

LLAMADAS A PROCEDIMIENTOS REMOTOS (RCP)

UABCS - DASC -IDS
SISTEMAS DISTRIBUIDOS

Dirigido a:

M.T.I. Nelson Israel Higuera Castillo



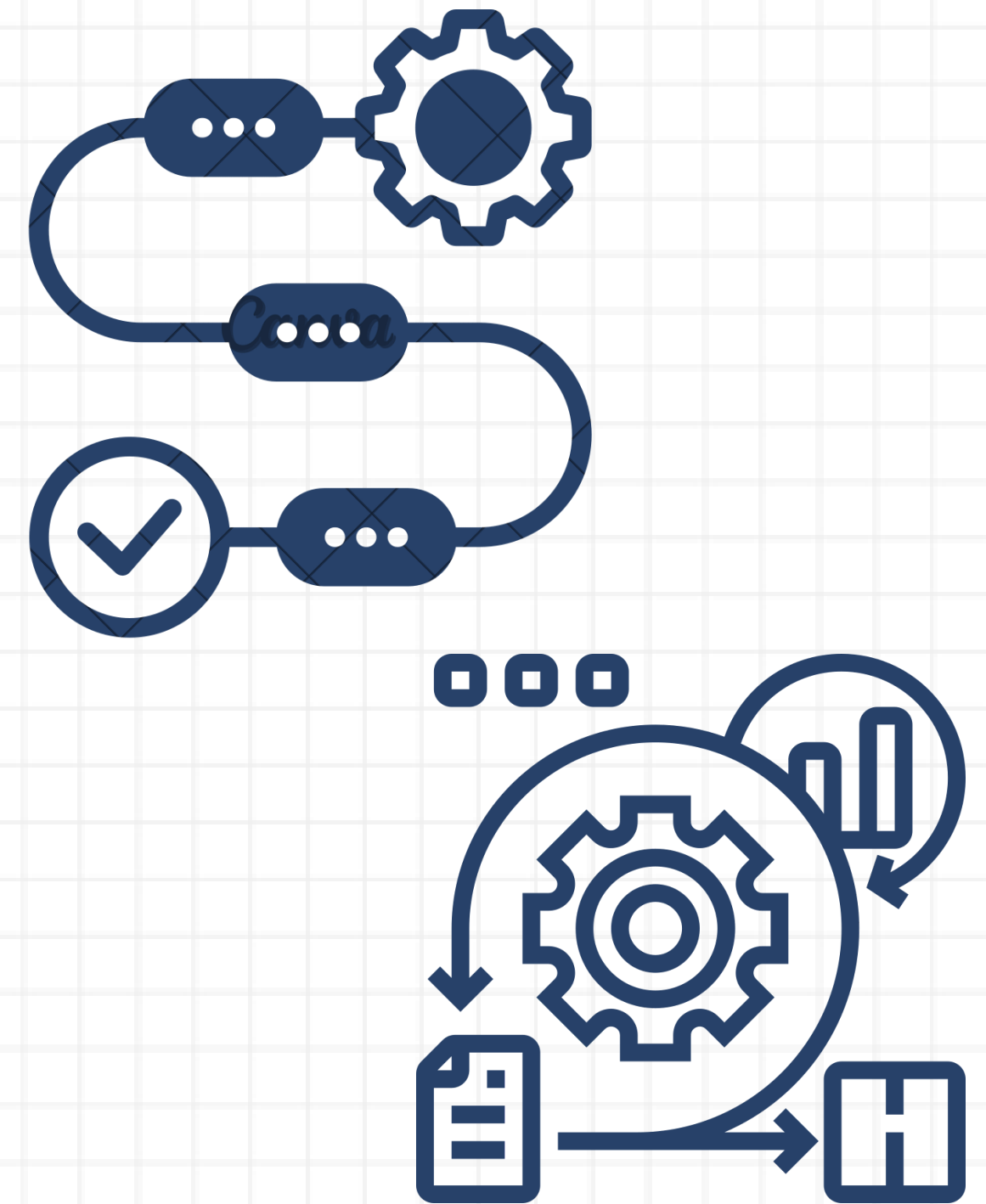
DEFINICIÓN GENERAL

Son las llamadas a procedimientos remotos (RPC), un componente de aplicación puede enviar de manera efectiva una petición a otro componente de aplicación, si realiza una llamada a un procedimiento local, lo cual resulta en una petición que se empaca como un mensaje y se envía al componente invocado (remoto).

De igual manera, el resultado se enviará de regreso y será devuelto a la aplicación como resultado de la llamada al procedimiento.

OPERACIÓN BÁSICA RPC Y FUNCIONAMIENTO

La RPC (Remote Procedure Call) es una operación básica en sistemas distribuidos que permite a un proceso en un sistema enviar una solicitud a un proceso en otro sistema, como si estuvieran en la misma máquina. Esto permite a los procesos trabajar juntos en diferentes sistemas y compartir recursos.



