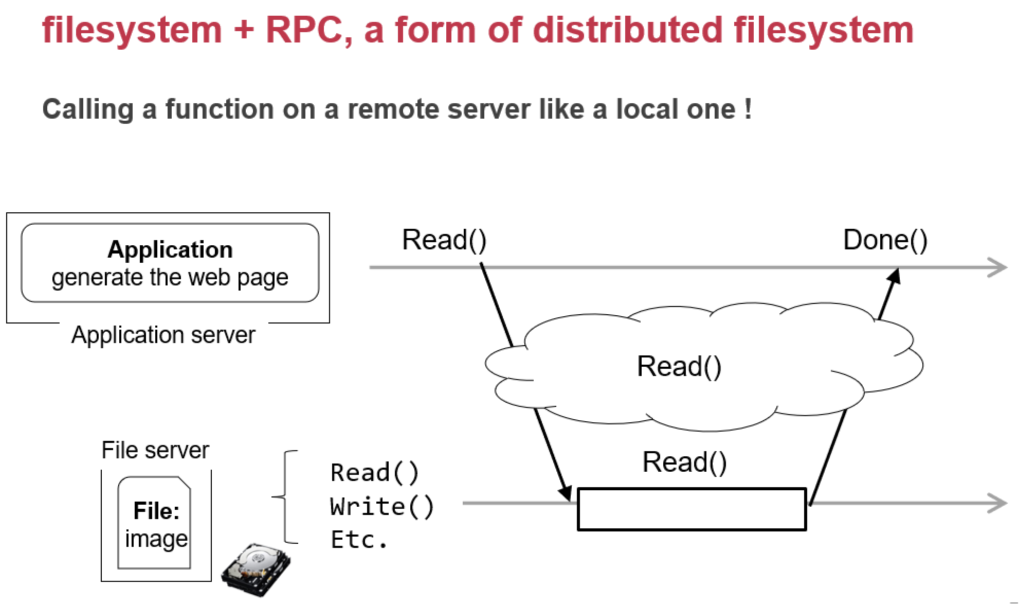
【计算机系统工程】RPC

@credits :Yubin Xia, IPADS

1. Why RPC?

我们希望把file system从单个扩展到多个。这个在接近40年前就有人做了。SUN公司希望开发没有硬盘的计算机，数据是不是都能全部来自网络（显示在一台机器上，运行在另一台机器上——这现在也是一个很先进的理念！比如云游戏！）。对用户是透明的，最底层的核心技术就是RPC。RPC不仅仅是用来提供Open，Write，Read接口，RPC还能实现跨机器的函数调用。



传统的，如果我们不用RPC，大家要调用网络的时候，就要使用socket，打开一个socket，服务端要listen，然后可以accept得到一个fd，然后在双方之间做交互，我们需要写很多glue code，不是很方便，我们希望有更加简单的操作，client的代码可以直接写一个fool()调用的是服务器端的函数，返回值就是对应的返回值。换句话说我们希望RPC和PC（procedure call）是一样的，我们不需要care它是local还是remote的。除了在framework级别，在语言级别甚至都包含了RPC相似的机制，比如Java中的RMI（remote method invocation）。说明RPC是一个非常非常基础的作用。

2. What RPC?

Example of RPC

我们希望测量函数的运行时间：

Example of RPC

```
procedure MEASURE (func)
    start ← GET_TIME (SECONDS)
    func () // invoke the function
    end ← GET_TIME (SECONDS)
    return end – start
```

```
procedure GET_TIME (units)
    time ← CLOCK
    time ← CONVERT_TO_UNITS (time, units)
    return time
```

The implementation of GET_TIME.

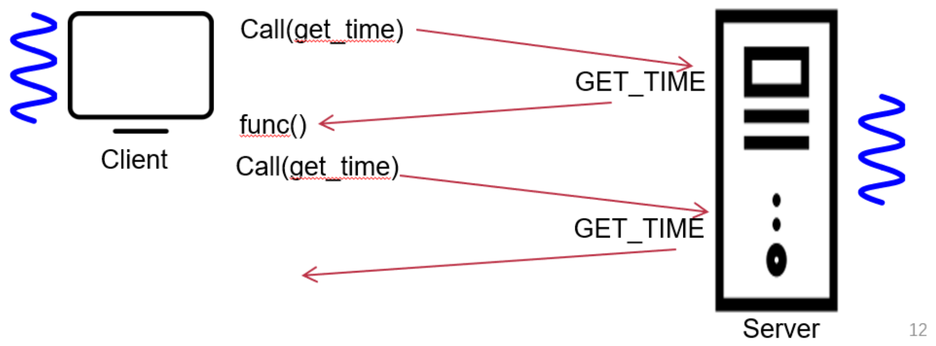
Suppose we want to measure the execution time of *func()*

Assumption:

– Only the server has the implementation of GET_TIME

How can the client call server's GET_TIME?

