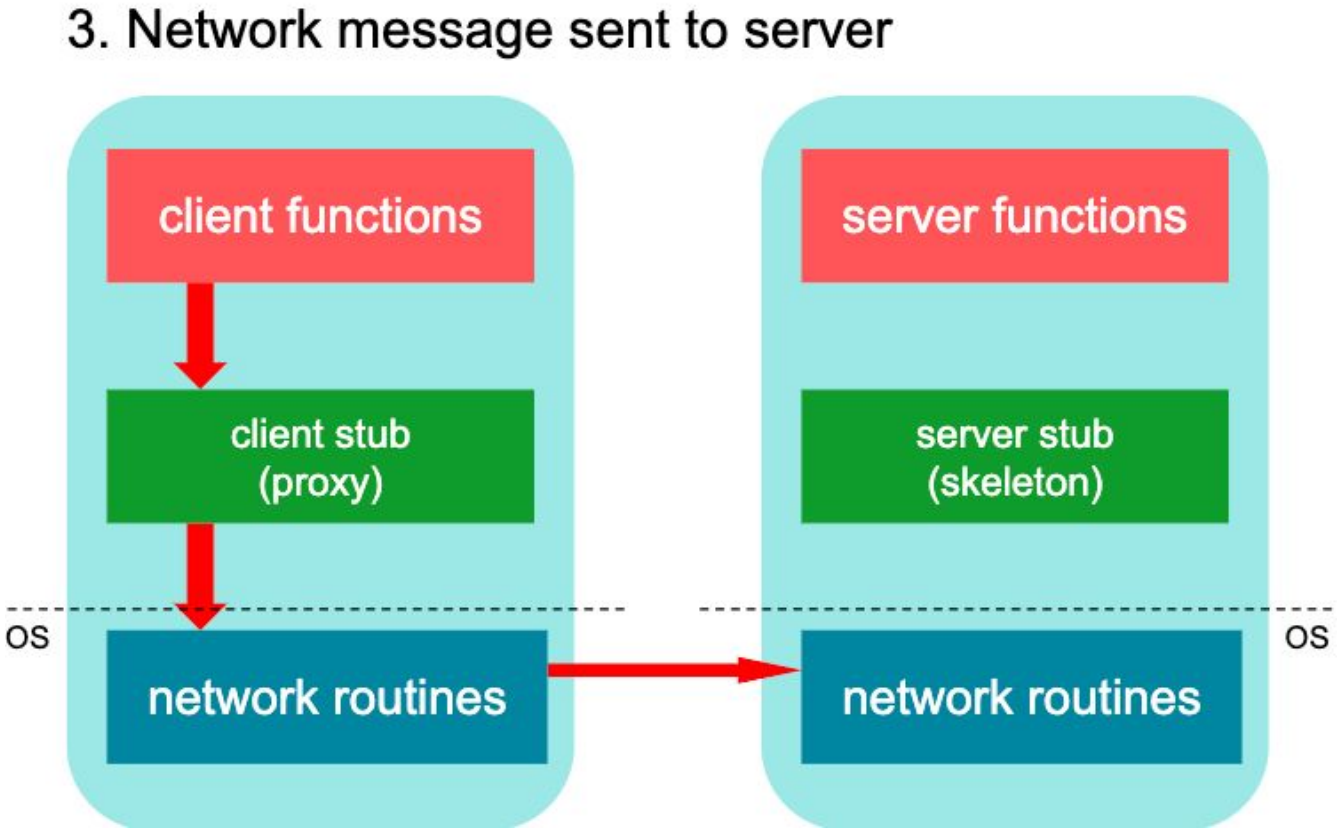
Funciones STUB



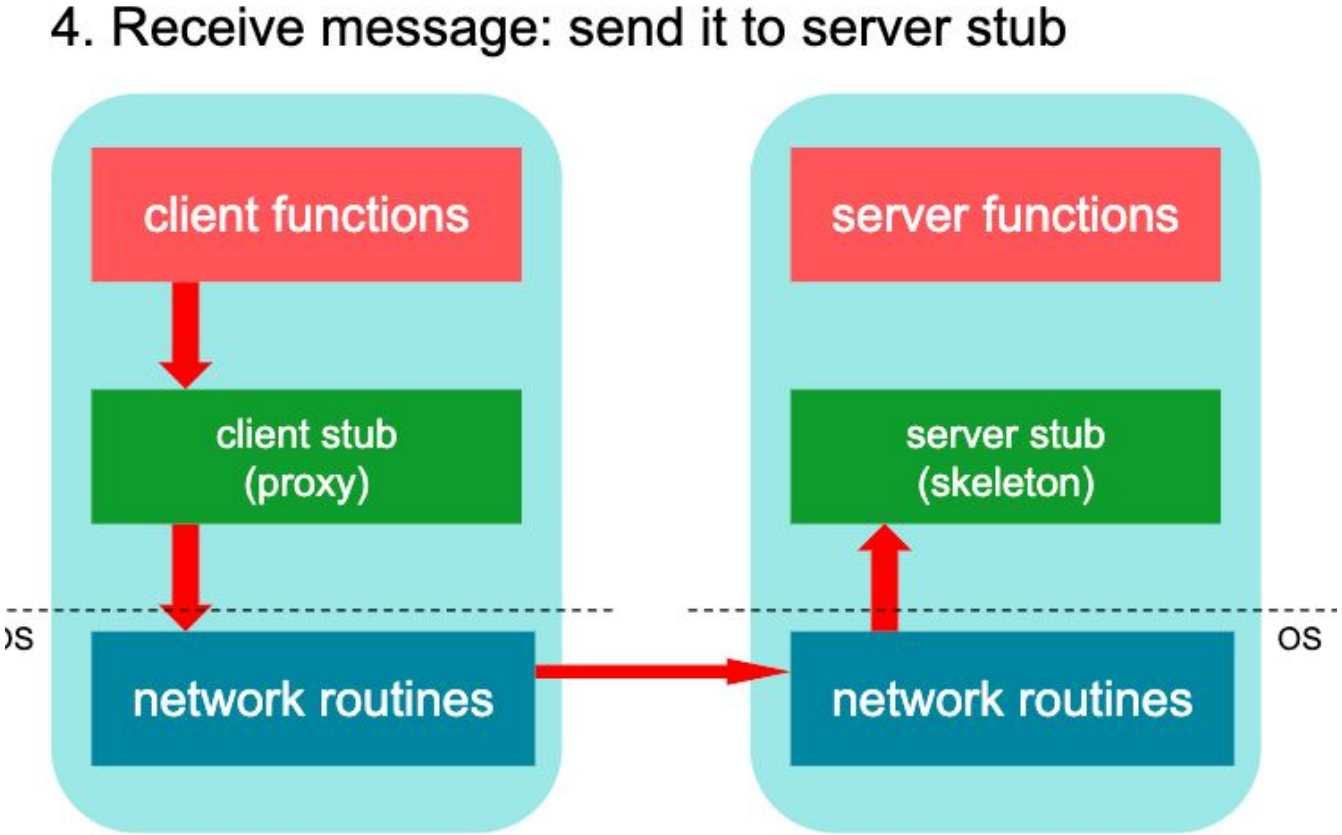
Funciones STUB



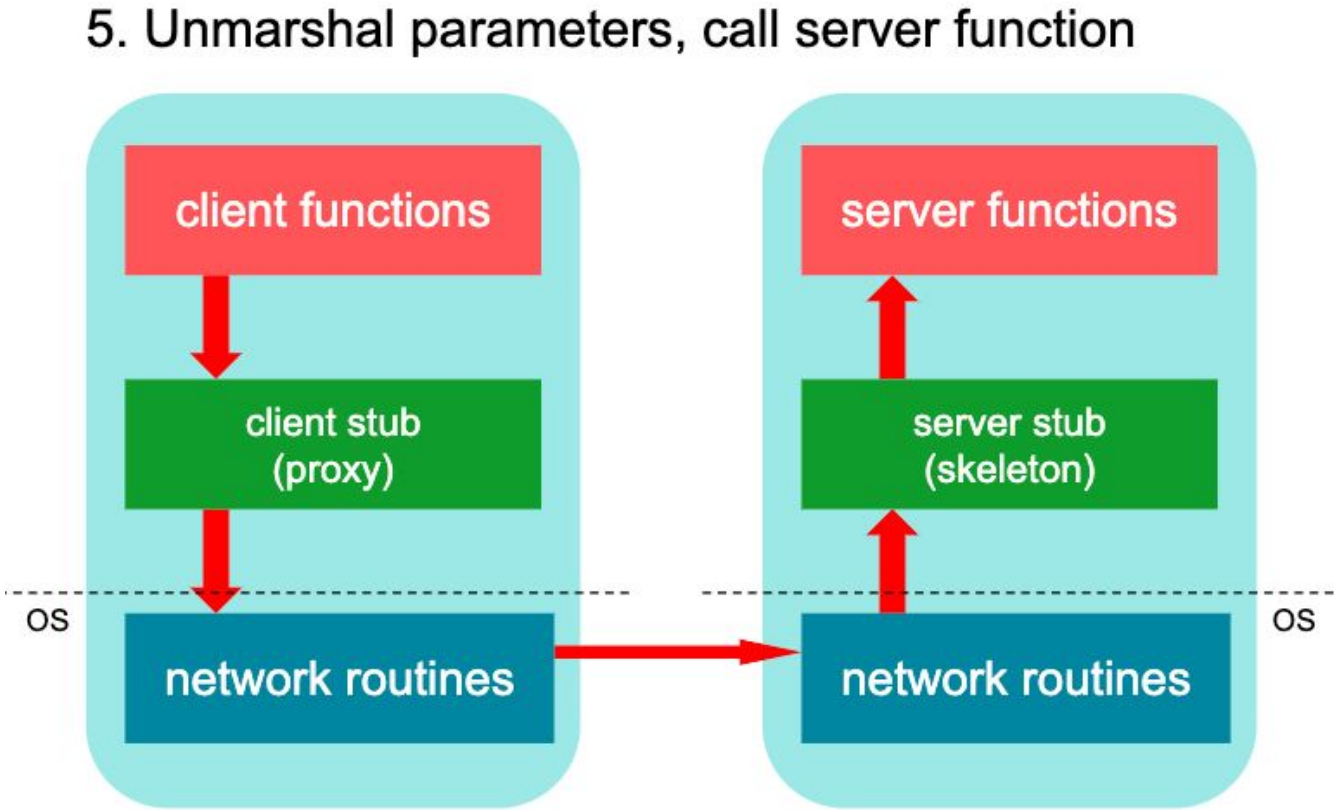
Funciones STUB



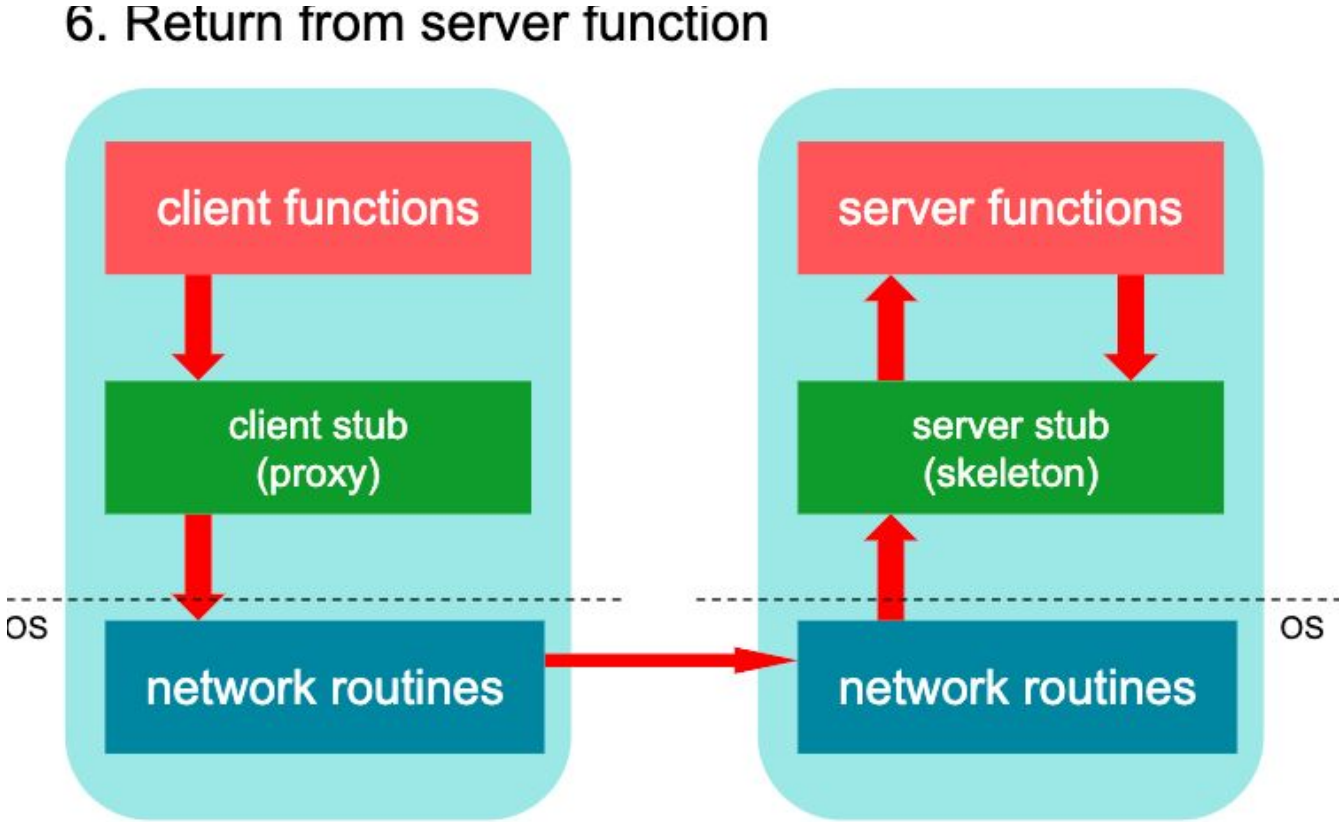
Funciones STUB



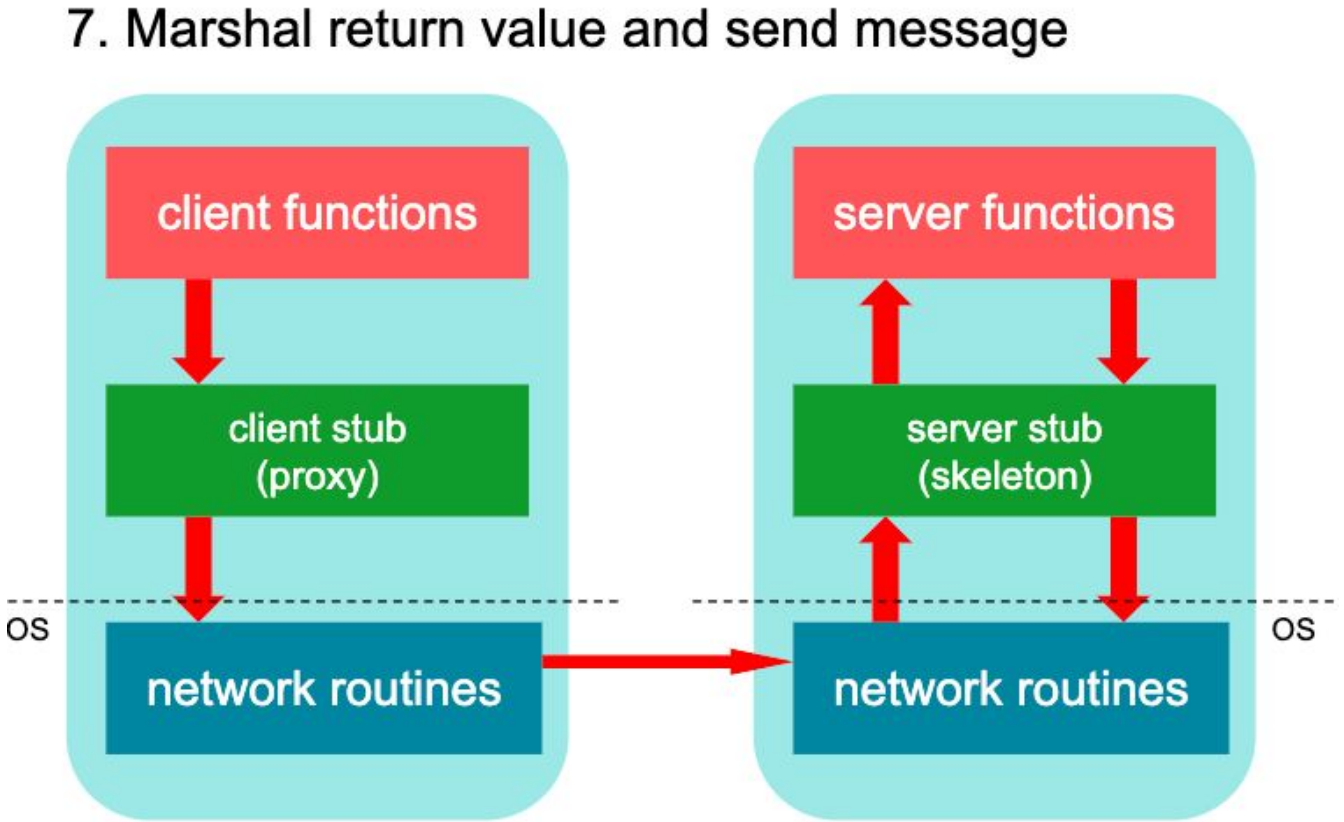
Funciones STUB



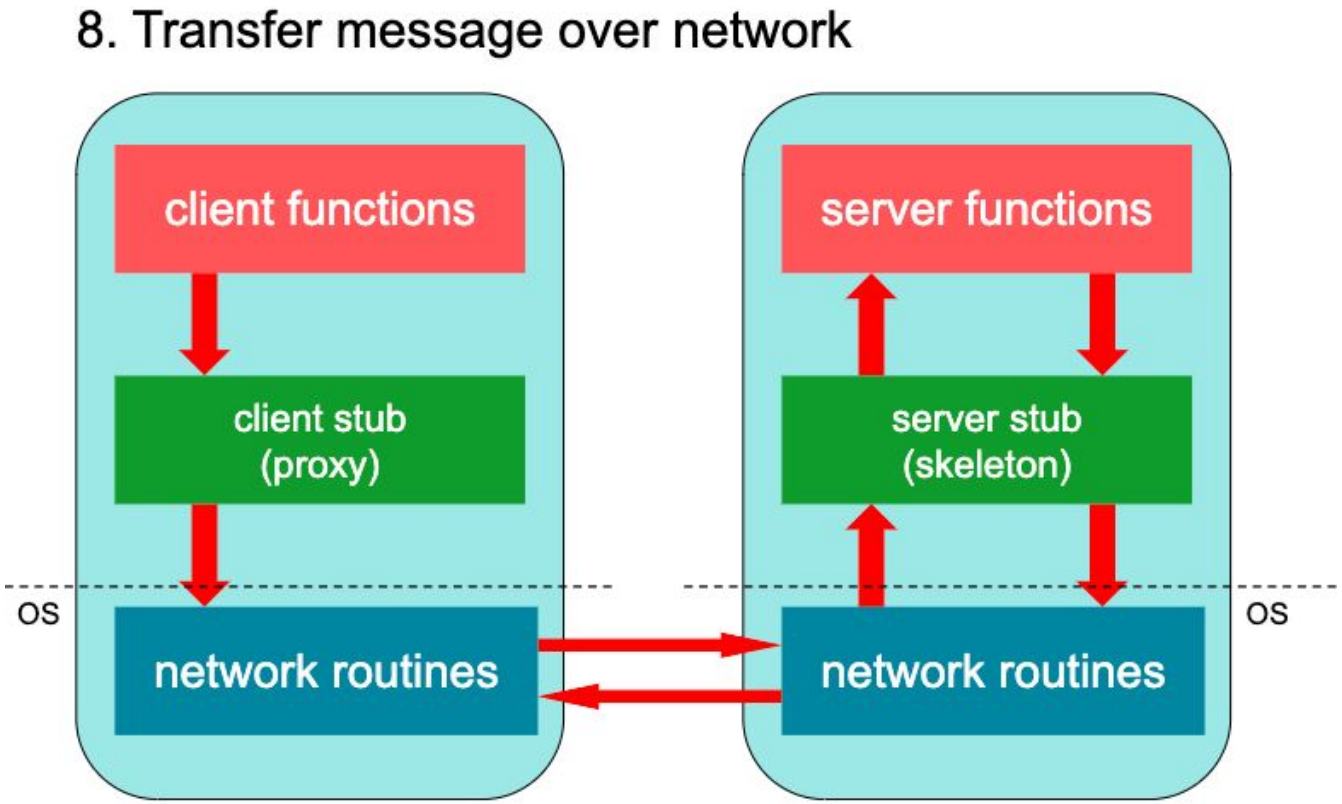