● COMPONENTES DE UN RCP

✓ CAMBIAMOS LA FUNCIÓN ORIGINAL EN LOCAL POR UN *STUB*

✓ LOS PARÁMETROS DEBERÁN SER SERIALIZADOS EN UN PROCESO DE *MARSHALLING*

✓ CLIENTE DE RED

✓ LOS PARÁMETROS SERÁN DES SERIALIZADOS EN UN PROCESO *UNMARSHALLING*

● COMPONENTES DE UN RCP



EL SKELETON CAPTURARA EL RETORNO QUE DEVUELVE



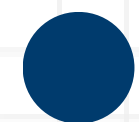
EL RETORNO DEBE SER SERIALIZADO



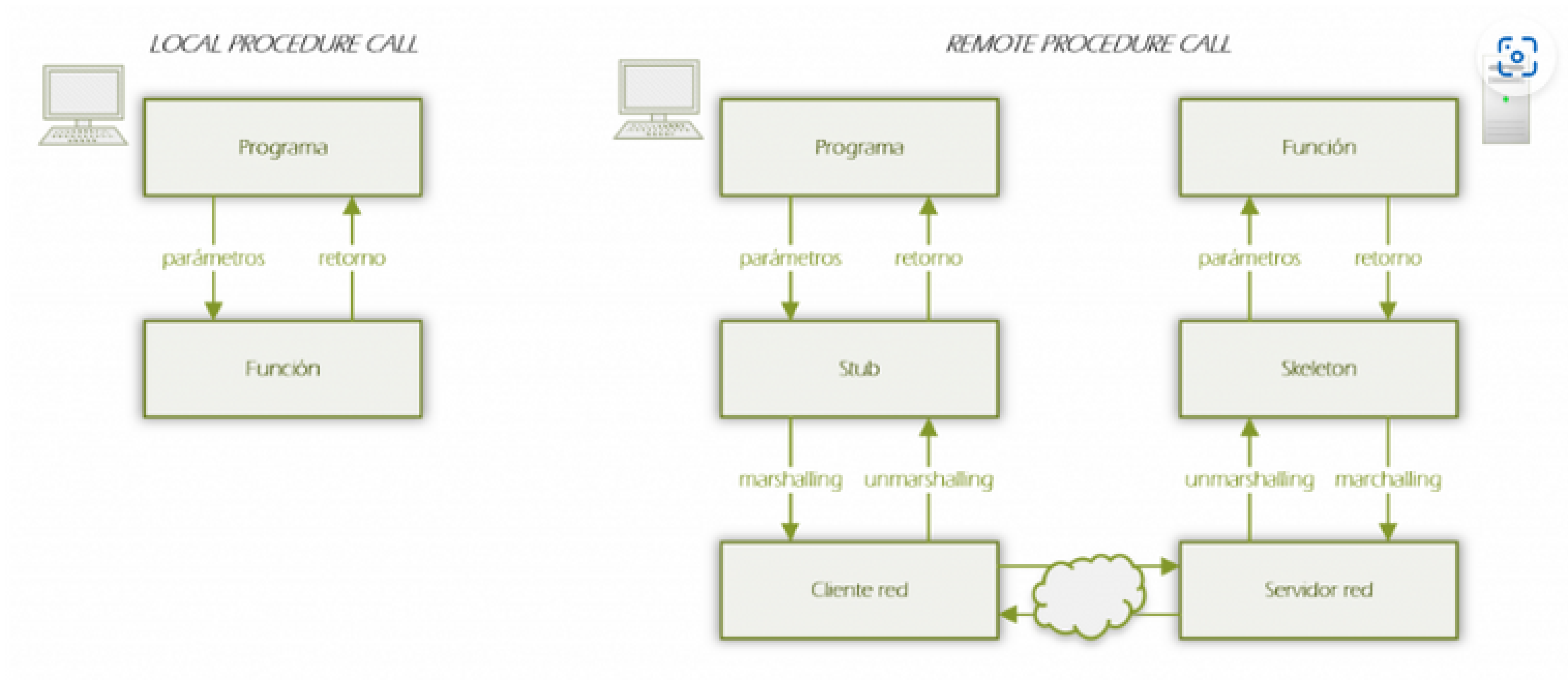
VALOR DE RETORNO ES PASADO AL SERVIDOR DE RED PARA QUE LO ENVÍE AL
CLIENTE



EL STUB DEVUELVE AL PROGRAMA QUE RETORNO QUE HA RECIBIDO



COMPONENTES



PARA RESUMIR, UNA LLAMADA A UN PROCEDIMIENTO REMOTO OCURRE EN LOS SIGUIENTES PASOS:

1

El procedimiento cliente llama al resguardo del cliente de manera normal.

2

El resguardo del cliente construye un mensaje y llama al sistema operativo local.

3

.El sistema operativo del cliente envía el mensaje al sistema operativo remoto

4

El sistema operativo remoto da el mensaje al resguardo del servidor.

5

El resguardo del servidor desempaca los parámetros y llama al servidor.