所以如果将 `GET_TIME` 放到服务器上，本地调用远程，应该如何做?

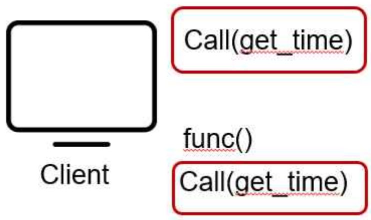


最简单的就是多ping几次，取均值，我们假设了网络时延基本上不变，并且假设了过去的时间等于回来的时间。一旦在谈到分布式的时候，我们会发现时钟是看起来简单但是背后是一个非常大的问题，因为没有两个人的时钟是一样的。

Handwritten code: client

Local

```
procedure MEASURE (func)
    start ← GET_TIME (SECONDS)
    func () // invoke the function
    end ← GET_TIME (SECONDS)
    return end – start
```

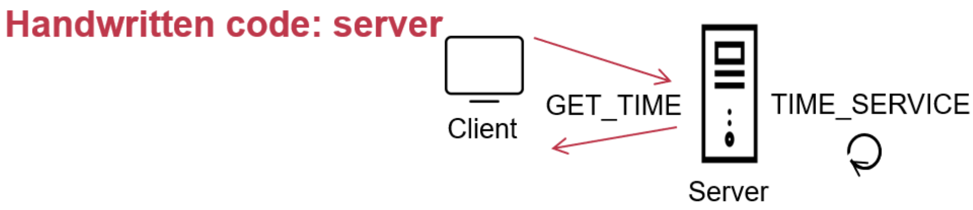


Client program

RPC

```
procedure MEASURE (func)
    SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
    response ← RECEIVE_MESSAGE (NameForClient)
    start ← CONVERT2INTERNAL (response)
    func ()      // invoke the function
    SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
    response ← RECEIVE_MESSAGE (NameForClient)
    end ← CONVERT2INTERNAL (response)
    return end – start
```

我们需要把second参数convert一下，因为网络上的数据是big-endian，然后传一个Get time打包成一个Message发送给服务器，得到response。然后再convert回来。




```
procedure TIME_SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE (NameForTimeService)
        opcode ← GET_OPCODE (request)
        unit ← CONVERT2INTERNAL(GET_ARGUMENT (request))
        if opcode = "Get time" and (unit = SECONDS or unit = MINUTES) then
            time ← CONVERT_TO_UNITS (CLOCK, unit)
            response ← {"OK", CONVERT2EXTERNAL (time)}
        else
            response ←{"Bad request"}
        SEND_MESSAGE (NameForClient, response)
```

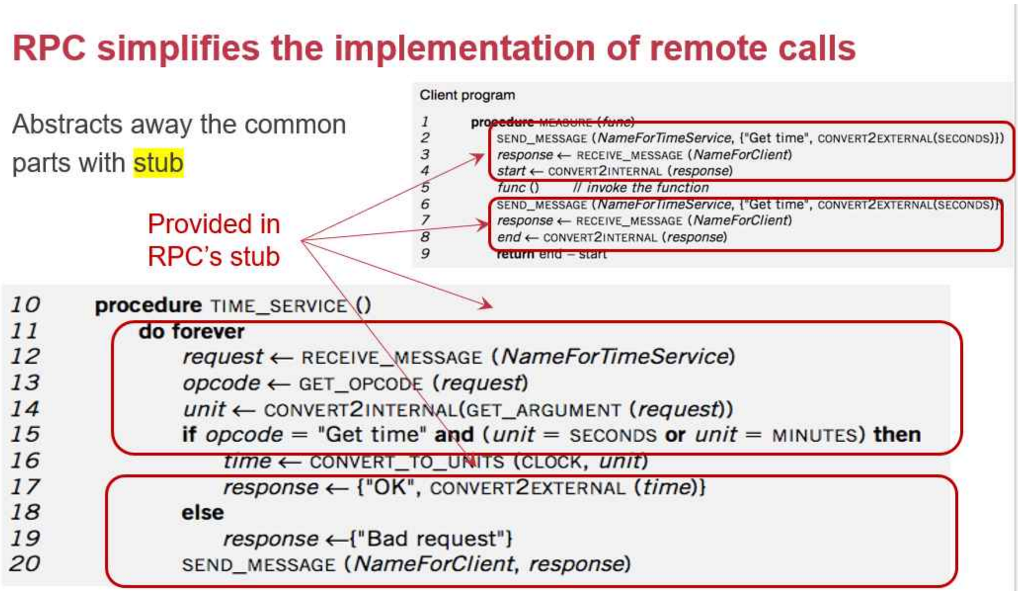
服务器做的事情，就是得到request后，对其做解析得到opcode和unit。得到数据以后，返回一个response（OK和bad request两种可能）。

