分布式系统

RPC引言

林坤辉

本地过程调用示例

- 假设要调用函数add来计算i+j的结果:

```
1 int add(int x, int y) {
2     int z = x + y;
3     return z;}
4
5 int i = 10;
6 int j = 20;
7 int i_add_j = add(i, j);
```

- 那么在第7行时，实际上执行了以下操作：
- 将 i 和 j 的值压栈
- 进入add函数，取出栈中的值10 和 20，赋予 x 和 y
- 执行第2行代码，计算 x + y ，并将结果存在 z
- 将 z 的值压栈，然后从add返回
- 第7行，从栈中取出返回值 30 ，并赋值给 i_add_j

远程过程调用（RPC--Remote Procedure Call ）

- 远程过程调用（RPC--Remote Procedure Call ），不同进程之间的过程的调用。
- RPC的目的是可以像本地程序调用一样使用远程服务。RPC 使用的是客户机/服务器模式。

  服务程序就是一个服务器(server)，server提供一个或多个远程过程；请求程序就是一个客户机(client)，client向server发出远程调用。

1. **程序调用：**如何使远程的过程调用看起来和本地的过程调用没有区别的问题

2. **网络通信：**调用进程和被调用进程间的网络比本地计算机有更复杂的特性。例如， 它可能限制消息尺寸，并且有丢失和重排消息的可能，安全问题；

3. **操作系统：**运行调用和被调用进程的计算机可能有明显不同的体系结构和数据表示格式；

4. **编程语言：**跨语言之间的互操作问题；

远程过程调用带来的新问题

- 在远程调用时，需要执行的函数体是在远程的机器上的，即，add是在另一个进程中执行的。这就带来了几个新问题：
- 1. 怎么告诉远程机器要调用的是函数add，而不是Multip或其他的函数？
- 2. 客户端怎么把参数值传给远程的函数呢？
- 3.远程调用往往用在网络上，客户端和服务端是通过网络连接的，数据如何通过网络传输？

远程过程调用问题的解决

- **1. Call ID映射。**在本地调用中，函数体是直接通过函数指针来指定的，但是在远程调用中，函数指针是不行的，因为两个进程的地址空间是完全不一样的。所以，在RPC中，所有的函数都必须有自己的一个ID。这个ID在所有进程中都是唯一确定的。客户端在做远程过程调用时，必须附上这个ID。然后我们还需要在客户端和服务端分别维护一个｛函数<--> Call ID｝的对应表。两者的表不一定需要完全相同，但相同的函数对应的Call ID必须相同。当客户端需要进行远程调用时，它就查一下这个表，找出相应的Call ID，然后把它传给服务端，服务端也通过查表，来确定客户端需要调用的函数，然后执行相应函数的代码。

远程过程调用问题的解决

- **2. 序列化和反序列化。**客户端怎么把参数值传给远程的函数呢？在本地调用中，我们只需要把参数压到栈里，然后让函数自己去栈里读就行。但是在远程过程调用时，客户端跟服务端是不同的进程，不能通过内存来传递参数。甚至有时候客户端和服务端使用的都不是同一种语言（比如服务端用C++，客户端用Java或者Python）。这时候就需要客户端把参数先转成一个字节流，传给服务端后，再把字节流转成自己能读取的格式。这个过程叫序列化和反序列化。同理，从服务端返回的值也需要序列化反序列化的过程。

远程过程调用问题的解决

- **3. 网络传输。**远程调用往往用在网络上，客户端和服务端是通过网络连接的。所有的数据都需要通过网络传输，因此就需要有一个网络传输层。网络传输层需要把Call ID和序列化后的参数字节流传给服务端，然后再把序列化后的调用结果传回客户端。只要能完成这两者的，都可以作为传输层使用。因此，它所使用的协议其实是不限的，能完成传输就行。尽管大部分RPC框架都使用TCP协议，但其实UDP也可以，而gRPC干脆就用了HTTP2。Java的Netty也属于这层的东西。

实现RPC的机制

- *// Client端*
- *//　　int i_add_j = Call(ServerAddr, add, i, j)*

- 1. 将这个调用映射为Call ID。这里假设用最简单的字符串当Call ID的方法
- 2. 将Call ID，i和j序列化。可以直接将它们的值以二进制形式打包
- 3. 把2中得到的数据包发送给ServerAddr，这需要使用网络传输层
- 4. 等待服务器返回结果
- 5. 如果服务器调用成功，那么就将结果反序列化，并赋给i_add_j

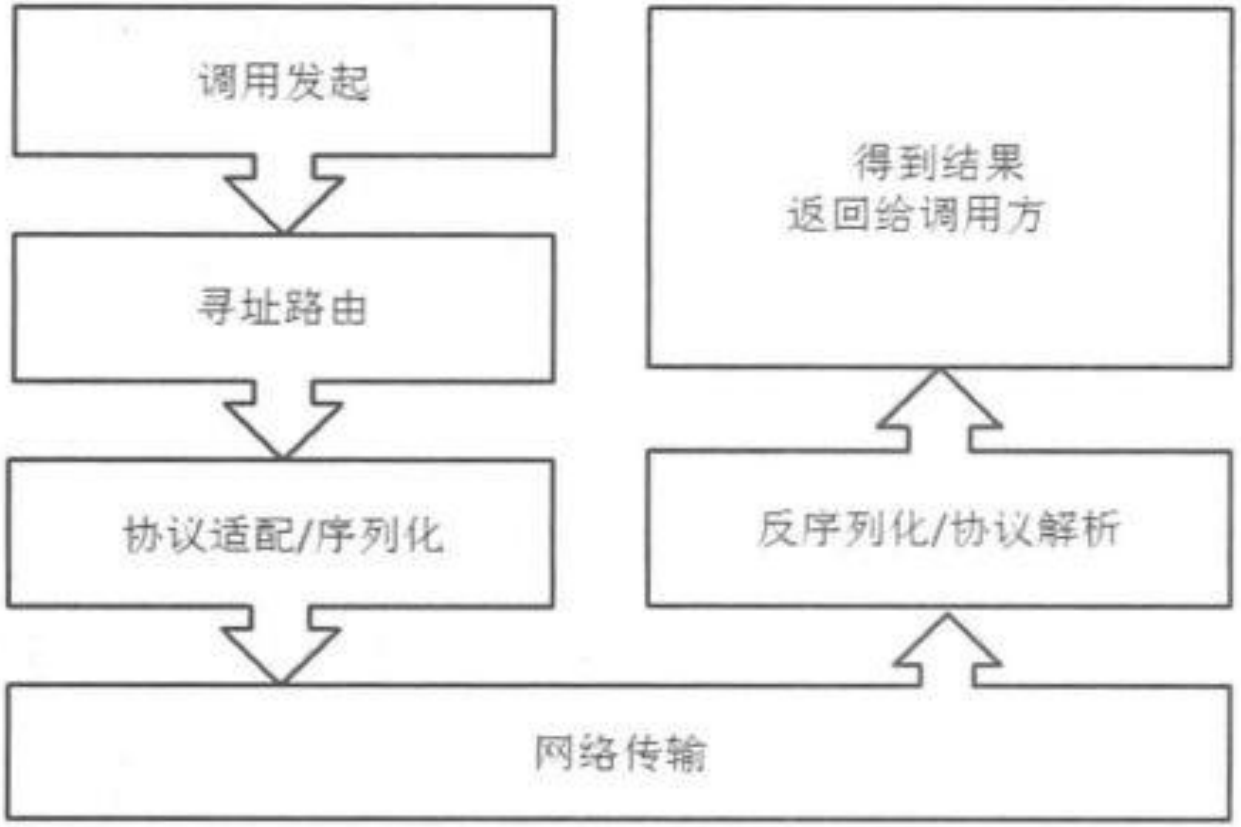
实现RPC的机制

- *// Server端*
- 1. 在服务端维护一个Call ID到函数指针的映射call_id_map，可以用std::map<std::string, std::function<>>
- 2. 等待请求
- 3. 得到一个请求后，将其数据包反序列化，得到Call ID
- 4. 通过在call_id_map中查找，得到相应的函数指针
- 5. 将i和j反序列化后，在本地调用add函数，得到结果
- 6. 将结果序列化后通过网络返回给Client