Funciones STUB



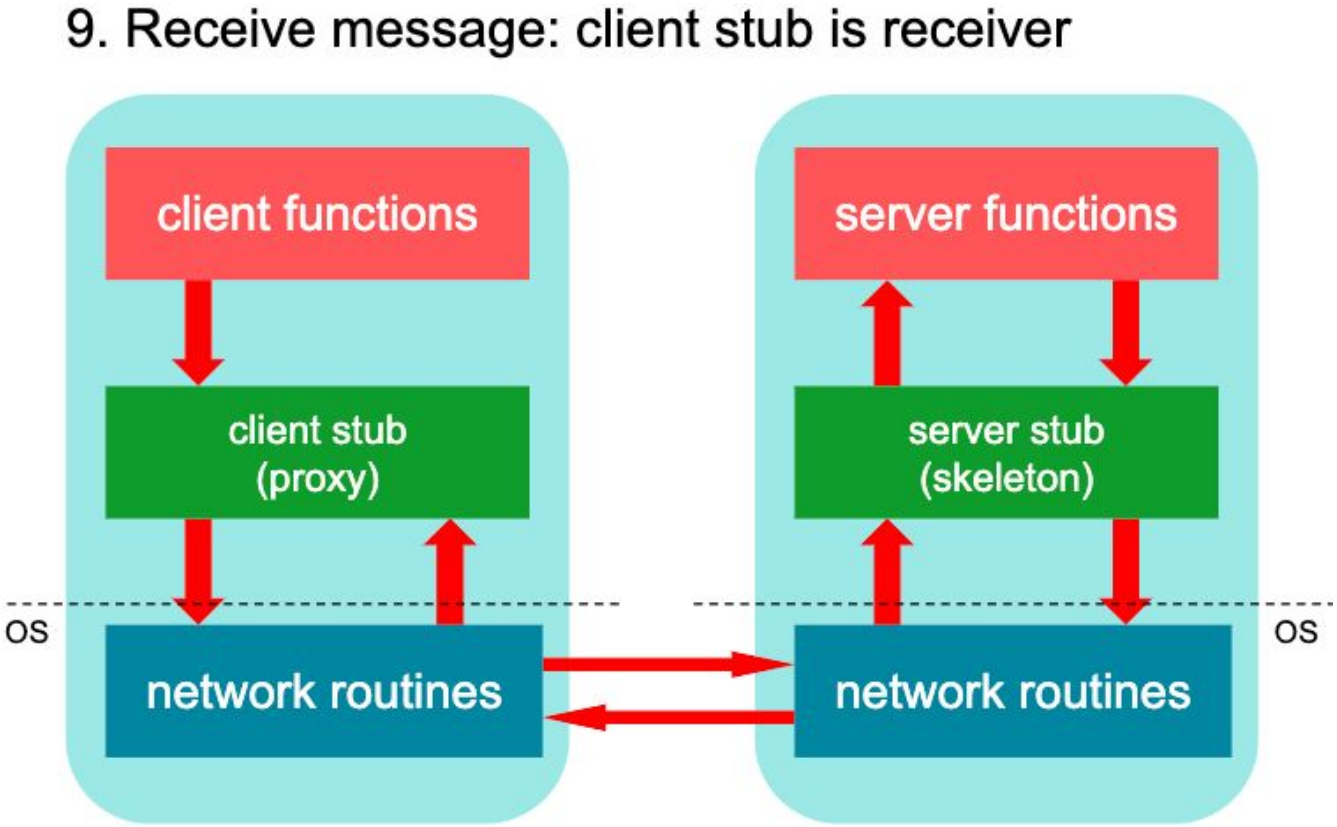
Funciones STUB



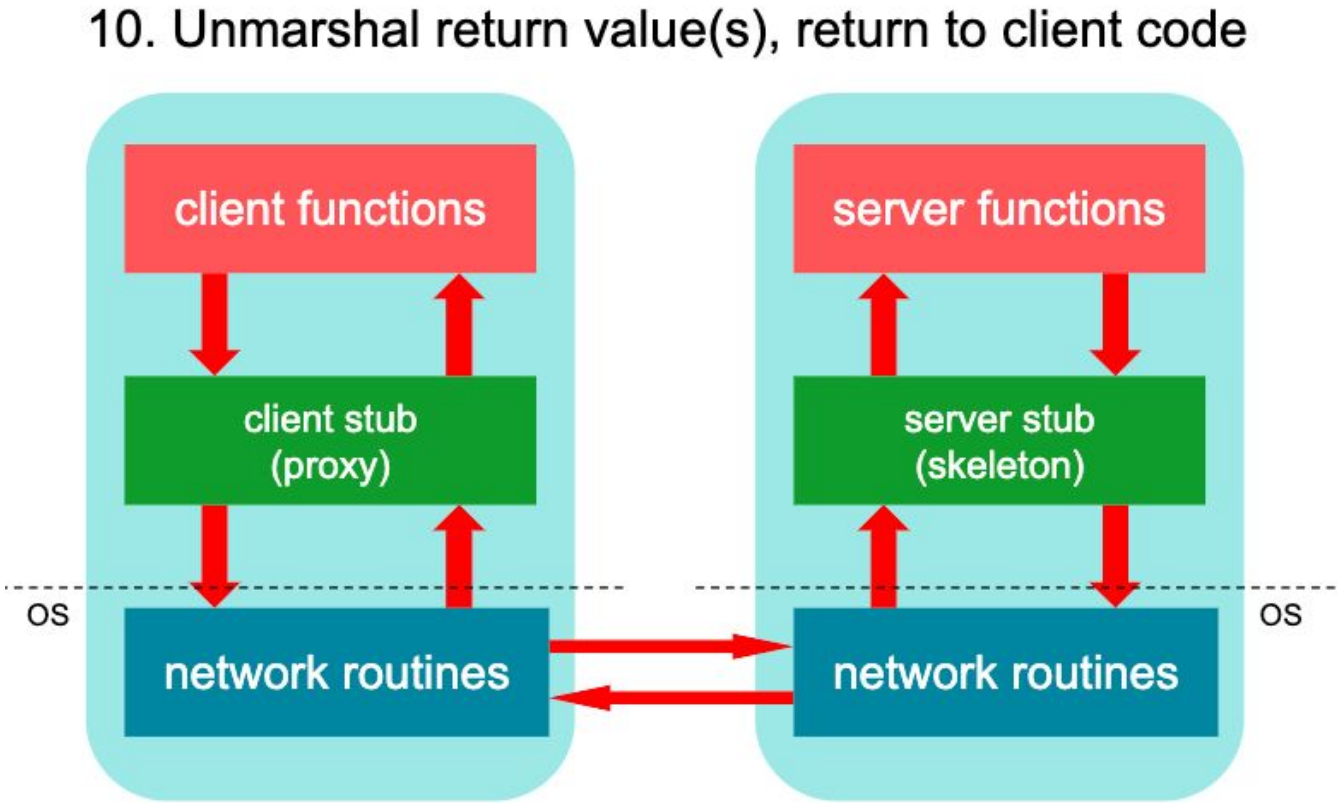
Funciones STUB



Funciones STUB



Funciones STUB



Implementando RPC

No hay soporte arquitectónico para llamadas a procedimientos remotos.

Simular con las herramientas disponibles (llamadas a procedimientos locales).