3. How RPC?

RPC

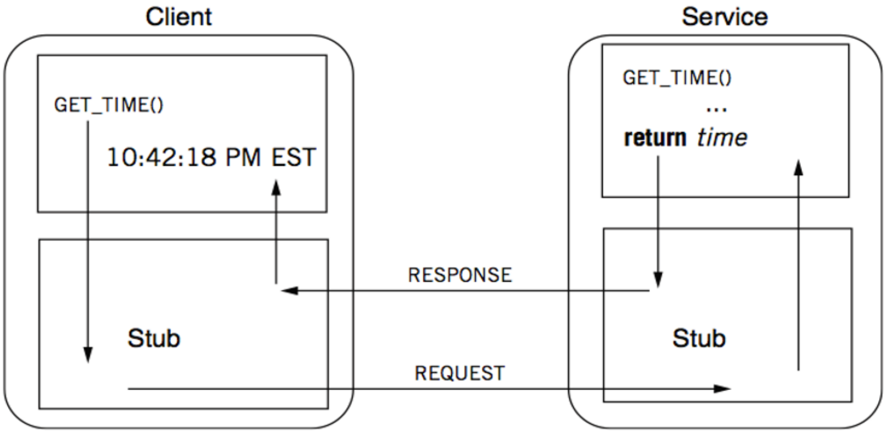
Basic idea

RPC就把上面这一大坨反手给你包装了一下，直接模块化：



这些都是可以放在RPC的stub（桩代码，不用程序员管的）中，全部用库的方式去封装起来。

RPC: a complete calling process



RPC stub

Client stub

– Put the arguments into a request
– Send the request to the server
– Wait for a response

> ***Stub**: hide communication details from up-level code, so that up-level code does not change.*

Service stub

– Wait for a message
– Get the parameters from the request
– Call a procedure according to the parameters (e.g. GET_TIME)
– Put the result into a response
– Send the response to the client

在stub中会实现GET_TIME并且发送给远端得到response。Stub就很好地封装了底层网络通讯的细节。

这是非常经典的解耦方式，程序员不用考虑函数是在本地还是在远端，当然这是在不出异常的情况。

SMTFC: Show me the friendly code!

Client Program using RPC

```
procedure MEASURE (func)
    start <- GET_TIME(SECONDS)
    func()
    end <- GET_TIME(SECONDS)
    return end - start
```

Note: code is not changed comparing with the single-machine case

This is the **stub** of client

```
procedure GET_TIME (units)
    SEND_MESSAGE(ServerName, {"Get time", CONVERT2EXTERNAL(units)})
    response <- RECEIVE_MESSAGE(ClientName)
    if GET_RETCODE(response) != "OK"
        HANDLE_ERROR(response)
    else
        return CONVERT2INTERNAL(GET_ARGUMENT(response))
```

Server Program using RPC

```
procedure GET_TIME (units)
    time <- CLOCK
    time <- CONVERT_TO_UNITS(time, units)
    return time
```

Note: this code is not changed

This is the **stub** of server

```
procedure TIME_SERVICE ()
    do forever
        request <- RECEIVE_MESSAGE(ServerName)
        opcode <- GET_OPCODE(request)
        arg <- CONVERT2INTERNAL(GET_ARGUMENT(request))
        if opcode = "Get time" and (arg = SECONDS or arg = MINUTES) then
            retval <- GET_TIME(arg)
            response <- {"OK", CONVERT2EXTERNAL(retval)}
        else
            response <- {"Bad request"}
        SEND_MESSAGE(ClientName, response)
```

What's inside the request message?