6

El servidor realiza el trabajo y devuelve el resultado al resguardo.

7

El resguardo del servidor empaca el resultado en un mensaje y llama a su sistema operativo local.

8

El sistema operativo del servidor envía el mensaje al sistema operativo del cliente

9

El sistema operativo del cliente da el mensaje al resguardo del cliente.

10

El resguardo desempaca el resultado y lo regresa al cliente.

● STUB, SKELETON, MARSHALLING Y UNMARSHALLING

● En sistemas distribuidos, los procesos se comunican a través de la red para realizar una tarea. Para que la comunicación sea posible, es necesario que los datos se transfieran entre diferentes lenguajes de programación y plataformas de hardware. Para este propósito, se utilizan técnicas como stub, skeleton, marshalling y unmarshalling.

● TÉCNICAS

1

STUB

(también conocido como cliente stub) es una versión local de un objeto remoto en el proceso cliente. El cliente llama a un método en el stub, que se encarga de enviar la solicitud al servidor remoto.

2

SKELETON

(también conocido como servidor stub) es una versión local del objeto remoto en el proceso servidor. El servidor recibe la solicitud del cliente a través del skeleton y llama al objeto remoto real para realizar la tarea solicitada.

● TÉCNICAS

3


MARSHALLING

(también conocido como serialización) es el proceso de convertir un objeto o estructura de datos en un formato que pueda ser transmitido a través de la red. El marshalling es necesario porque los diferentes lenguajes de programación y plataformas de hardware pueden representar los datos de manera diferente.

4

UNMARSHALLING

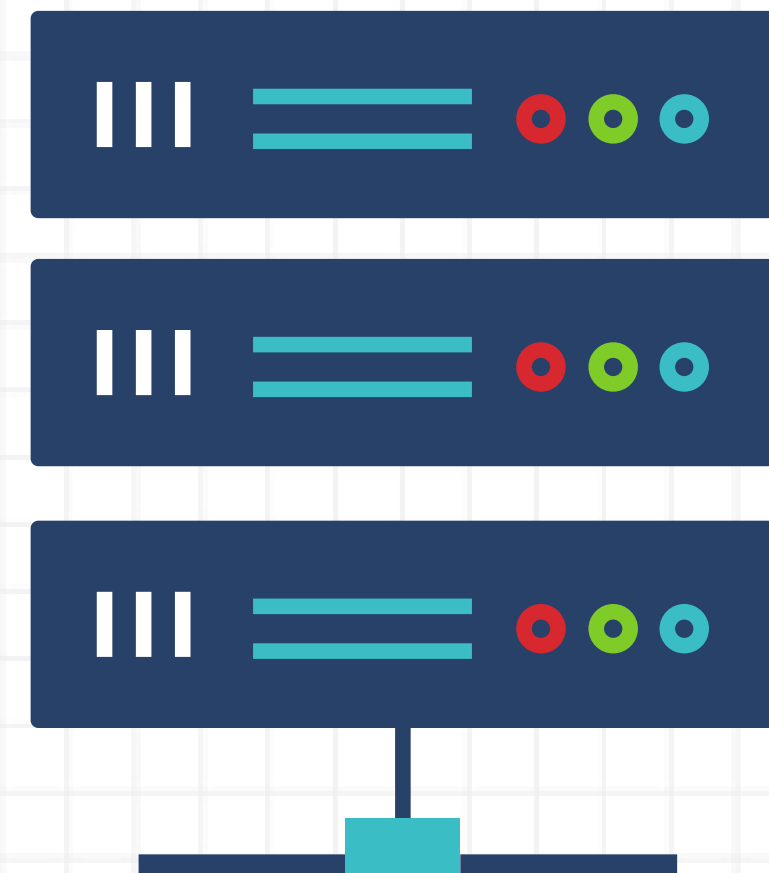
Es el proceso de convertir los datos transmitidos a través de la red en un formato que pueda ser entendido por el proceso receptor. Es el proceso inverso del marshalling.



RPC ASÍNCRONA

La **RCP asincrónica** (RPC, por sus siglas en inglés) es un patrón de comunicación entre procesos en sistemas distribuidos.