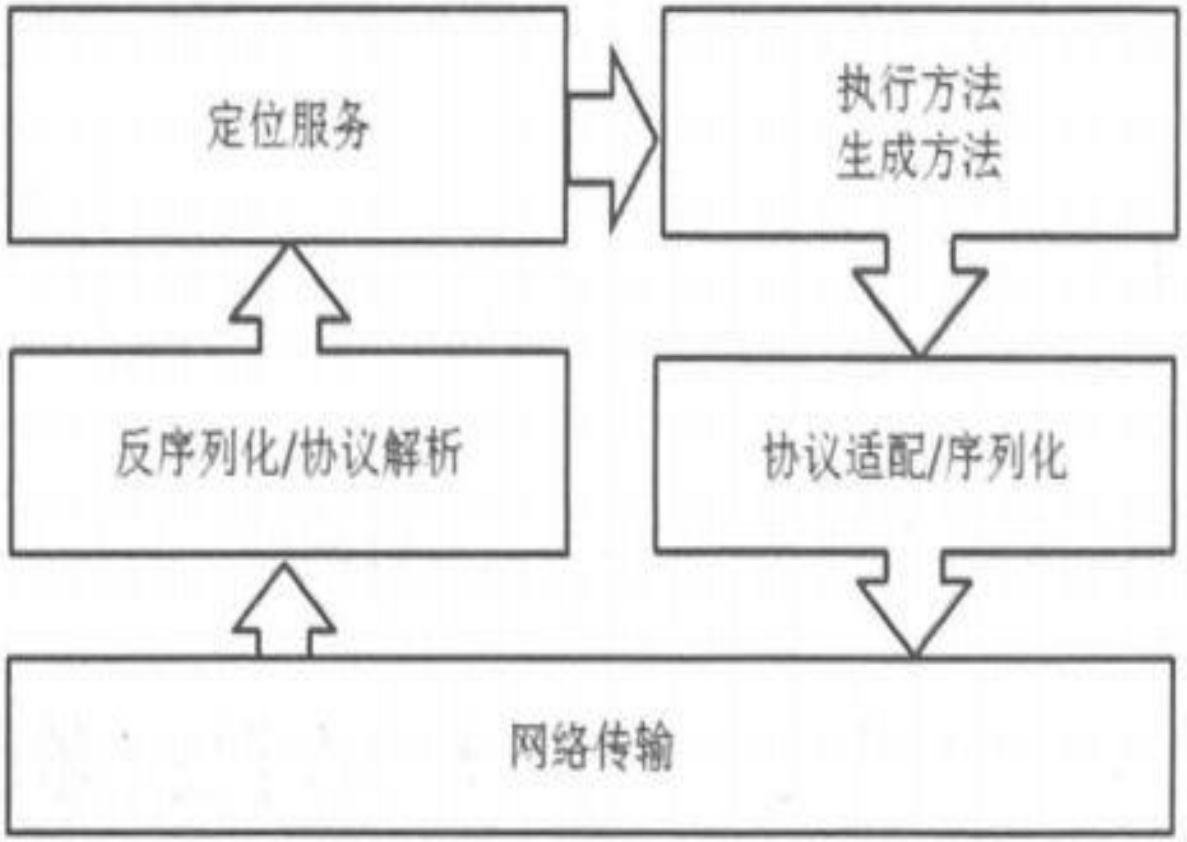
实现一个RPC框架

- 1. Call ID映射可以直接使用函数字符串，也可以使用整数ID。映射表一般就是一个哈希表。
- 2. 序列化反序列化可以自己写，也可以使用XML或者JSON之类的。
- 3. 网络传输库可以自己写socket，或用其他相关类。

服务调用端（本地机器）



服务提供端（远程机器）：



RPC：远程调用流程

通过RPC框架，使得我们可以像调用本地过程一样地调用远程机器上的过程：

- 1、本地调用某个函数或过程
- 2、本地机器的RPC框架把这个调用信息封装起来（调用的函数、入参等），序列化(json、xml等)后，通过网络传输发送给远程服务器
- 3、远程服务器收到调用请求后，远程机器的RPC框架反序列化获得调用信息，并根据调用信息定位到实际要执行的过程，执行完这个过程后，序列化执行结果，通过网络传输把执行结果发送回本地机器
- 4、本地机器的RPC框架反序列化出执行结果，并返回结果

Stub

- RPC引入了存根（Stub）的概念，比如服务端有某个函数fn()，它为了能够被远程调用，需要通过编译器生成两个stub：

  客户端的一个stub：c_fn()

  服务器端的一个stub：s_fn()

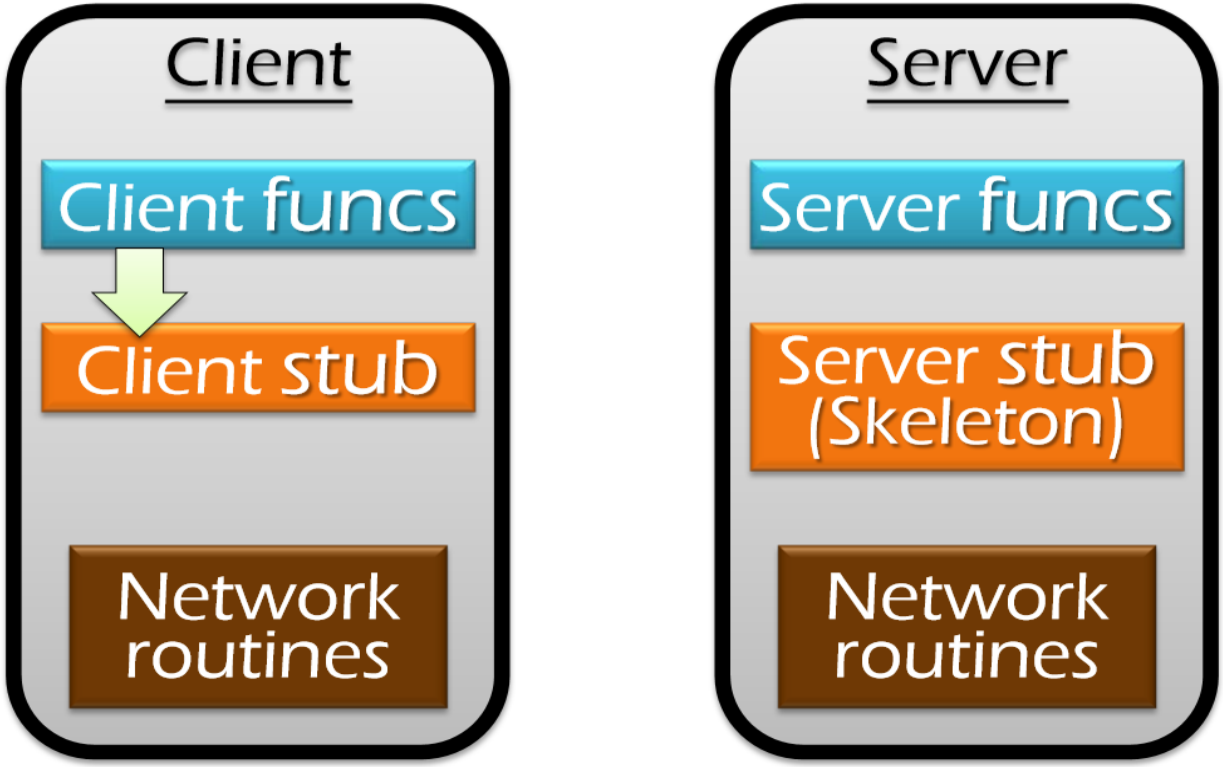
  在客户端，一个进程在执行过程中调用到了函数 fn()，此函数的具体实现是在远程的某台机器上，那么此进程实际上是调用了位于当地机器上的另外一个版本的fn()（即c_fn()），当客户端的消息发送到服务器端时，服务器端也不是把消息直接就交给真正的fn(),而是同样先交给一个不同版本的fn()（即s_fn()）

- **Stub的主要功能:**是对要发送的参数进行 marshal(可理解成一种打包操作)和对接受 到的参数（或返回值）进行unmarshal(解包)。Marshal操作将要发送的数据制成一种标准的格式（在DCE RPC系统中，此格式称做Network Data Representation（NDR）格式）unmarshal再从NDR格式数据包中读出所需数据。
- **Client stub的功能：**收集调用远程函数需要的参数，将这些参数marshal成消息，即把消息转化成标准的网络数据表示(network data representation , NDR) 格式，用于在网络上传递，调用客户端的运行时系统（Client runtime system）将此消息发送给服务器端。当服务器端将结果消息返回后，将结果消息unmarshal，把结果返回给应用进程。

- **Server stub的功能：**对发送给它的参数消息unmarshal，收集参数调用位于本机上的过程，将此过程执行的结果marshal成消息，然后调用服务器端的运行时系统将结果消息发送给客户端。
- **IDL compiler的功能:**就是对编辑好的IDL 文件进行编译，编译后生成了下面的三个文件：Header是一个头文件，此头文件包含了此IDL文件中的全局唯一标示符，数据类型定义，有关的常量定义，以及函数原型。客户代码和服务器代码中包含都要包含header文件，Client stub即客户端的stub程序。Server stub即服务器端的stub程序。