La simulación convierte a RPC en una construcción a nivel de lenguaje,

(El compilador crea código para enviar mensajes e invocar funciones remotas)

en lugar de una construcción del sistema operativo.

(El sistema operativo nos proporciona sockets)

Implementando RPC

El truco:

- Crear funciones auxiliares
- para que el usuario considere que la llamada es local.

En el cliente.

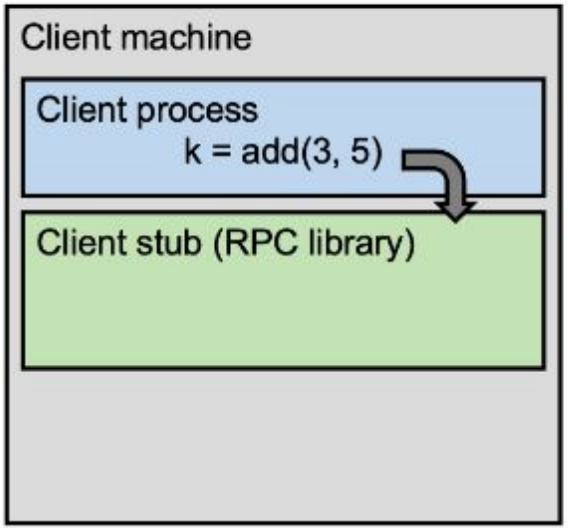
- La función auxiliar (proxy) tiene la interfaz de la función.
- Empaqueta los parámetros y llama al servidor.

En el servidor.

- La función auxiliar (esqueleto) recibe la solicitud y llama a la función local.