Question: what is inside a message?

```
procedure GET_TIME (units)
    SEND_MESSAGE(ServerName, {"Get time", CONVERT2EXTERNAL(units)})
    response <- RECEIVE_MESSAGE(ClientName)
    if GET_RETCODE(response) != "OK"
        HANDLE_ERROR(response)
    else
        return CONVERT2INTERNAL(GET_ARGUMENT(response))
```

```
procedure TIME_SERVICE ()
    do forever
        request <- RECEIVE_MESSAGE(ServerName)
        opcode <- GET_OPCODE(request)
        arg <- CONVERT2INTERNAL(GET_ARGUMENT(request))
        if opcode = "Get time" and (arg = SECONDS or arg = MINUTES) then
            retval <- GET_TIME(arg)
            response <- {"OK", CONVERT2EXTERNAL(retval)}
        else
            response <- {"Bad request"}
        SEND_MESSAGE(ClientName, response)
```

20

我们需要把数据用合理的方式marshal（序列化，又称为serialize）排队发到网上去。

RPC request message

RPC request:

- **Xid** → X is short for "transaction"
  Client reply dispatch uses xid
  Client remembers the xid of each call
- call/reply
- rpc version
- **program #**
- program version
- **procedure #**
- auth stuff
- arguments

Server dispatch uses prog#, proc#

RPC reply message

RPC reply:

- **Xid**
- call/reply
- **accepted**? (Yes, or No due to bad RPC version, auth failure, etc.)
- auth stuff
- **success**? (Yes, or No due to bad prog/proc #, etc.)
- **results**

Request

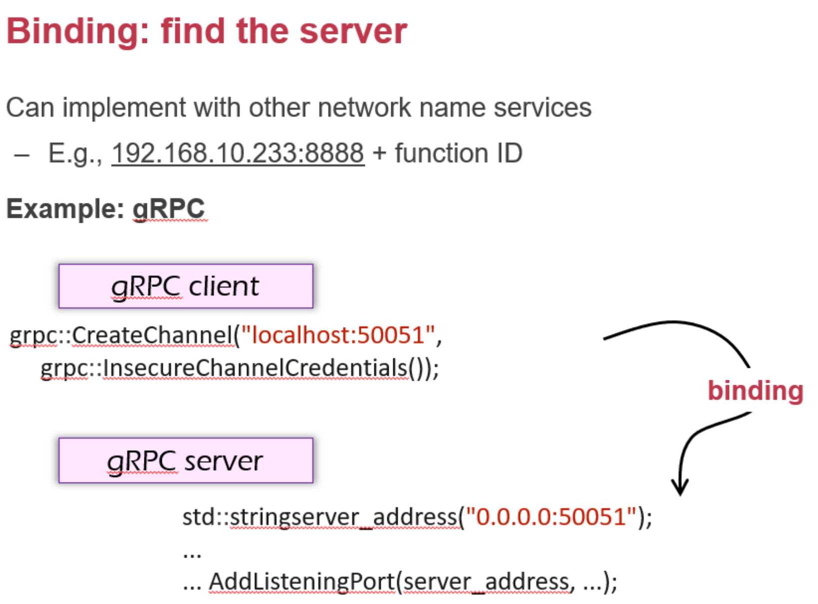
1. Xid，X是transaction的缩写，它是用告诉server请求是发过了还是没发过，在出错的时候非常有用
2. Call/reply
3. RPC version，意味着可以支持多个version，一台单机的应用程序和库通常是合在一起的，但是client和server的版本可能不相同，要考虑到兼容性
4. Program, program version, procedure，可能调用一个可执行文件所提供的函数
5. Auth stuff，来做验证
6. Arguments，参数

Reply

1. Accepted
2. Success， accepted和success的区别是什么呢？Accepted是和RPC相关的，而Success是和我们调用的serveice相关的。换句话说如果accepted是false，说明RPC阶段就错了，没有调用到程序，而success是说明RPC之间是可以正常对话，但是具体执行的时候出错了

3. Results
4. Auth stuff

Binding: find the server



以Google的gRPC为例。一开始的时候我们要create a channel。我们这里假设的本机到本机，那ip就知道了，然后是端口，假设50051，连过去了。对于server来说，一开始就要定好端口。后面就是通过channel来传。

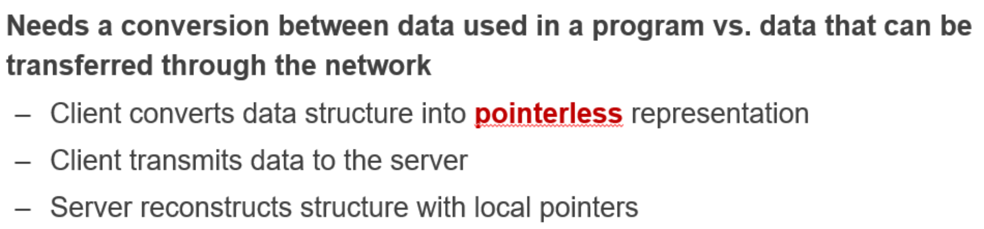
How to pass data?