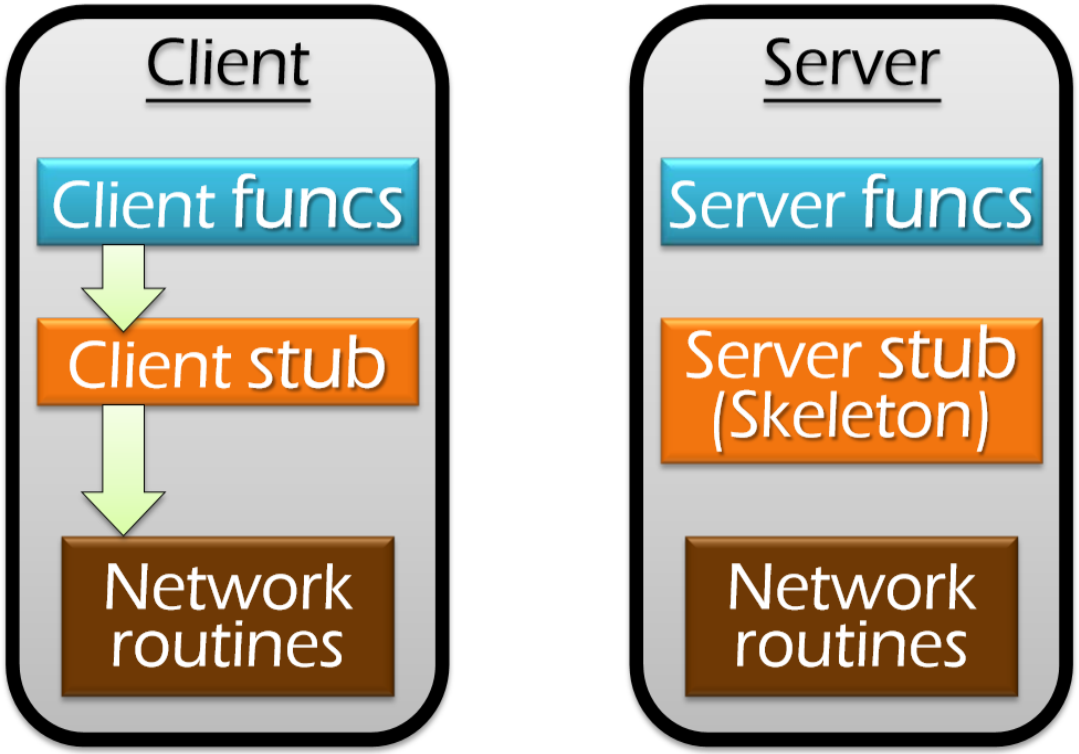
Stub Functions

1. Client calls **stub** (params on stack)



Stub Functions

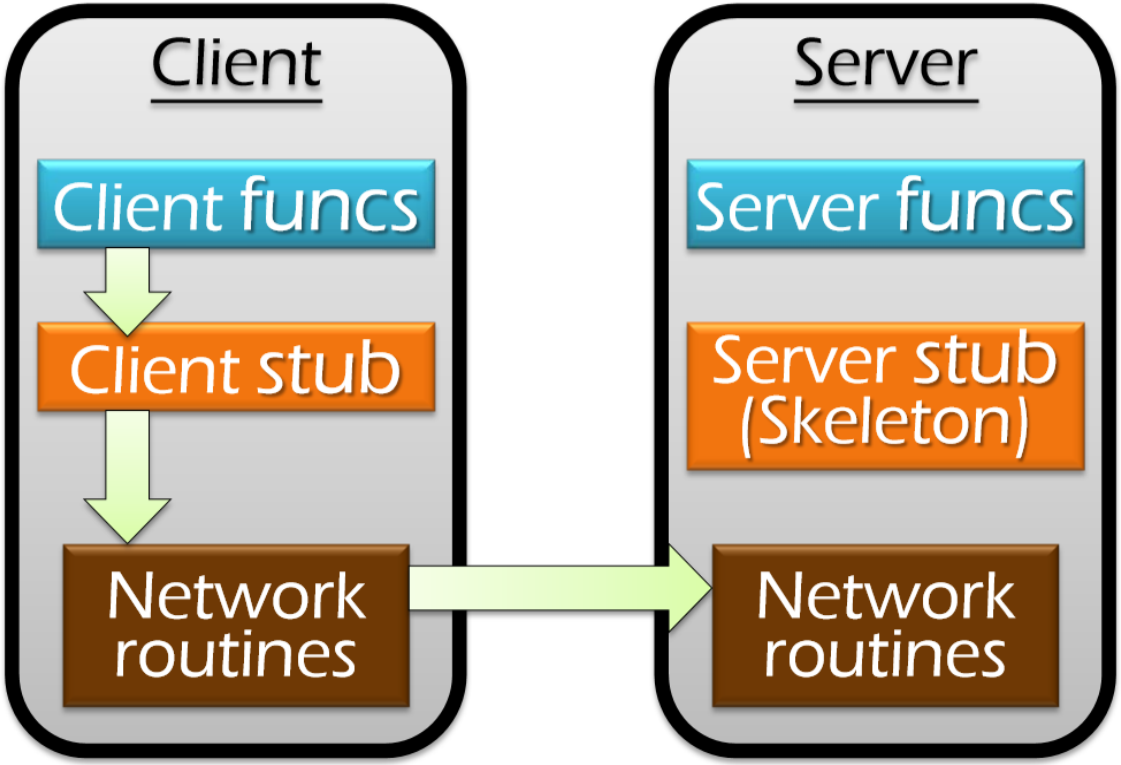
2. **Stub** marshals params to net **message**