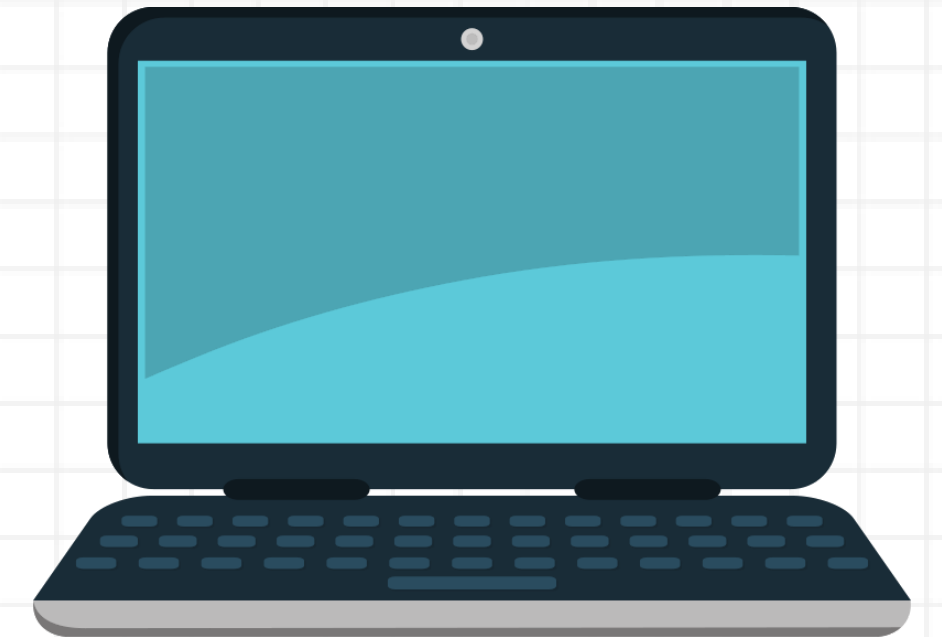
Una **RPC asincrónica** es una forma de hacer llamadas a procedimientos remotos sin bloquear el programa que las realiza, permitiendo que el cliente realice otras operaciones mientras espera la respuesta del servidor.



DCE RPC

Distributed Computing Environment Remote Procedure Call (DCE RPC) es un modelo de comunicación para sistemas distribuidos que fue desarrollado por la Open Software Foundation (OSF) en la década de 1980.

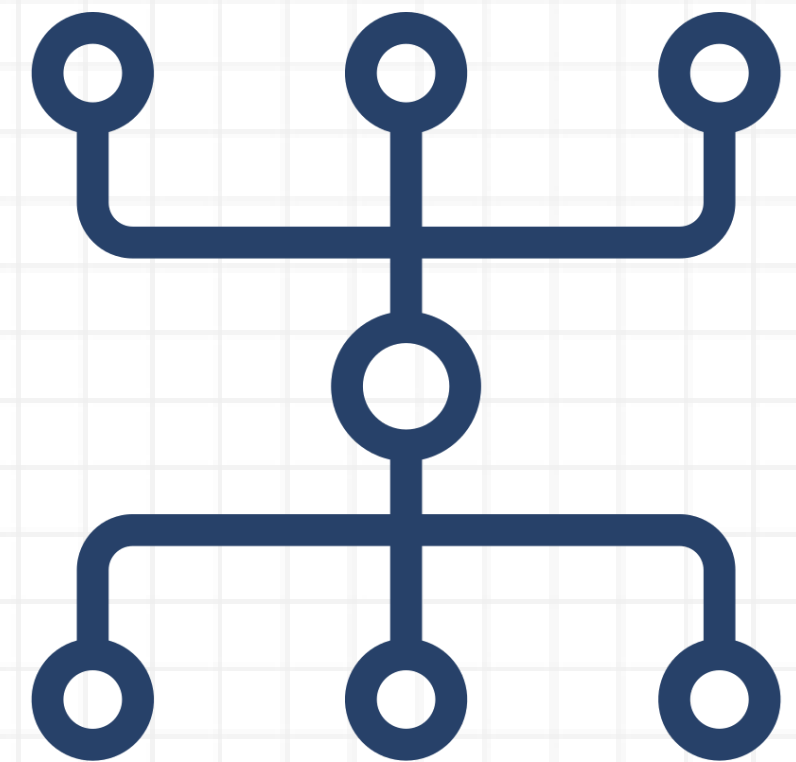
DCE RPC es un modelo de comunicación para sistemas distribuidos que utiliza stubs, skeletons, marshalling y unmarshalling para permitir que los procesos en diferentes máquinas se comuniquen entre sí. DCE RPC es utilizado por sistemas operativos para proporcionar servicios de red y ofrece una amplia gama de características adicionales que lo hacen adecuado para entornos empresariales.



PROTOCOLO ONC-RPC Y XML-RPC

ONC RPC y XML-RPC son dos protocolos utilizados en sistemas distribuidos para permitir la comunicación entre procesos en diferentes máquinas. **ONC RPC** utiliza el formato XDR para transmitir datos a través de la red y es ampliamente utilizado en sistemas Unix y Linux.