By value or by reference?

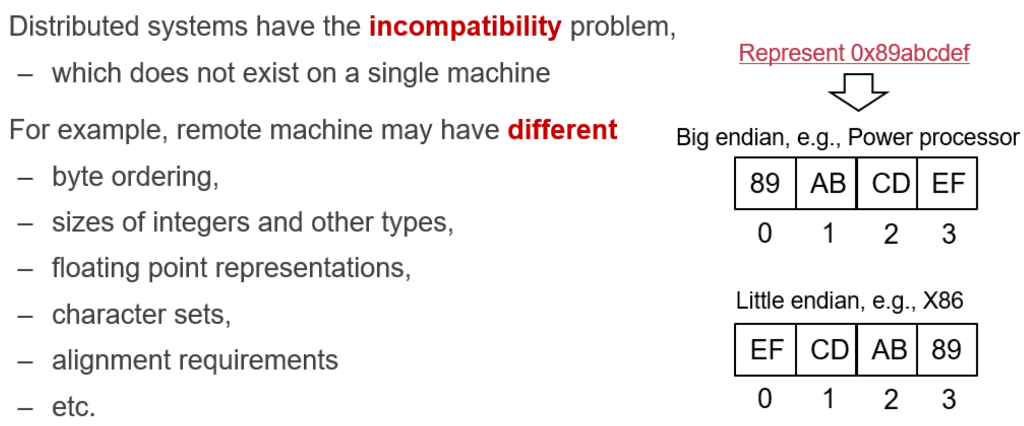
我们在传paramater的时候，必须pass by value而不是pass by reference（跨机器的时候，地址空间是完全不一样的，传指针没有任何的意义；我们先假设最简单的场景，两者都有各自的地址空间，我们用值传递来做）。

当然其实也是可以有引用传递的——只要我们大家跨机器地共享一块memory；但我们这里先不讨论这种复杂的情况，我们只考虑client和server彼此各有各的地址空间。

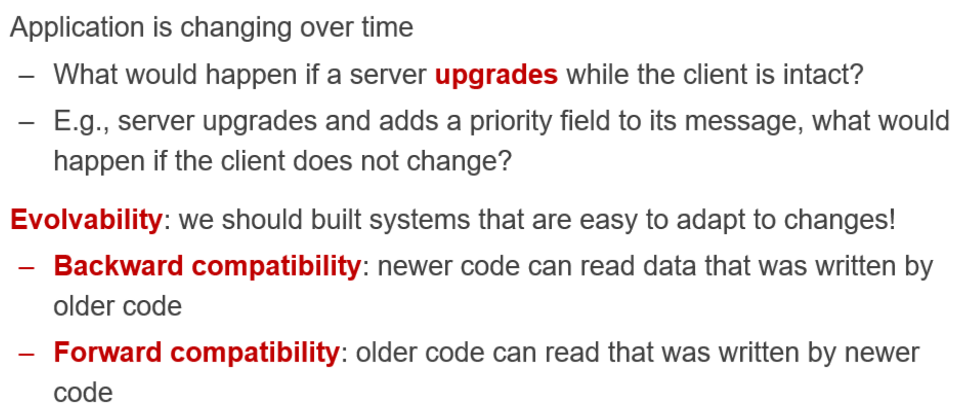
所以实际操作是：



Challenges



传参到底有什么挑战呢？核心就是每台机子对数字的理解是不一样的，比如IEEE 754的浮点解释标准……etc。是64位还是32位还是更大，包括对齐方式等，否则我们的数据就会发生丢失，为了解决这些问题，就必须要有一些标准，大家统一满足什么规范。