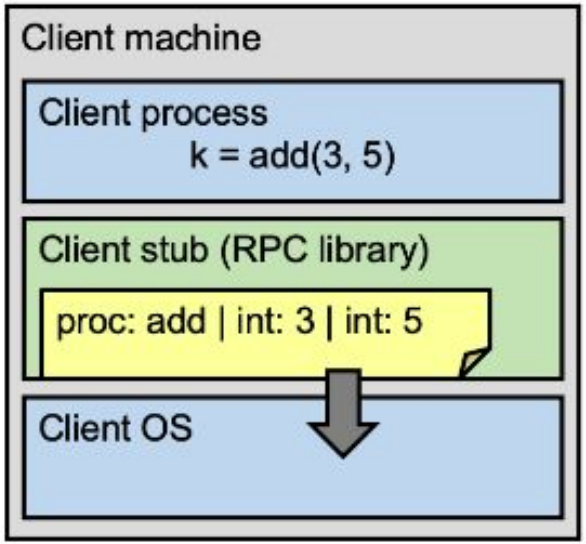
De nuevo…

1. **Client calls stub function (pushes parameters onto stack)**



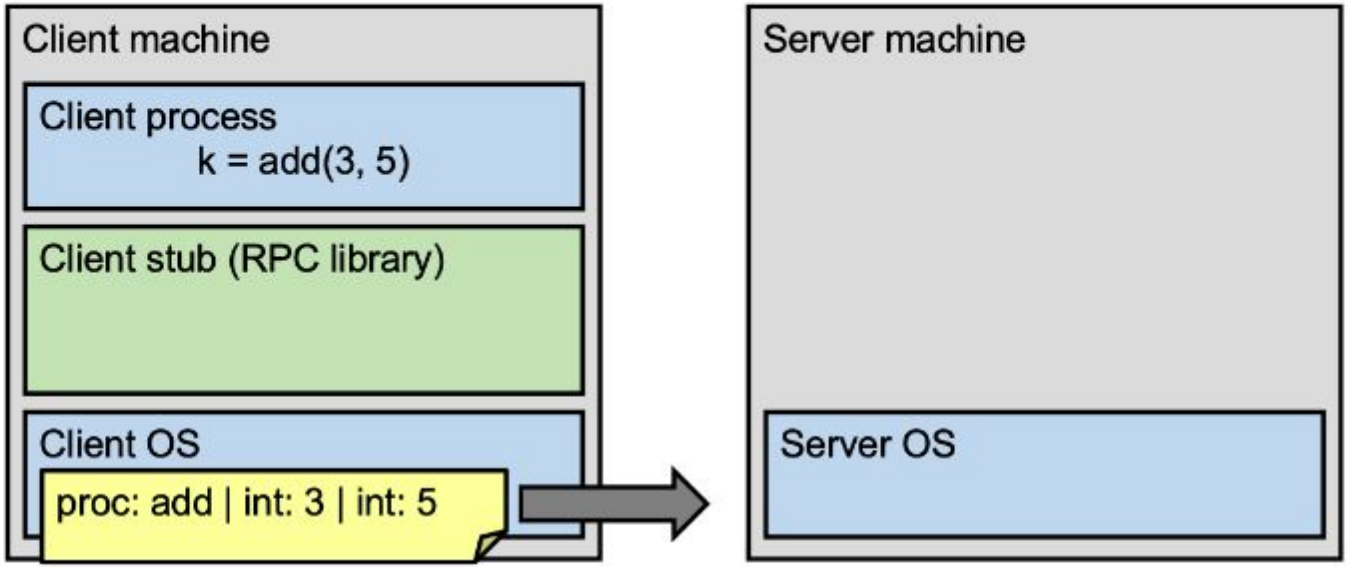
De nuevo…

1. Client calls stub function (pushes parameters onto stack)
2. **Stub marshals parameters to a network message**



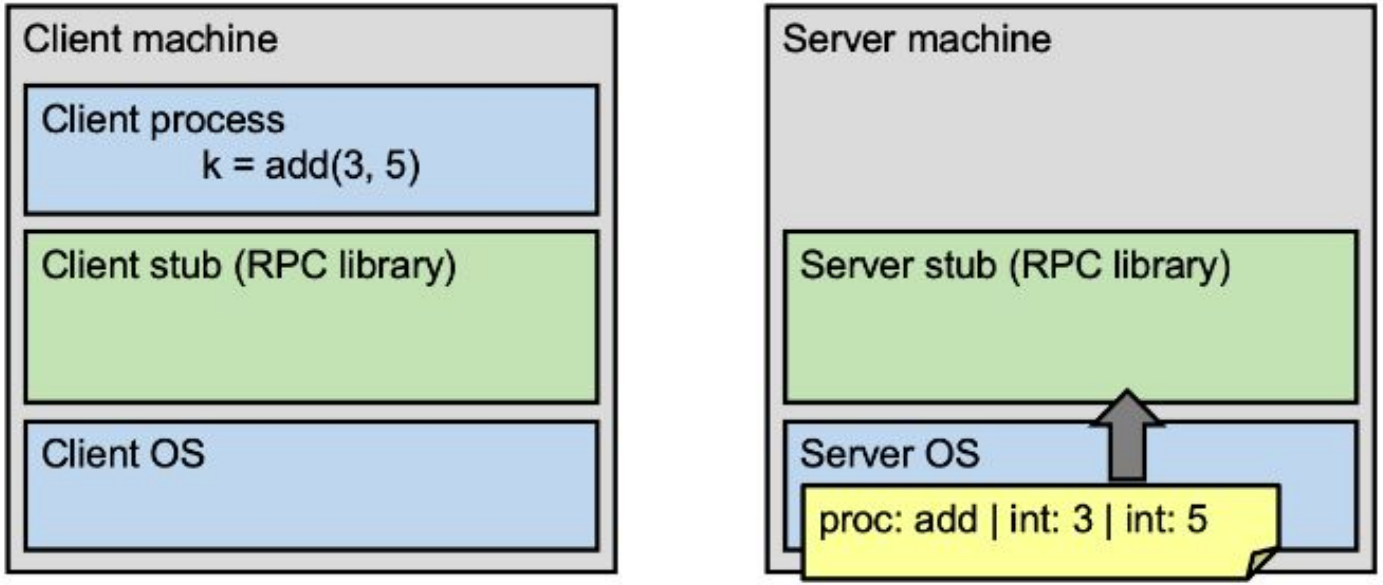
De nuevo…

2. Stub marshals parameters to a network message
3. **OS sends a network message to the server**



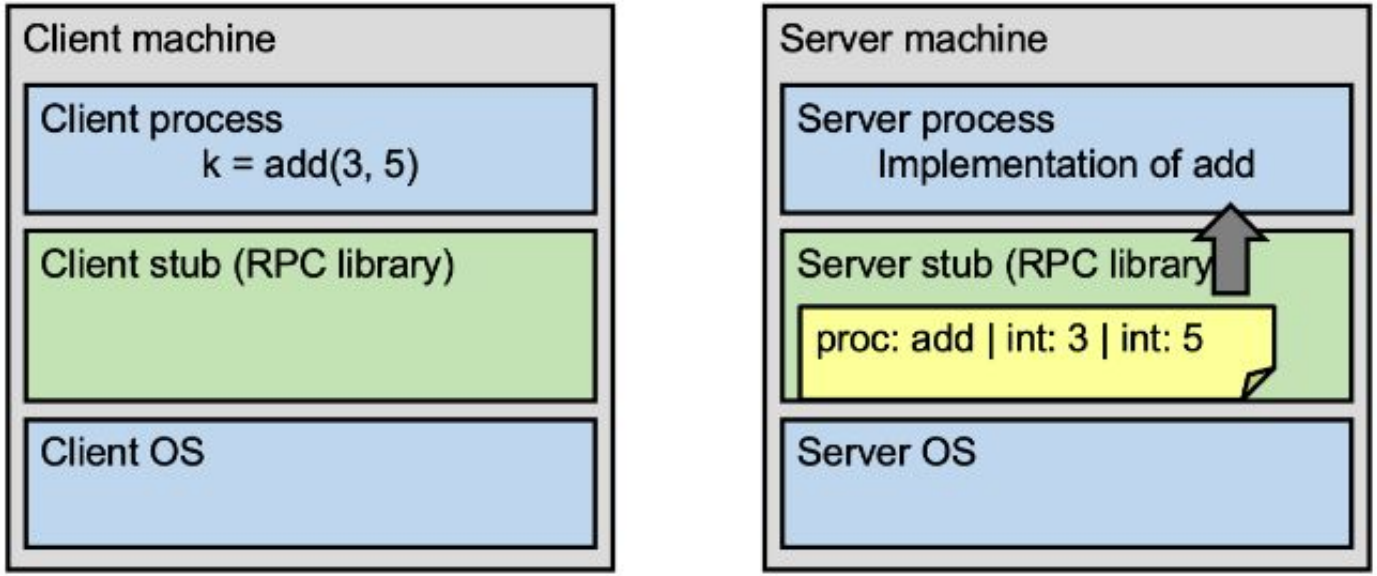
De nuevo…

3. OS sends a network message to the server
4. **Server OS receives message, sends it up to stub**



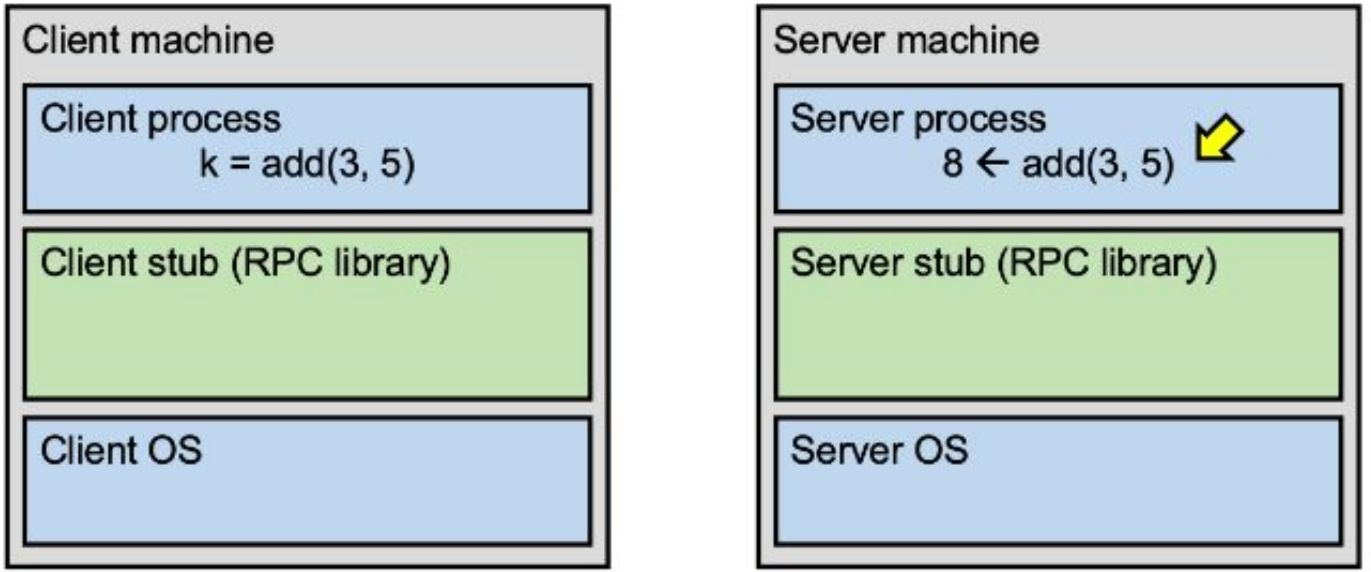
De nuevo…

4. Server OS receives message, sends it up to stub
5. **Server stub unmarshals params, calls server function**



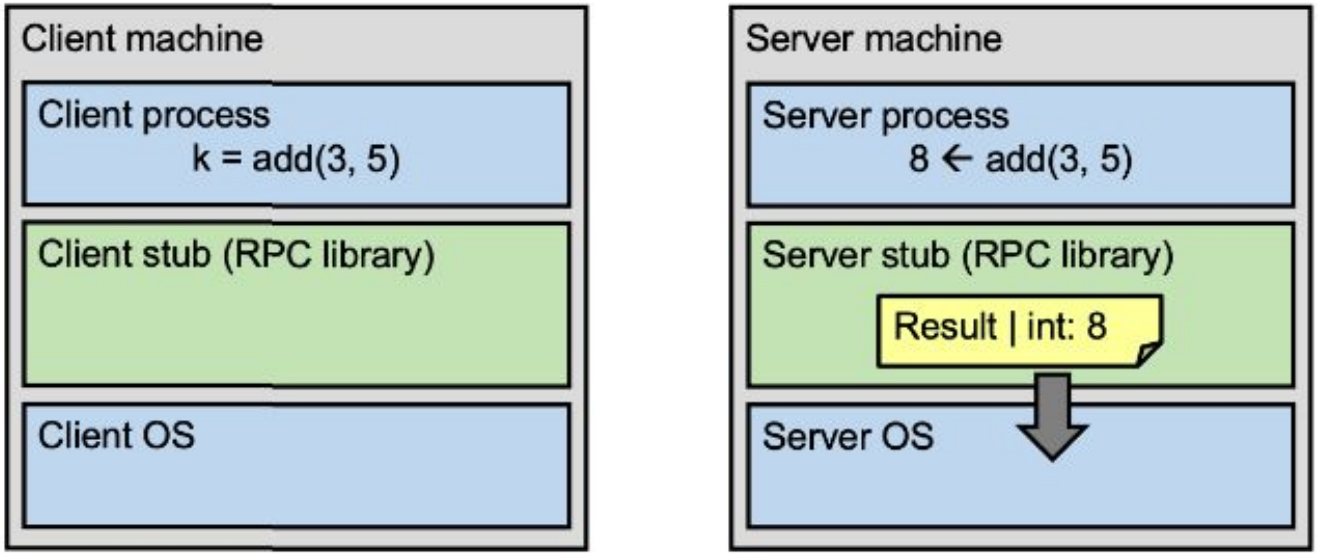
De nuevo…

5. Server stub unmarshals params, calls server function
6. **Server function runs, returns a value**



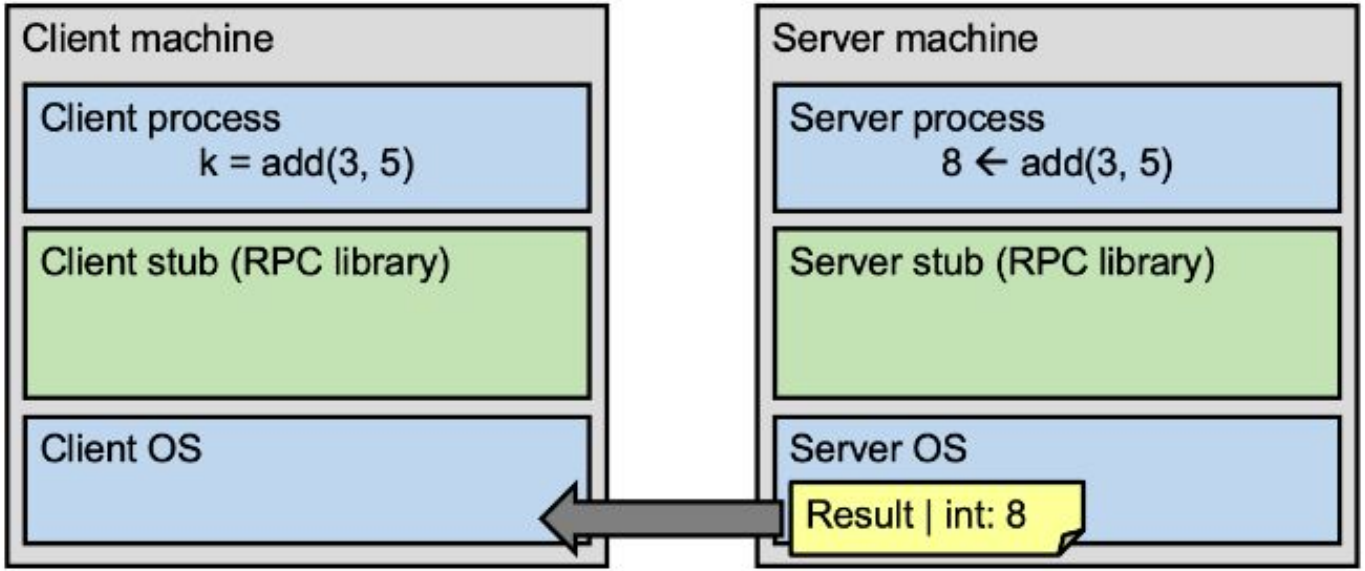
De nuevo…

6. Server function runs, returns a value
7. **Server stub marshals the return value, sends message**