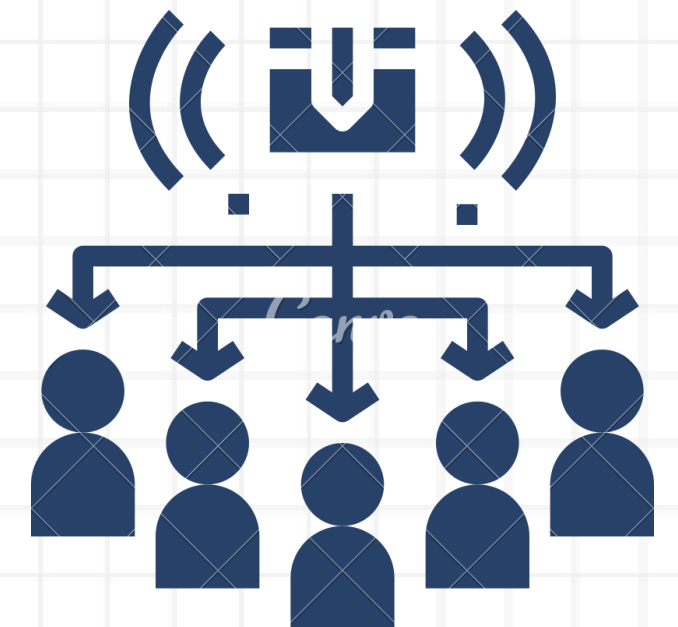
XML-RPC utiliza XML para codificar datos y puede comunicarse a través de Internet utilizando HTTP como protocolo de transporte. Ambos protocolos permiten la comunicación entre procesos de diferentes lenguajes y plataformas.



LIBRERÍAS DE RPC EN JAVASCRIPT:

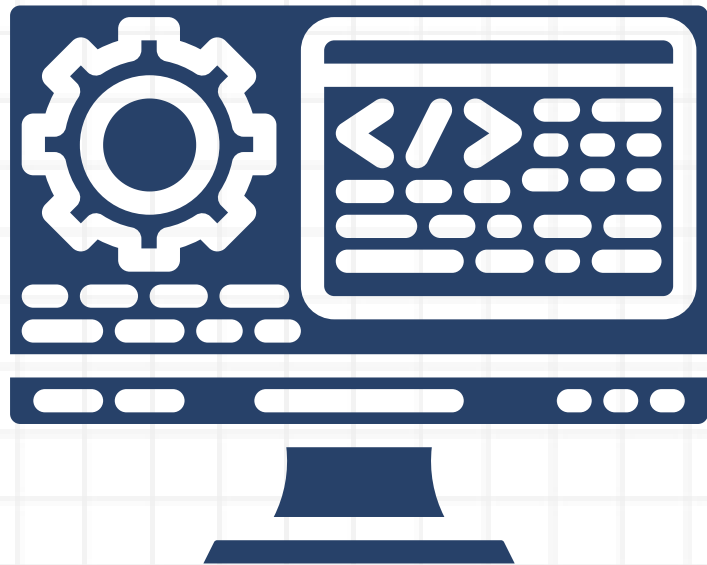
JSRPC

La librería `jsRPC` es una biblioteca de JavaScript para la implementación de llamadas a procedimientos remotos utilizando el protocolo de comunicación JSON-RPC. JSON-RPC es un protocolo simple para la comunicación de datos en sistemas distribuidos basados en el formato JSON.



IMPLEMENTACION JSRCP

Instalar mediante npm la libreria jsrpc de la siguiente manera:



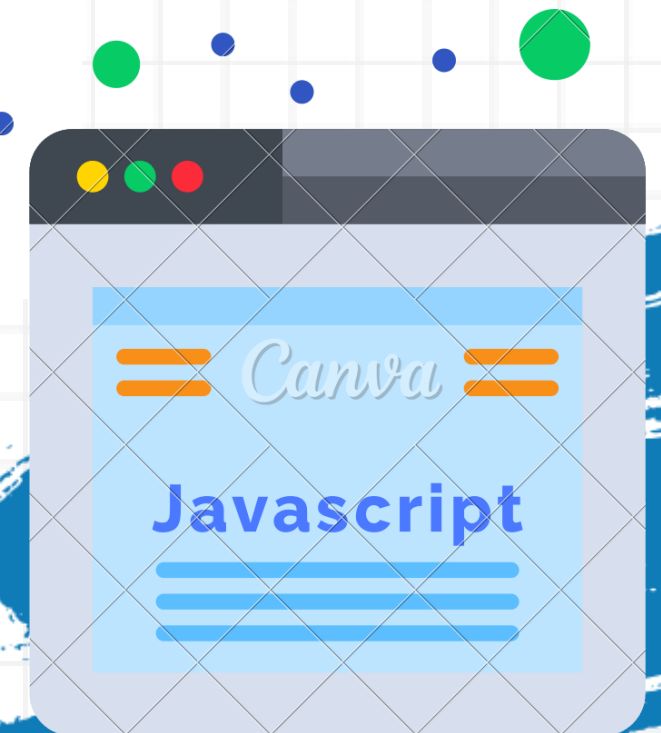
```
1 | npm install @todojs/jsrpc --save
```

IMPLEMENTACION

Vamos a utilizar un pequeño ejemplo, sin sentido práctico alguno, que nos va a servir para comprender cómo funciona el sistema. Es una librería de cálculo matemático muy simple:

arithmetic.js

```
1  module.exports = {  
2    addition      : async (x, y) => await x + y,  
3    subtraction   : async (x, y) => await x - y,  
4    multiplication : async (x, y) => await x * y,  
5    division       : async (x, y) => await x / y  
6  };
```