- Marshals = put data in a form suitable for transmission over a network

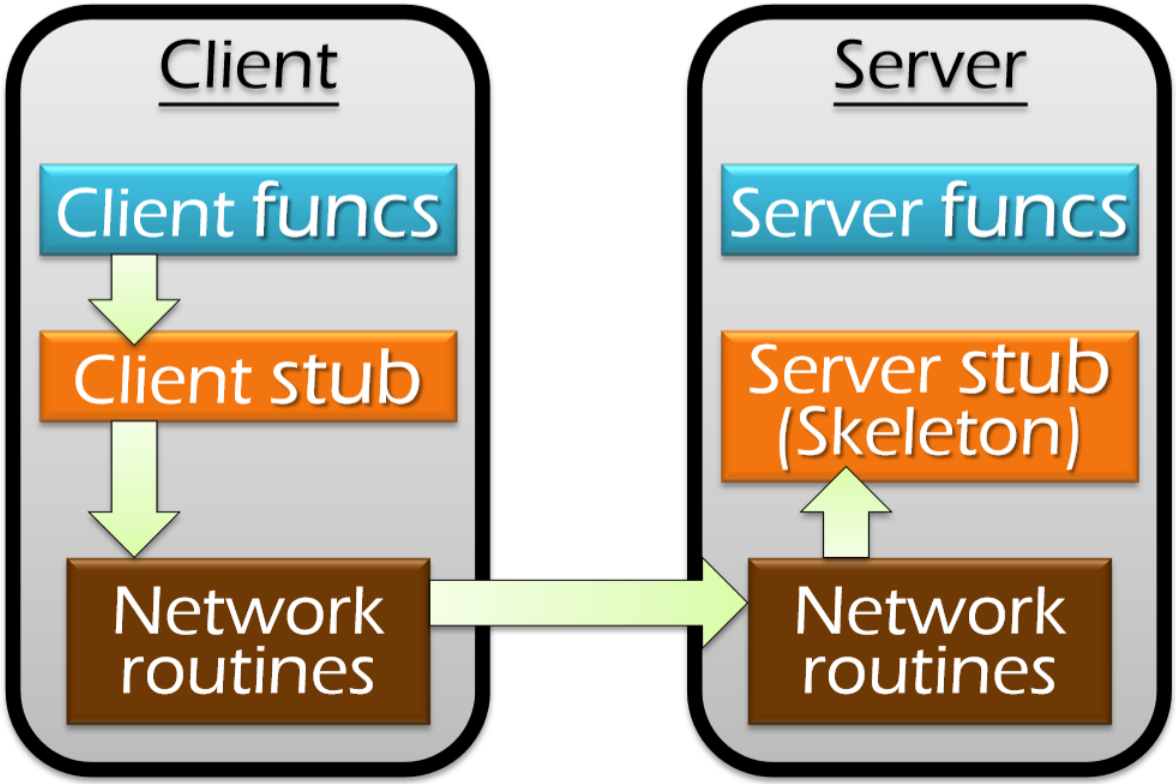
Stub Functions

3. Network routines sends **message** to server



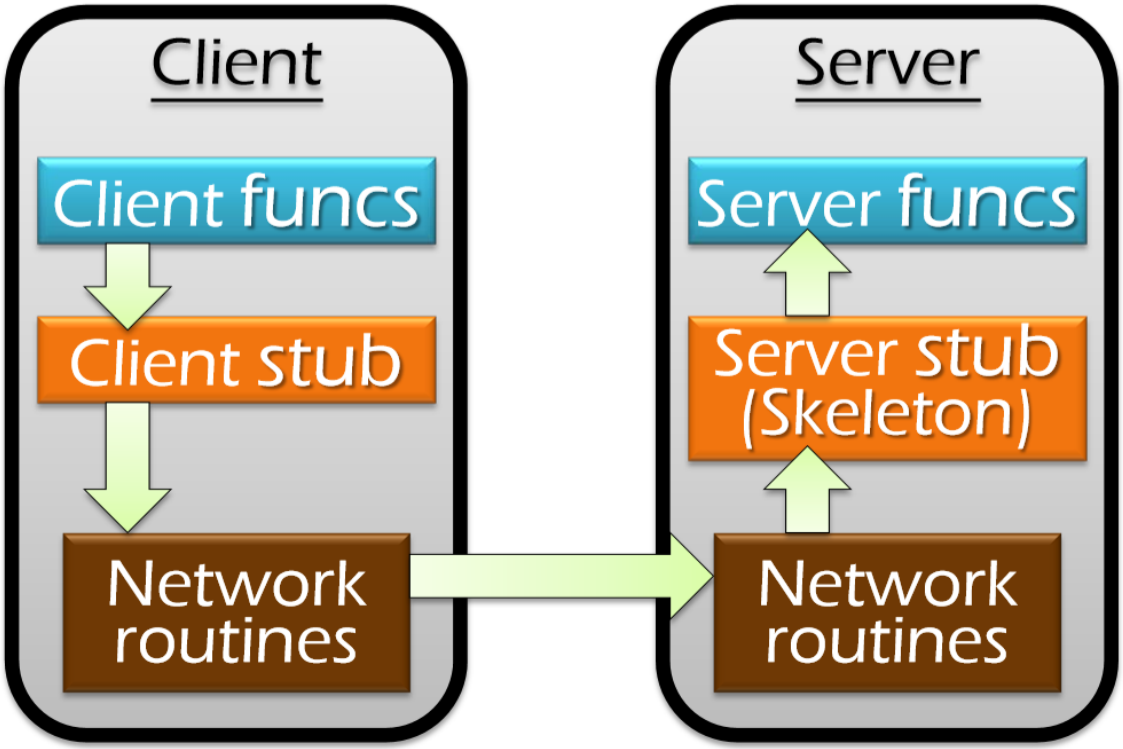
Stub Functions

4. Receive **message**: send it to server **stub**



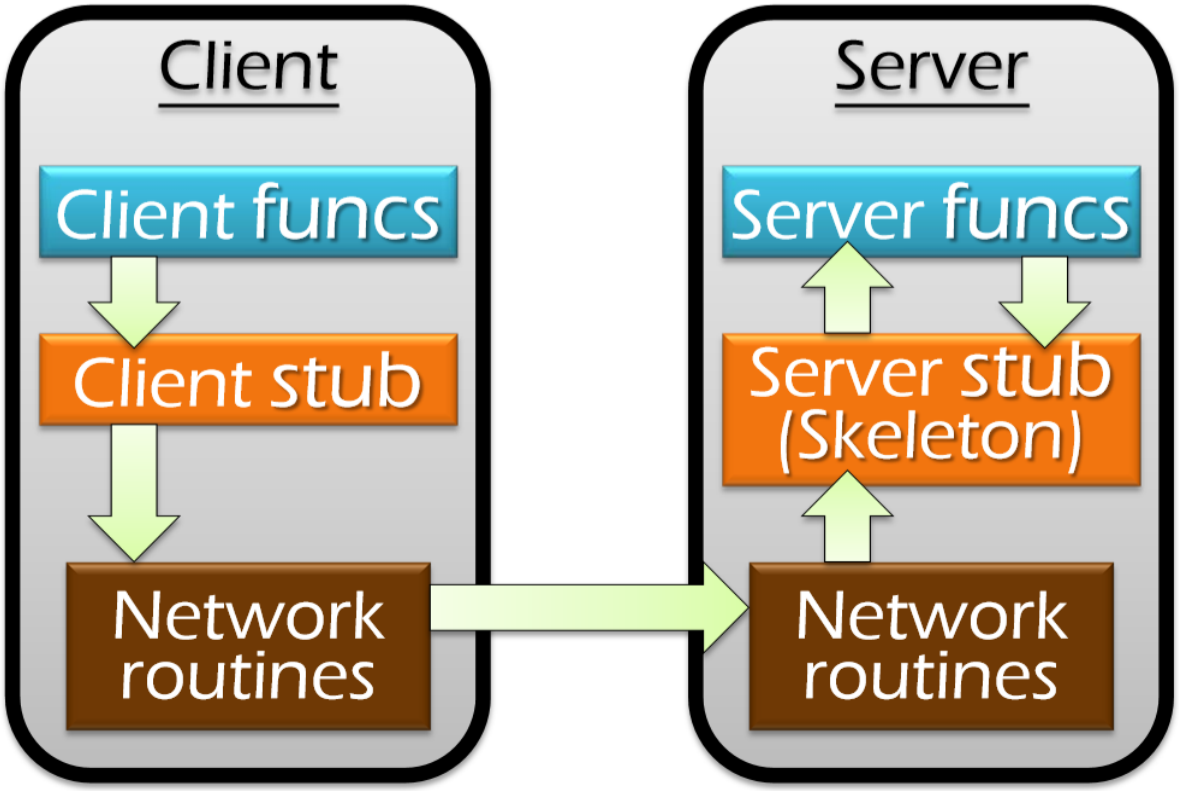
Stub Functions

5. Unmarshal parameters, call server function



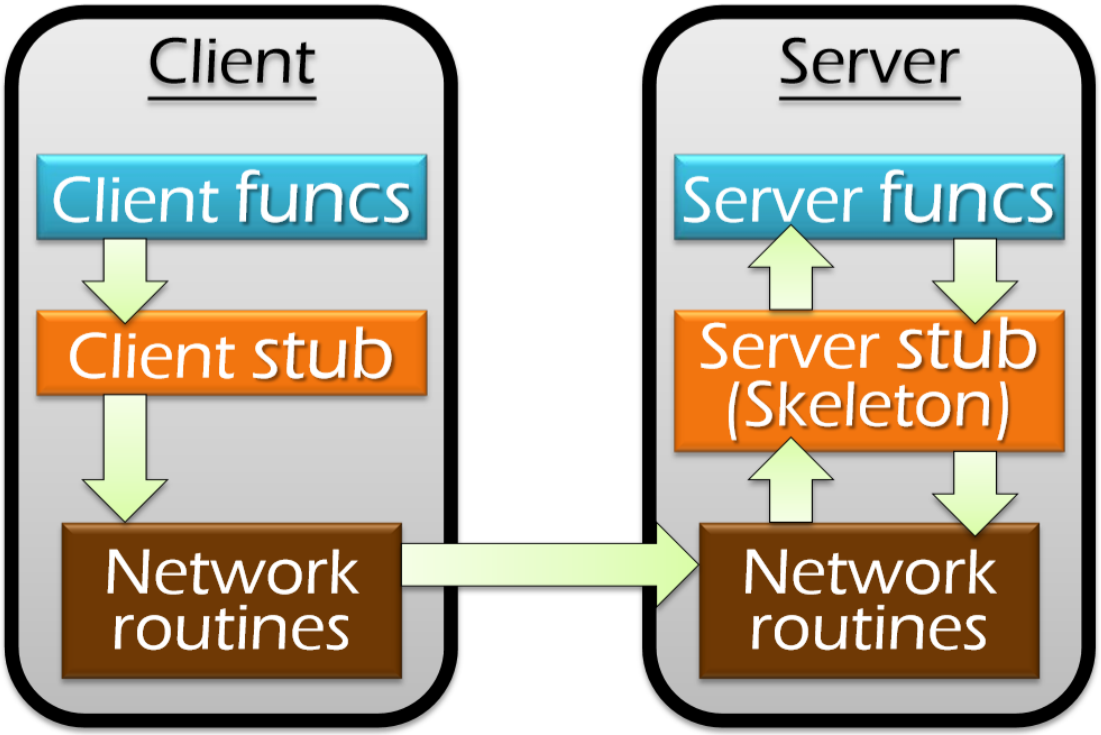
Stub Functions

6. Return from server function



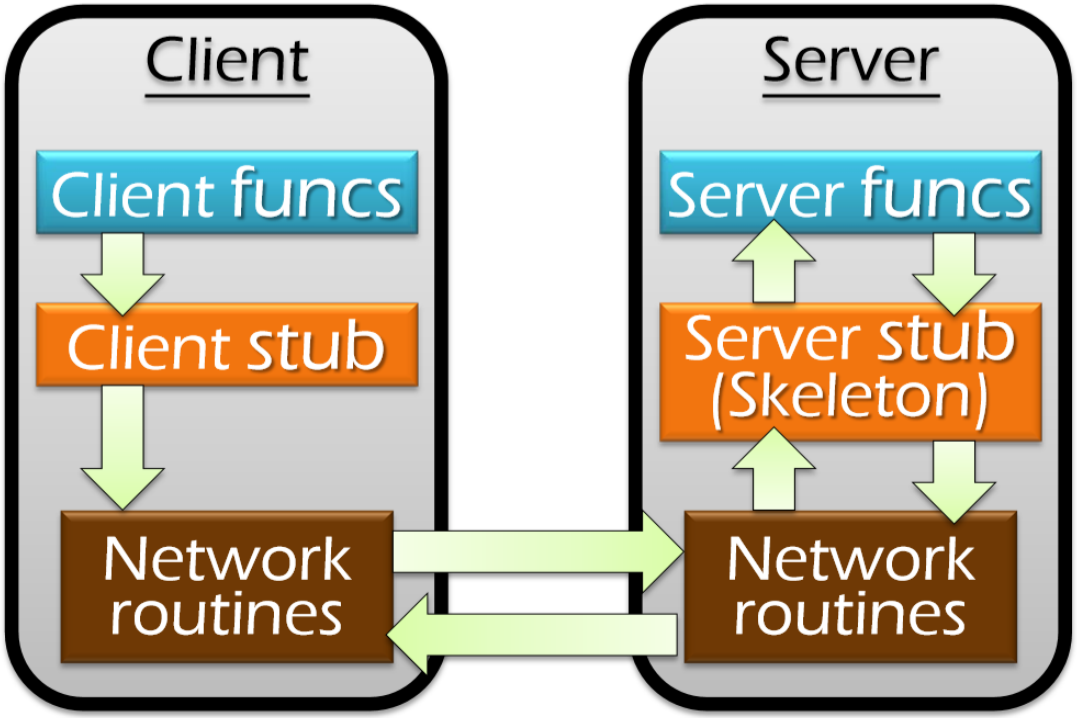