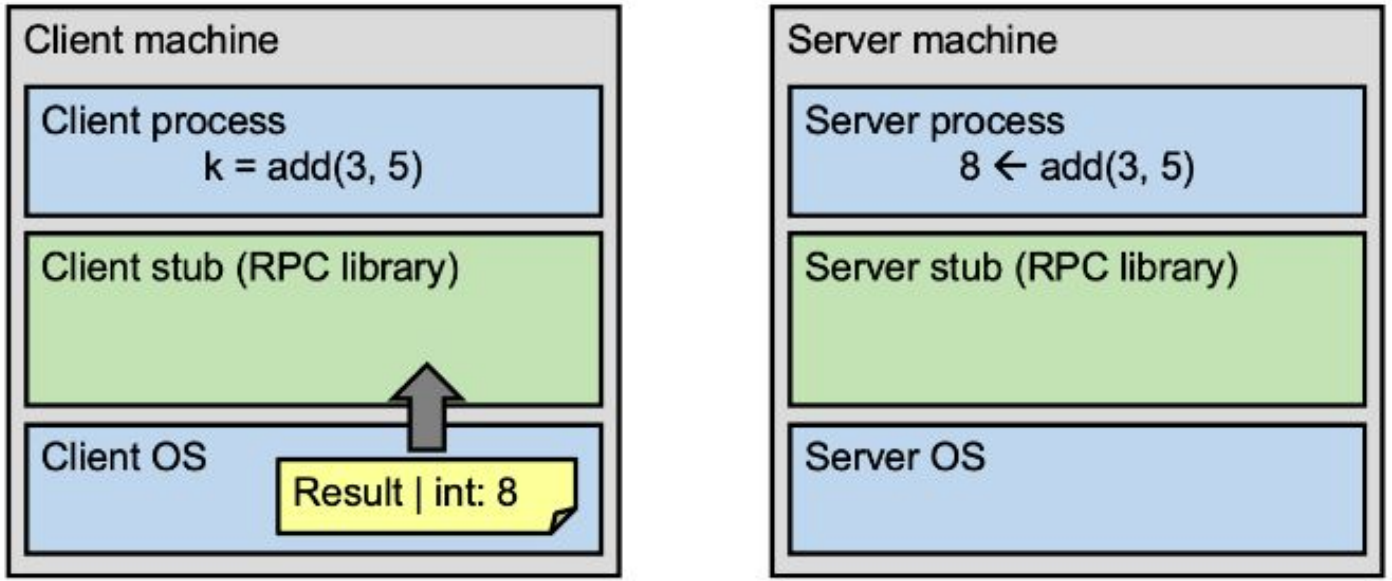
De nuevo…

7. Server stub marshals the return value, sends message
8. **Server OS sends the reply back across the network**



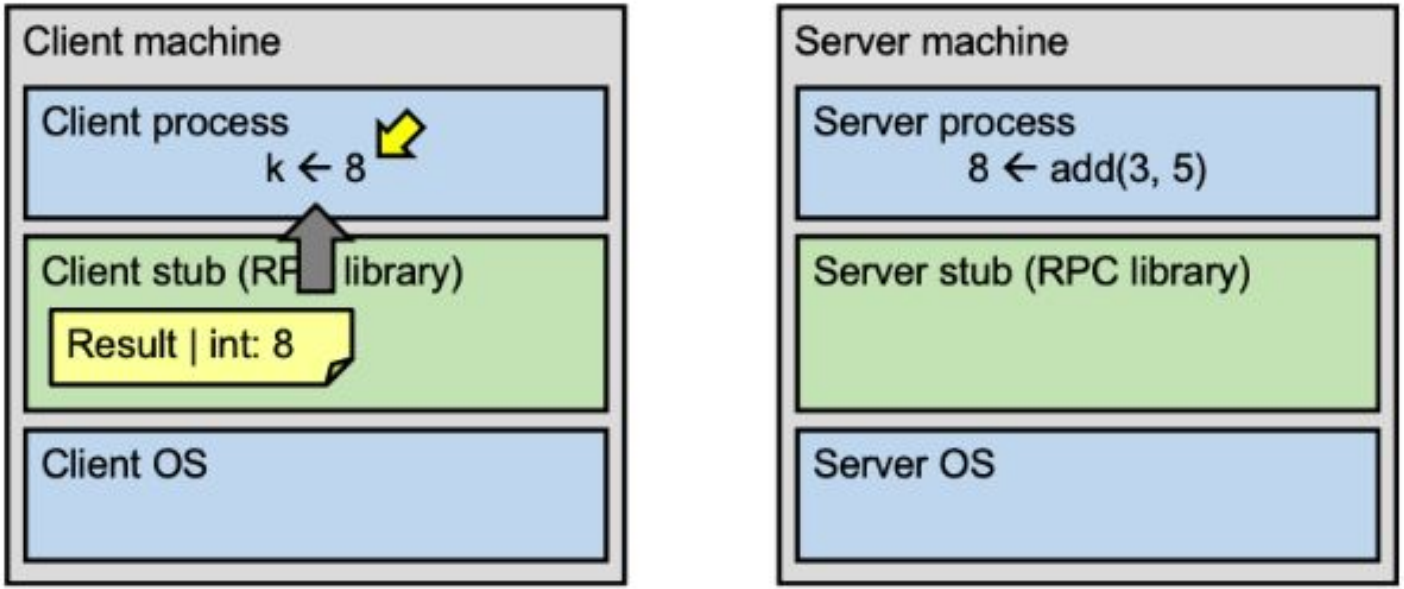
De nuevo…

8. Server OS sends the reply back across the network
9. **Client OS receives the reply and passes up to stub**



De nuevo…

9. Client OS receives the reply and passes up to stub
10. **Client stub unmarshals return value, returns to client**



RPC en perspectiva

VENTAJAS

RPC proporcionó un mecanismo para implementar aplicaciones distribuidas de forma sencilla y eficiente.

RPC seguía las técnicas de programación de la época (lenguajes procedimentales) y se integraba perfectamente con los lenguajes de programación más comunes (C), facilitando así su adopción por parte de los diseñadores de sistemas.

RPC permitió el diseño modular y jerárquico de grandes sistemas distribuidos:

- El cliente y el servidor son entidades independientes.
- El servidor encapsula y oculta los detalles de los sistemas back-end (como las bases de datos).

Beneficios RPC

RPC nos proporciona una interfaz de llamada a procedimientos.

• Simplifica la escritura de aplicaciones.

– RPC oculta todo el código de red en funciones auxiliares.

– Los programadores de aplicaciones no tienen que preocuparse por los detalles.

• Sockets, números de puerto, ordenación de bytes.

• ¿Dónde se ubica RPC en el modelo OSI?

– Capa 5: Capa de sesión: Gestión de conexiones.

– Capa 6: Presentación: Representación de datos.

– Utiliza la capa de transporte (4) para la comunicación (TCP/UDP).

RPC en perspectiva

DESVENTAJAS

RPC no es un estándar, es una idea que se ha implementado de muchas maneras diferentes (no necesariamente compatibles).

RPC permite a los diseñadores construir sistemas distribuidos, pero no resuelve muchos de los problemas que genera la distribución. En ese sentido, es solo una construcción de bajo nivel.

RPC se diseñó con un solo tipo de interacción en mente: cliente/servidor. Esto reflejaba las arquitecturas de hardware de la época, cuando la distribución implicaba pequeñas terminales conectadas a un mainframe.