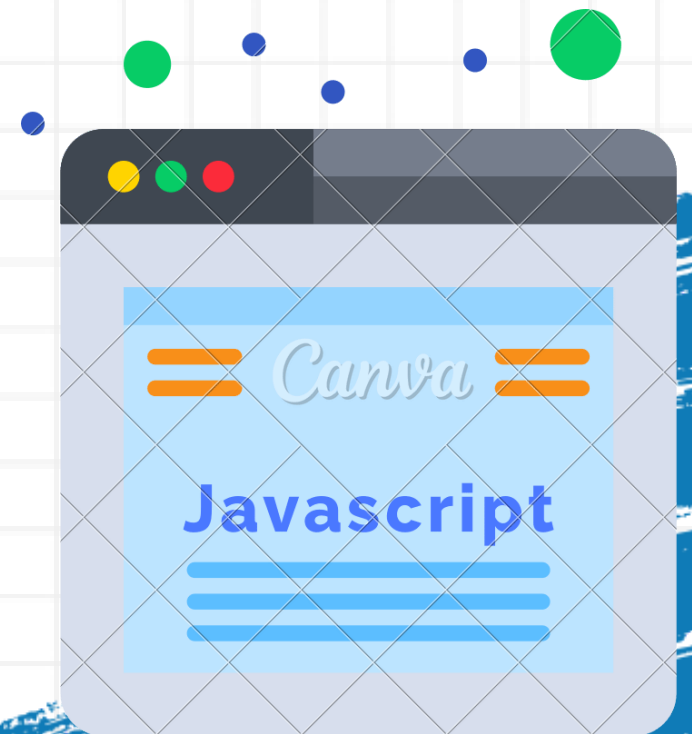


IMPLEMENTACION JS

Este sería un ejemplo de cómo se utilizaría esta librería en forma de llamada local:

example01.js

```
1  const Arithmetic = require ('./arithmetic');
2
3  (async () => {
4    console.assert (await Arithmetic.addition (2, 3) === 5);
5    console.assert (await Arithmetic.subtraction (2, 3) === -1);
6    console.assert (await Arithmetic.multiplication (2, 3) === 6);
7    console.assert (await Arithmetic.division (2, 3) === 2 / 3);
8  }) ();
```

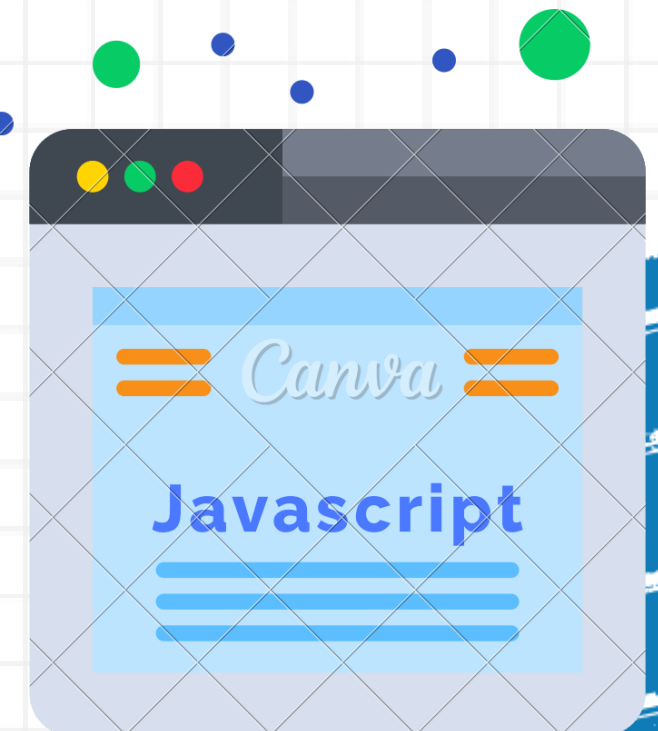


IMPLEMENTACION JS

Para crear el stub vamos a utilizar una función de apoyo que hemos llamado stubify()

arithmetic-stub.js

```
1  const stubify = require('@todojs/jsrpc/stubify');
2  module.exports = stubify (
3    "http://localhost:9000",
4    'arithmetic',
5    [
6      'addition',
7      'subtraction',
8      'multiplication',
9      'division'
10   ]
11 );
```