Stub Functions

7. Marshal return value and send **message**



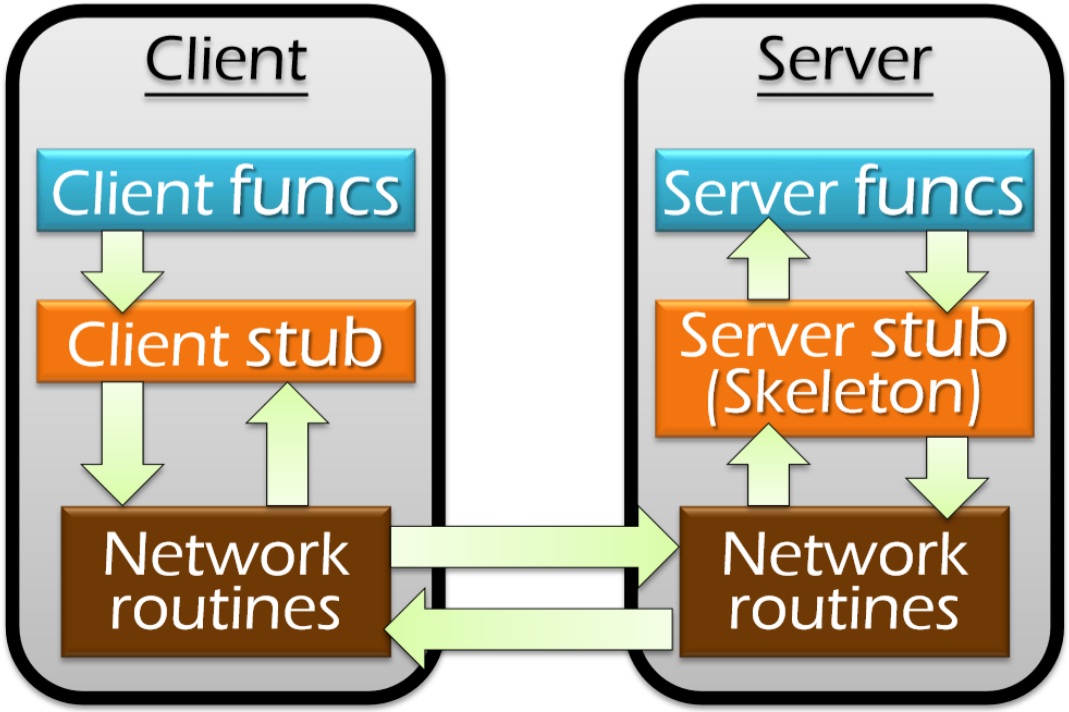
Stub Functions

8. Transfer **message** over network



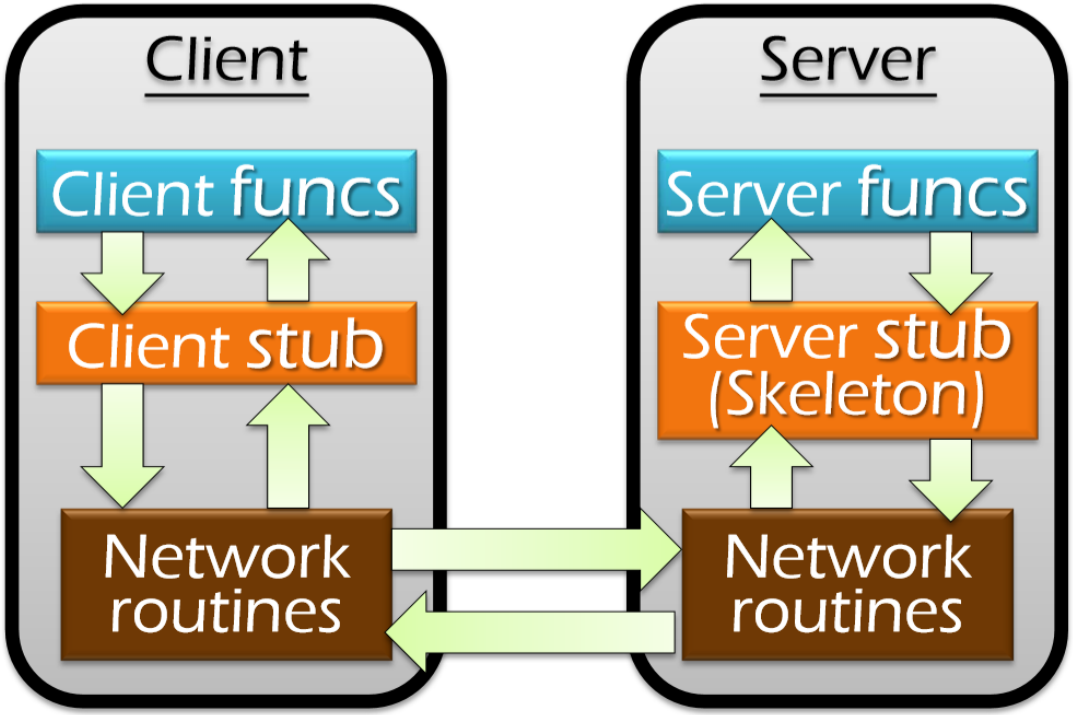
Stub Functions

9. Receive **message**: client **stub** is receiver

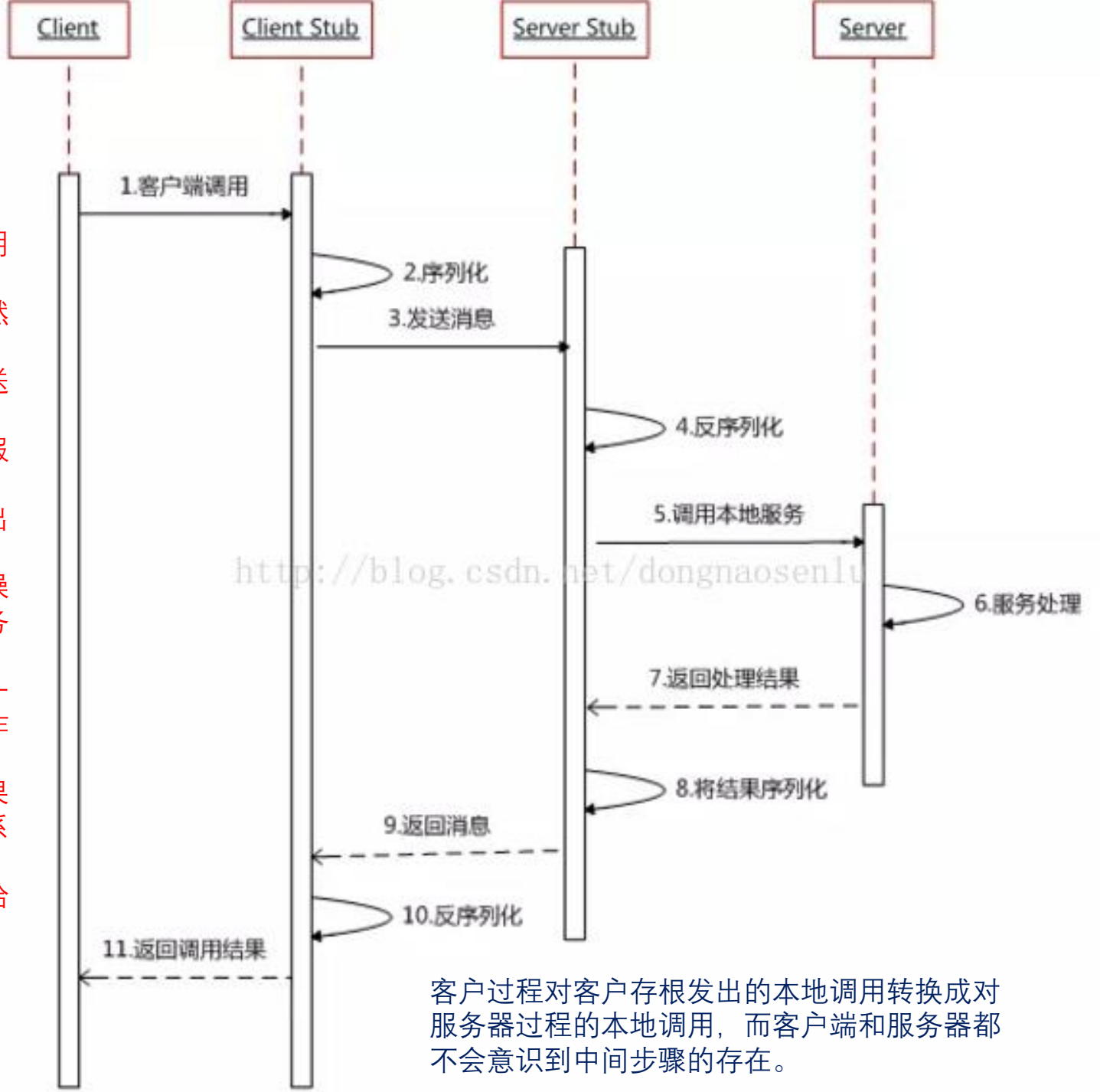


Stub Functions

10. Unmarshal return value, return to client code



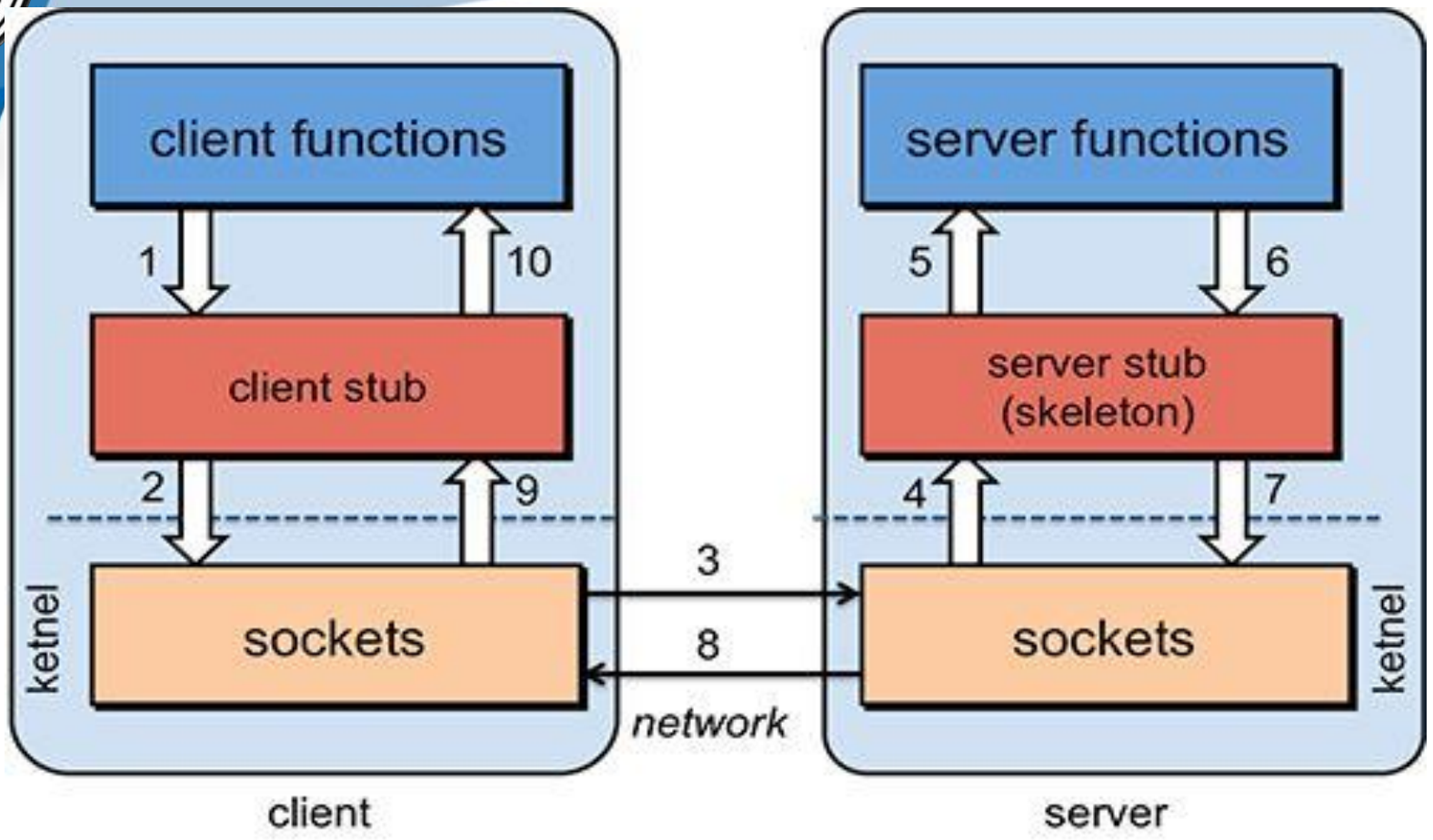
客户过程以正常的方式调用客户存根；
客户存根生成一个消息，然后调用本地操作系统；
客户端操作系统将消息发送给远程操作系统；
远程操作系统将消息交给服务器存根；
服务器存根调将参数提取出来，而后调用服务器；