A medida que el hardware y las redes evolucionaban, se necesitaba mayor flexibilidad.

RPC problemas del sistema

RPC fue una de las primeras herramientas que permitió el diseño modular de aplicaciones distribuidas.

Las implementaciones de RPC tienden a ser bastante eficientes, ya que no añaden demasiada sobrecarga. Sin embargo, un procedimiento remoto siempre es más lento que un procedimiento local:

- ¿Debería ser transparente un procedimiento remoto (idéntico a un procedimiento local)? (Sí: fácil de usar; No: aumentar la conciencia del programador)
- ¿Debería ser transparente la ubicación? (Sí: flexibilidad y tolerancia a fallos; No: diseño más sencillo, menor sobrecarga)
- ¿Debería haber un servidor de nombres centralizado (vinculador)?

RPC se puede utilizar para construir sistemas con muchas capas de abstracción.

Sin embargo, cada llamada RPC implica:

- Varios mensajes a través de la red
- Al menos un cambio de contexto (en el cliente cuando realiza la llamada, pero podría haber más)

Cuando una aplicación distribuida es compleja, se deben evitar las cadenas RPC profundas.

Retos/dificultades RPC

Paso de parámetros

- ¿Paso por valor o por referencia?
- Representación sin punteros

Enlace de servicios. ¿Cómo localizamos el endpoint del servidor?

- Base de datos central
- Base de datos de servicios por host

Protocolo de transporte

- ¿TCP? ¿UDP? ¿Ambos?

Cuando algo falla

- Posibilidades de fallo

Seguridad

- Los mensajes pueden ser visibles en la red. ¿Es necesario ocultarlos?
- ¿Autenticar al cliente?
- ¿Autenticar al servidor?

Representando datos

No existen problemas de incompatibilidad en el sistema local.

La máquina remota puede tener:

– Orden de bytes diferente.

– Tamaños diferentes de enteros y otros tipos.

– Representaciones de coma flotante diferentes.

– Conjuntos de caracteres diferentes.

– Requisitos de alineación.

Representando datos

IP (encabezados) obligó a todos a usar el ordenamiento de bytes big endian para valores de 16 y 32 bits.

Big endian: Byte más significativo en memoria baja.

– SPARC < V9, Motorola 680x0, PowerPC anterior.

Little endian: Byte más significativo en memoria alta.

– Intel/AMD IA-32, x64.

Bi-endian: El procesador puede operar en cualquier modo.

– ARM, PowerPC, MIPS, SPARC V9, IA-64 (Intel Itanium).

```
main() {
    unsigned int n;
    char *a = (char *)&n;

    n = 0x11223344;
    printf("%02x, %02x, %02x, %02x\n",
            a[0], a[1], a[2], a[3]);
}
```

Output on an Intel CPU:
44, 33, 22, 11

Output on a PowerPC:
11, 22, 33, 44

Representando datos

Se necesita una codificación estándar para permitir la comunicación entre sistemas heterogéneos.

• Serialización

– Convertir datos a un formato sin punteros: una matriz de bytes.

• Ejemplos

– XDR (Representación de Datos Externos), utilizado por ONC RPC.

– JSON (Notación de Objetos JavaScript).

– Lenguaje de Esquema XML del W3C.

– ASN.1 (Notación de Sintaxis Abstracta ISO).

– Buffers de Protocolo de Google.

XML: eXtensible Markup Language

```
<ShoppingCart>
    <Items>
        <Item>
            <ItemID> 00120 </ItemID>
            <Item> Bear Claw Black Telescopic Back Scratcher </Item>
            <Price> 5.99 </Price>
        </Item>
        <item>
            <ItemID> 00121 </ItemID>
            <Item> Scalp Massager </Item>
            <Price> 5.95 </Price>
        </Item>
    </Items>
</ShoppingCart>
```

Benefits:

- Human-readable
- Human-editable
- Interleaves structure with text (data)

Problems:

- Verbose: transmit more data than needed
- Longer parsing time
- Data conversion always required for numbers

JSON: JavaScript Object Notation

Formato ligero (relativamente eficiente) para el intercambio de datos

– Presentado como la alternativa más sencilla a XML

– Basado en JavaScript

• Escribible y legible por humanos

• Autodescriptivo (tipado explícitamente)

• Independiente del lenguaje

• Fácil de analizar

• Actualmente conversores para más de 50 idiomas

• Incluye compatibilidad con la invocación de RPC mediante JSON-RPC

Google Protocol Buffers

Mecanismo eficiente para serializar datos estructurados

– Mucho más simple, compacto y rápido que XML

• Independiente del lenguaje

• Define mensajes

– Cada mensaje es un conjunto de nombres y tipos

• Compila los mensajes para generar clases de acceso a datos para tu lenguaje

• Ampliamente utilizado en Google. Actualmente se definen más de 48.000 tipos de mensajes diferentes.

– Se utiliza tanto para RPC como para almacenamiento persistente.

Pruebas de eficiencia

```
<person>
    <name>John Doe</name>
    <email>jdoe@example.com</email>
</person>
```

XML version

```
person {
   name: "John Doe"
   email: "jdoe@example.com"
}
```

Text (uncompiled) protocol buffer

Mensaje codificado en binario: ~28 bytes de longitud, 100-200 ns de análisis

• Versión XML: ≥69 bytes, 5000-10 000 ns de análisis

• En general:

– Datos de 3 a 10 veces más pequeños

– Marshal/desmarshal de 20 a 100 veces más rápido

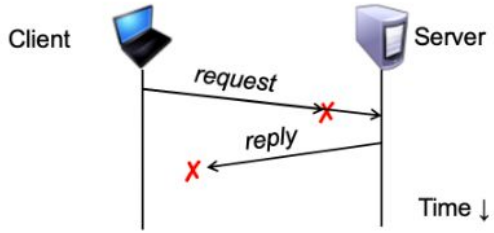
– Más fácil de usar programáticamente

Qué cosas pueden fallar?

– El cliente podría bloquearse y reiniciarse.

– Se podrían perder paquetes.

– Se podrían perder algunos paquetes individua



The cause of the failure is hidden from the client!

– Un enrutamiento interrumpido provoca la pérdida de muchos paquetes.

– El servidor podría bloquearse y reiniciarse.

– La red o el servidor podrían funcionar muy lentos.

gRPC	REST
HTTP/2	HTTP/1.1
gRPC utiliza Protocol Buffer de forma predeterminada para serializar los datos de la carga útil.	REST se basa principalmente en formatos JSON o XML para enviar y recibir datos.
Protocol Buffers es binario, de menor tamaño y con un procesamiento menos intensivo de la CPU.	JSON es texto sin formato, de mayor tamaño y con un procesamiento más intensivo de la CPU.
Compatibilidad bidireccional, asíncrona y de transmisión.	Solo compatible con solicitud/respuesta cliente-servidor.
Generación de código nativo compatible con muchos lenguajes a través de Protocol Buffer.	Generación de código posible a través de OpenAPI usando herramientas Swagger, lo cual es un paso adicional.
Basado en RPC, gRPC hace la instalación por nosotros.	Estructura HTTP, tenemos que usar una biblioteca de terceros

https://www.ime.usp.br/~reverbel/SMW-07/Slides/alonso-ch2-RPC.pdf

https://courses.cs.duke.edu/spring15/cps210/slides/rpc.pdf

https://people.cs.rutgers.edu/~pxk/417/notes/content/03-rpc-slides.pdf

https://changlousys.github.io/CS4740/spring24/slides/Course5-RPC.pdf

https://users.cs.duke.edu/~mlentz/papers/mrpc_nsdi2023_slides.pdf

https://www.slideshare.net/slideshow/introduction-to-grpc-application-presentation/266033766