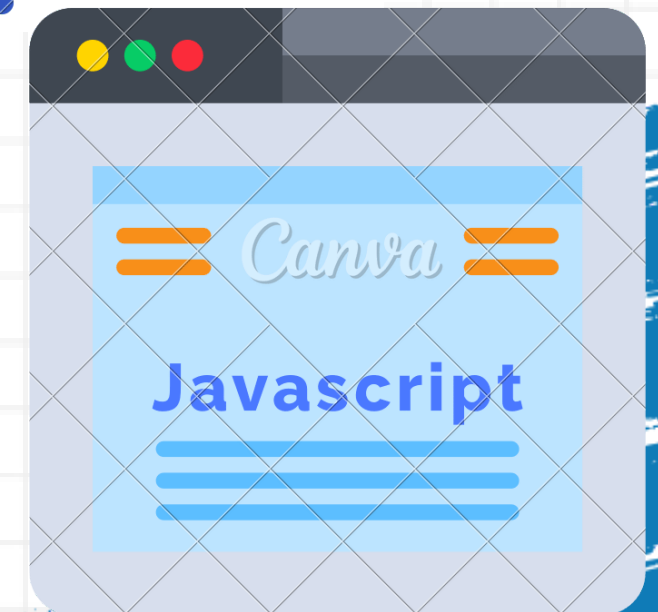


IMPLEMENTACION JS

Para crear el *skeleton* vamos a utilizar una función de apoyo que hemos llamado `skeletonify()`

`arithmetic-skeleton.js`

```
1 | const Arithmetic = require ('./arithmetic');  
2 | const skeletonify = require ('@todojs/jsrpc/skeletonify');  
3 | skeletonify ('arithmetic', Arithmetic);
```

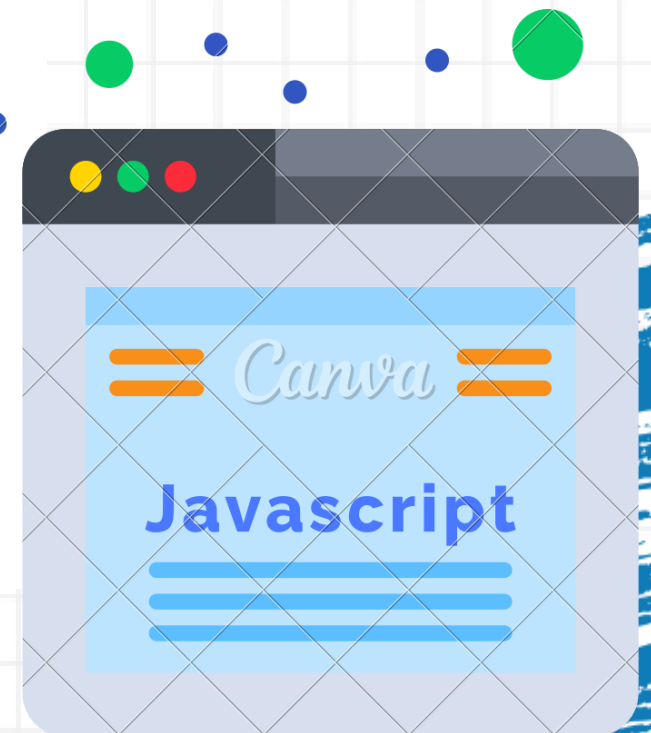


IMPLEMENTACION JS

Ahora, si ejecutamos el servidor con `node arithmetic-skeleton.js` y volvemos a ejecutar el ejemplo inicial, pero con el *stub* veremos que las funciones se ejecutan en el servidor, en vez de localmente.

example02.js

```
1  const Arithmetic = require ('./arithmetic-stub');
2
3  (async () => {
4    console.assert (await Arithmetic.addition (2, 3) === 5);
5    console.assert (await Arithmetic.subtraction (2, 3) === -1);
6    console.assert (await Arithmetic.multiplication (2, 3) === 6);
7    console.assert (await Arithmetic.division (2, 3) === 2 / 3);
8    try {
9      await Arithmetic.other (2, 3)
10   } catch(err) {
11     console.assert(err.message === 'Method not found')
12   }
13 }) ();
```



IMPLEMENTACION DE JSRCP

En esta implementación de RPC en Javascript hemos tomado algunas decisiones de diseño para utilizar una perspectiva idiomática y ajustarnos lo más posible a las características de este lenguaje:

✓ EL STUB NOS RECUERDA LAS FUNCIONALIDADES DE PROXY

✓ MARSHALLING Y UNMARSHALLING: JSON.STRINGIFY Y JSON.PARSER

✓ HTTP CON FETCH

✓ SKELETON COMO MANEJADOR DE UNA RUTA

IMPLEMENTACION DE JSRCP



GESTIONAR ASINCRONIA DE LA LLAMADA REMOTA POR UTILIZAR PROMESAS



NO ES NECESARIO UTILIZAR IDL, MANEJO DE OWKEYS Y
GETOWNPROPERTYDESCRIPTOR EN PROXY O
OBJECT.GETOWNPROPERTYDESCRIPTOR.