服务器执行要求的操作，操作完成后将结果返回给服务器存根；
服务器存根将结果打包成一个消息，而后调用本地操作系统；
服务器操作系统将含有结果的消息发送给客户端操作系统；
客户端操作系统将消息交给客户存根；
客户存根将结果从消息中提取出来，返回给调用它的客户。



客户过程对客户存根发出的本地调用转换成对服务器过程的本地调用，而客户端和服务器都不会意识到中间步骤的存在。

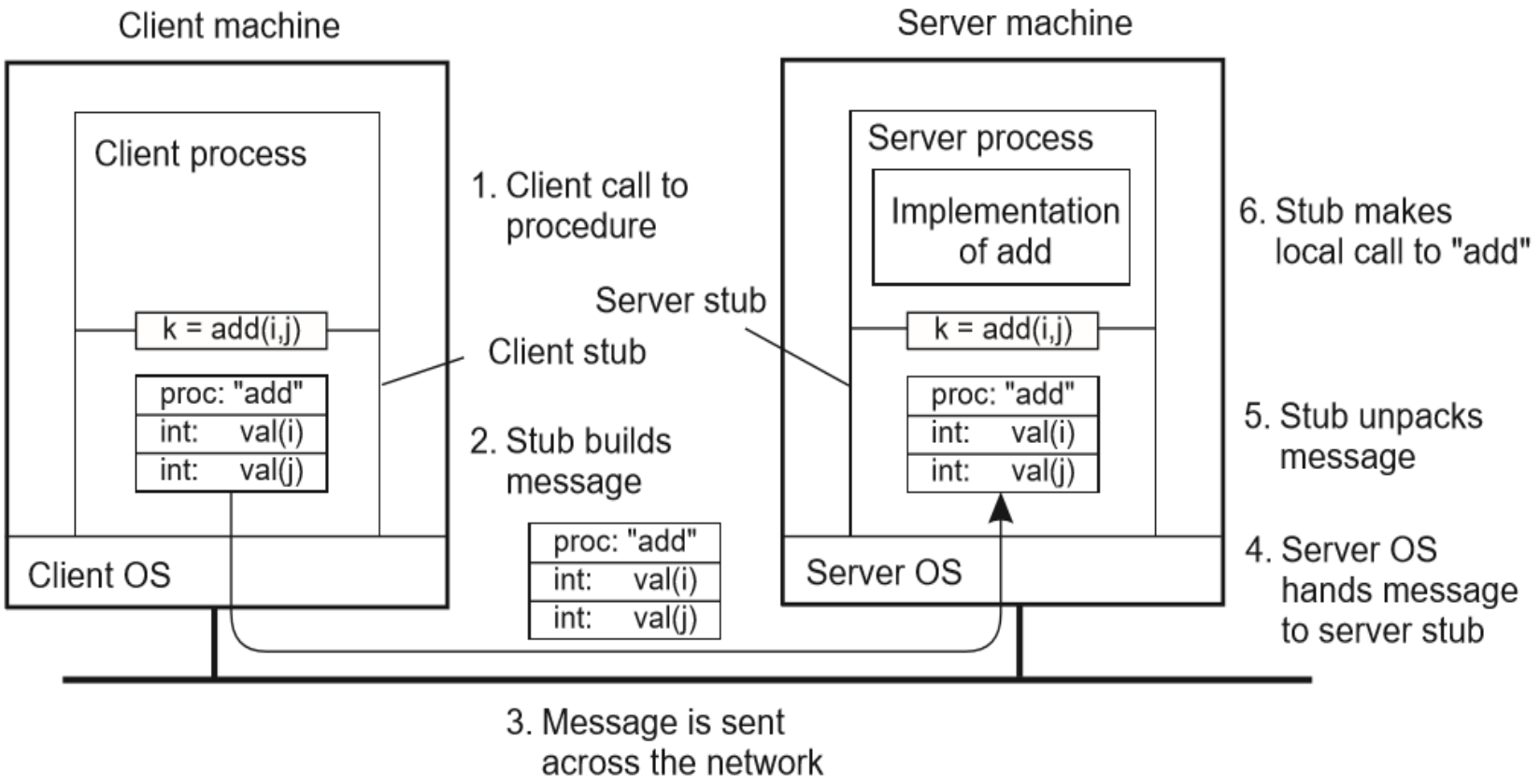
- 运行时系统：客户端的运行系统将客户端stub产生的消息可靠的传送给server，运行时系统利用TCP/UDP等协议，将消息发送到Server，服务端的运行时系统都侦听某个众所周知的socket端口，接受请求。

RPC通信过程



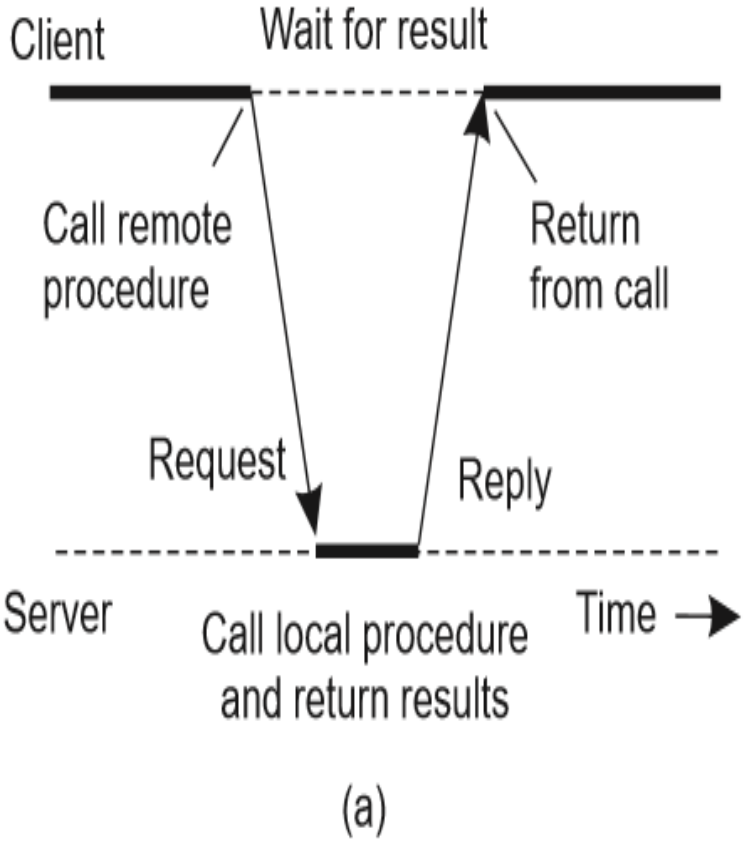
RPC 的主要好处是双重的。首先,程序员可以使用过程调用语义来调用远程函数并获取响应。其次,简化了编写分布式应用程序的难度,因为 RPC 隐藏了所有的网络代码存根函数。应用程序不必担心一些细节,比如 socket、端口号以及数据的转换和解析。在 OSI 参考模型,RPC 跨越了会话层和表示层。