CÓDIGO DE JSRPC

Esta función stubify() devuelve un objeto stub con todas sus funciones preparadas para ser llamadas de forma remota.

stubify.js

```
1  const fetch = require ('node-fetch');
2
3  module.exports = function (server, objName, methods = []) {
4    let id      = 1;
5    const proxy = new Proxy ({}, {
6      ownKeys      : () => methods,
7      getOwnPropertyDescriptor : (target, prop) => ({
8        value : proxy[ prop ], writable : true, enumerable : true
9      }),
10     get      : (() => {
11       const cache = [];
12       return (target, method) => {
13         return cache[method] || (cache[method] = async (...params) => {
14           const res = await fetch (`${ server }/${ objName }/${ method }`, {
15             method : 'POST',
16             headers : { 'Content-Type' : 'application/json' },
17             body    : JSON.stringify ({
18               jsonrpc : "2.0",
19               method  : method,
20               params  : params,
21               id      : id++
22             }) // jsonRPC 2.0 standard format
23           });
24           const data = await res.json ();
25           if (res.status === 200) {
26             return data.result;
27           }
28           throw new Error (data.error.message);
29         });
30       });
31     }) ();
32   });
33   return proxy;
34 }
```

CÓDIGO DE JSRPC

Al mostrar este código estamos desvelando algunas características de nuestro modelo de comunicación, ya que vamos a llamar a un servidor concreto, con una URL concreta.

server.js

```
1  const app = require ('http').createServer ();
2
3  const routes = new Map ();
4  app.listen (process.env.PORT || 9000);
5  app.on ('request', (request, response) => {
6    const bodyChunks = [];
7    request.on ('data', (chunk) => bodyChunks.push (chunk));
8    request.on ('end', async () => {
9      try {
10        request.bodyRaw = Buffer.concat (bodyChunks);
11        request.body = request.bodyRaw.length ? JSON.parse (r
12      } catch (err) {
13        return sendError (400, -37600, "Parse error");
14      }
15      try {
16        for (let route of routes) {
17          if (request.url.match (route[ 0 ]) && route[ 1 ][ requ
18            await route[ 1 ][ request.method.toLowerCase () ] (r
19            if (response.finished) {
20              return log ();
21            }
22          }
23        }
24        return sendError (401, -32601, "Method not found", reque
25      } catch (err) {
26        if (err.message === "Cannot read property 'apply' of und
27          return sendError (401, -32601, "Method not found", req
28        } else {
29          return sendError (500, -32000, err.message, request.bo
30        }
31      }
32    });
33  });
```

CÓDIGO DE JSRPC

Este es un servicio http basado en el servidor estándar de Node al que hemos añadido un par de funcionalidades interesantes como son la captura completa del body con versión del mismo en un objeto por medio de `JSON.stringify()` y la posibilidad de definir múltiples rutas por medio de la función que se retorna al llamar a la función `addRoute`.

```
34 function sendError (status, code, message, id = null) {
35   response.statusCode = status;
36   response.setHeader ('Content-Type', 'application/json');
37   response.end (JSON.stringify ({
38     "jsonrpc" : "2.0",
39     "error"   : {code, message},
40     "id"      : id
41   }));
42   log ()
43 }
44
45 function log () {
46   console.log (new Date ().toISOString (), request.connection
47     response.statusCode, request.url, request.body && request
48   )
49 }
50 });
51
52 module.exports = function addRoute (path, methods) {
53   routes.set (path, methods);
54   return app;
55 };
```

CÓDIGO DE JSRPC

Esta función `skeletonify()` permite crear un *skeleton* de forma muy sencilla y crea la ruta necesaria en el servidor http. También llama a las funciones originales pasando los parámetros y capturando la respuesta para enviarla al cliente en formato jsonRPC.

`skeletonify.js`


```
1  const addRoute = require ('./server');
2
3  module.exports = function skeletonify (objName, obj) {
4    return addRoute (new RegExp (`^\\/${objName}\\/`), {
5      post : async (request, response) => {
6        response.setHeader ('Content-Type', 'application/json');
7        response.end (JSON.stringify ({
8          jsonrpc : "2.0",
9          result  : await obj[ request.body.method ].apply (null
10             id      : request.body.id
11         }));
12       }
13     });
14   };
```




CONCLUSIÓN

En general nos hemos preocupado en exceso en nuestras aplicaciones por las comunicaciones entre el cliente y el servidor, o entre servidores, cuando perfectamente podemos delegar esta labor en librerías de RPC bien construidas que nos permiten usar nuestras funciones remotas de forma transparente en nuestro código.

La flexibilidad que ofrecen estos modelos es muy interesante y nos permite mantener un código centrado en la funcionalidad y no en la distribución de los elementos entre el cliente y el servidor.



BIBLIOGRAFIA

- Andrew S. Tanenbaum, Maarten van Oteer. Distributed Systems: Principles and Paradigms. México. Editorial Prentice Hall. (2007).
- Recuperado de:<https://www.todojs.com/uso-de-jsrpc-en-una-aplicacion-de-ejemplo/>
- Recuperado de:http://dccd.cua.uam.mx/libros/archivos/03IXStream_sistemas_distribuidos.pdf

INTEGRANTES DEL EQUIPO

Angel de Jesús
Aviles Cota

Suzett Joanna
León Victorio

Jesús Emmanuel
Carballo
Caballero