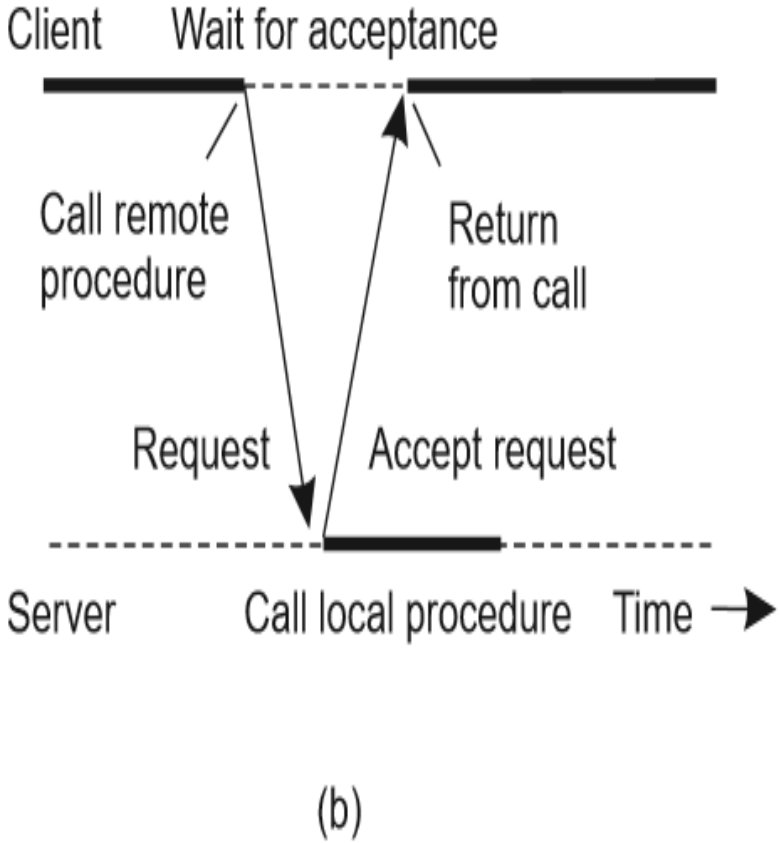
Basic RPC operation



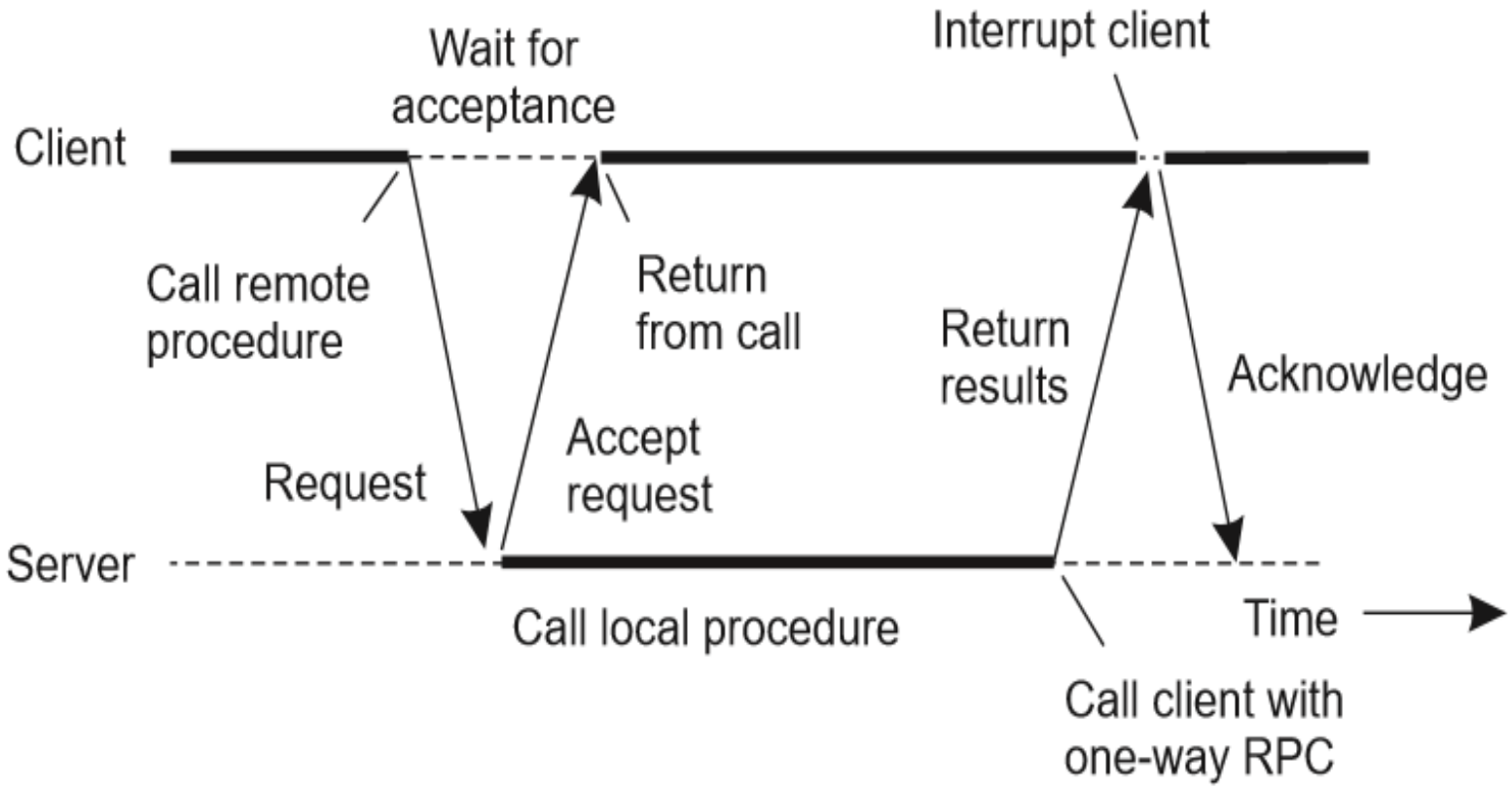
1. Client procedure calls client stub.
2. Stub builds message; calls local OS.
3. OS sends message to remote OS.
4. Remote OS gives message to stub.
5. Stub unpacks parameters and calls server.
6. Server returns result to stub.
7. Stub builds message; calls OS.
8. OS sends message to client's OS.
9. Client's OS gives message to stub.
10. Client stub unpacks result and returns to the client.

Synchronous RPCs





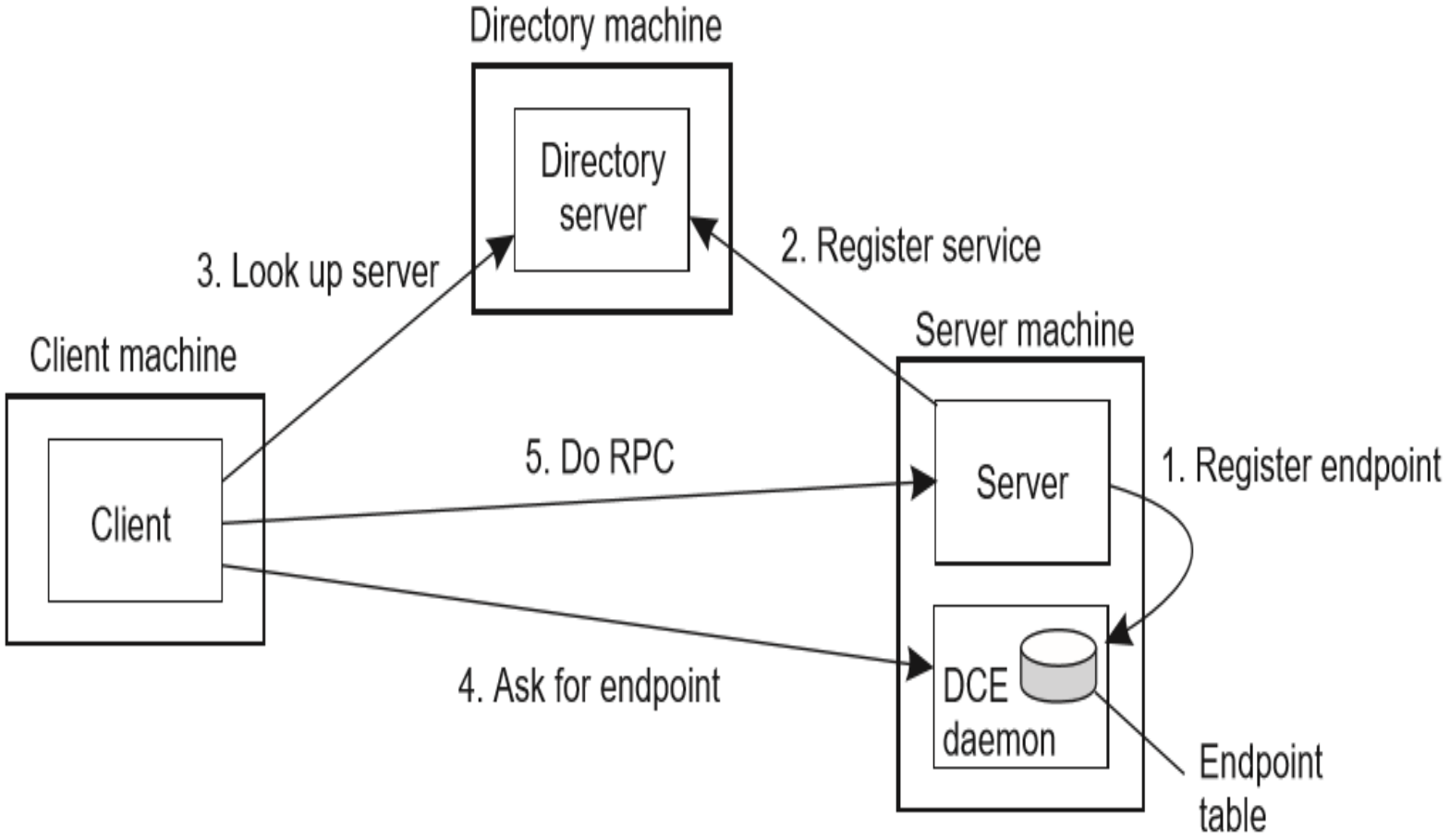
Deferred synchronous RPCs



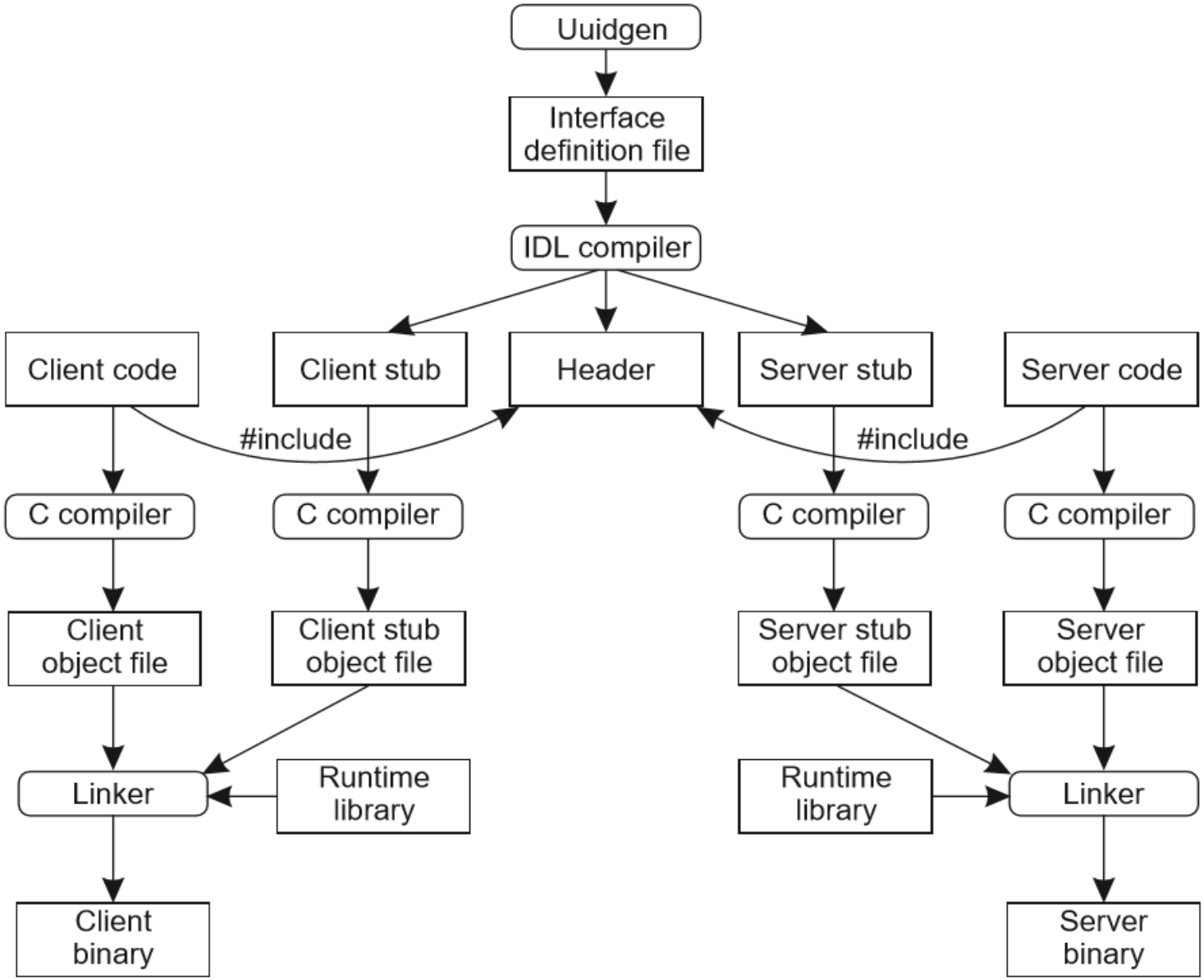
远程过程调用采用客户机/服务器(C/S)模式。请求程序就是一个客户机，而服务提供程序就是一台服务器。和常规或本地过程调用一样，远程过程调用是同步操作，在远程过程结果返回之前，需要暂时中止请求程序。

Client-to-server binding (DCE)

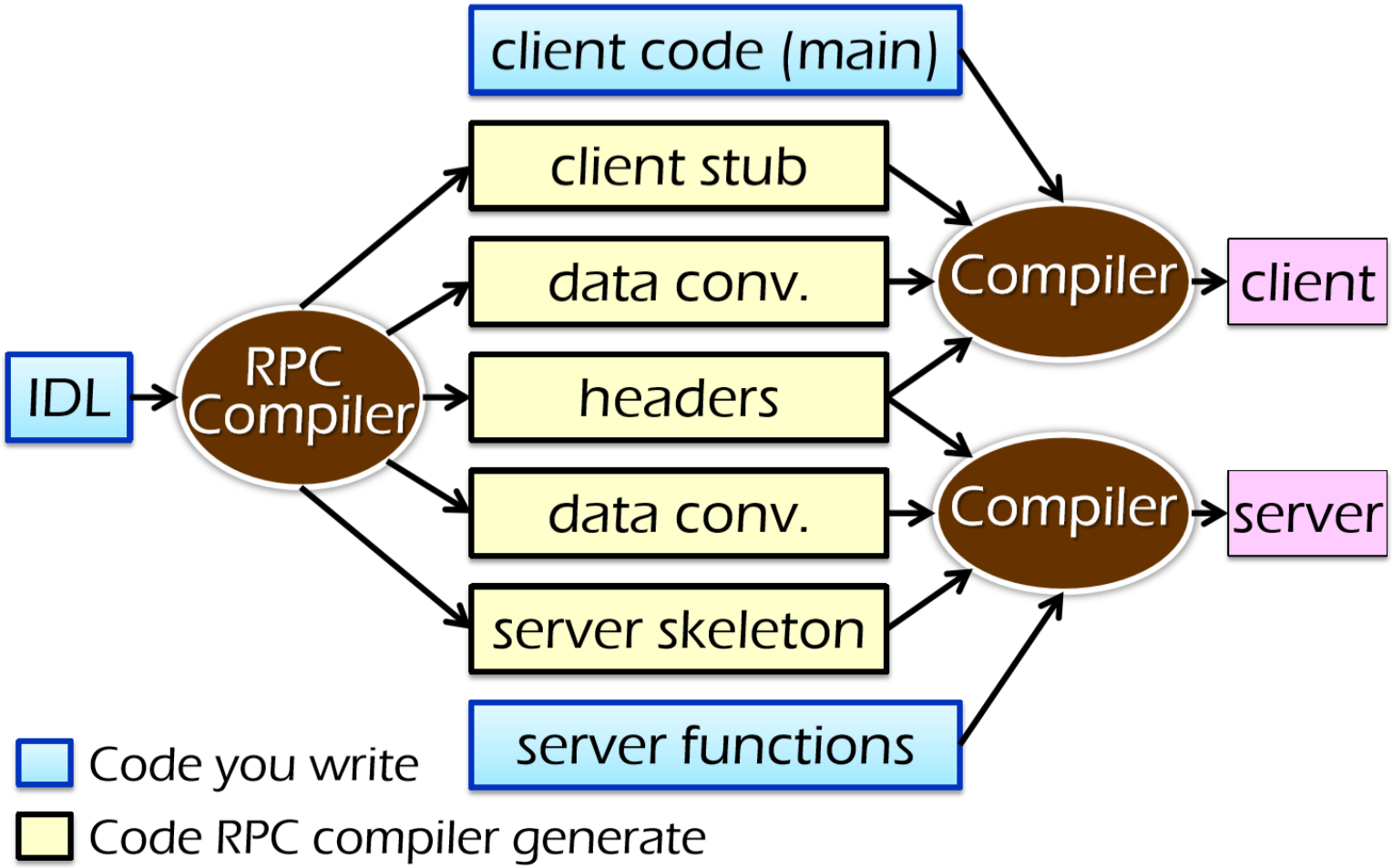
(1) Client must locate server machine, and (2) locate the server.



RPC in practice



RPC Compiler



XML

```
<?xml  version="1.0"  encoding="utf-8"?>
<country>
    <name>中国</name>
    <province>
        <name>黑龙江</name>
        <cities>
            <city>哈尔滨</city>
            <city>大庆</city>
        </cities>
    </province>
    <province>
        <name>广东</name>
        <cities>
            <city>广州</city>
            <city>深圳</city>
            <city>珠海</city>
        </cities>
    </province>
</country>
```

JSON(JavaScript Object Notation)

```
{
    "name": "中国",
    "province": [{
        "name": "黑龙江",
        "cities": {
            "city": ["哈尔滨", "大庆"]
        }
    }, {
        "name": "广东",
        "cities": {
            "city": ["广州", "深圳", "珠海"]
        }
    }]
}
```