还有更大的挑战：应用会更新！比如server更新了，但是client没有，就出现了不一致。

所以我们希望：

1. 新代码兼容旧的
2. 旧代码兼容新的：这听上去诡异，但也不是完全不可能。比如不认识的就跳过，别挂就行，尽可能去操作即可。

Encoding

所以需要一种data的表达方式，本质上就是大家都认可的一种标准。encoding就是把一串数据变成网络上的数据流，decoding就是把网络上的数据流转换为内存中的数据。

Need standard **encoding** to enable communication between heterogeneous systems & different versions of software

- Sun's RPC uses XDR (eXternal Data Representation)
- ASN.1 (ISO Abstract Syntax Notation)
- JSON
- Google Protocol Buffers
- W3C XML Schema Language

为什么不用language specific format（语言内置的表达方式）呢？比如java提供了原生的serializable接口，一旦继承了它，就可以变成序列化，Python也有pickle，但是缺点就是把我们自己绑死在某个语言上了，并且兼容性也不好——你客户端用java写的，我服务端用python写的。

System requirements for encoding/decoding

Transfer objects through the network

- Correctly encode and decode a object to a byte stream

Compatibility

- Support multiple language, multiple versions of program

Efficiency

- Reduce the traffic transferred from the network
- Network bandwidth is a scarce resource

说到高效，xml肯定不是一个好的选择。

JSON，XML...

Independent to a specific programming language

Textual formats:

- JSON
- XML
- CSV

Logic client request format

- Xid
- call/reply
- rpc version
- program #
- program version
- procedure #
- auth stuff
- arguments

JSON representation

```
{ "xid" : 12, "call": true,
  "rpc_version": 73,
  ...
}
```

这种格式的好处：human-readable，easy to debug。

缺点：

1. 关键的数据结构的encoding会出现二义性，比如12这个数字我们不知道type是什么，是signed还是unsigned之类的；
2. 如果我有一个string，它就是binary的文件，我们不得不转换成基于base64的格式，才可以放入到json里面去；但是这又是手动要做这样的一个转换；
3. 冗余：XML里面有各种tag。

Binary formats

好处就是很容易压缩、很紧凑，更精确（我就告诉你这是个uint32，你不用给他64个bit），很快就可以变成内存中的数据结构；缺点就是人类要读懂就很难。

Binary formats: schema

Both Thrift and Protocol Buffers require a **schema** for any data that is encoded

- **Benefits**: no need to encode things such as userName in the encoded data

Thrift interface definition language (IDL)

```
struct Person {
  1: required string userName,
  2: optional i64 favoriteNumber,
  3: optional list<string> interests
}
```

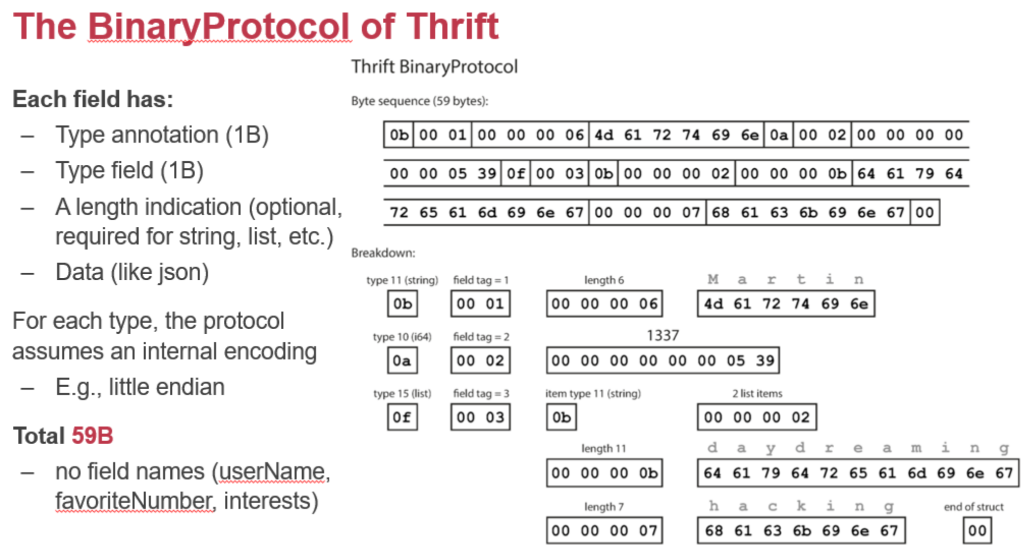
Protocol Buffers IDL

```
message Person {
   required string user_name = 1;
   optional int64 favorite_number = 2;
   repeated string interests = 3;
 }
```

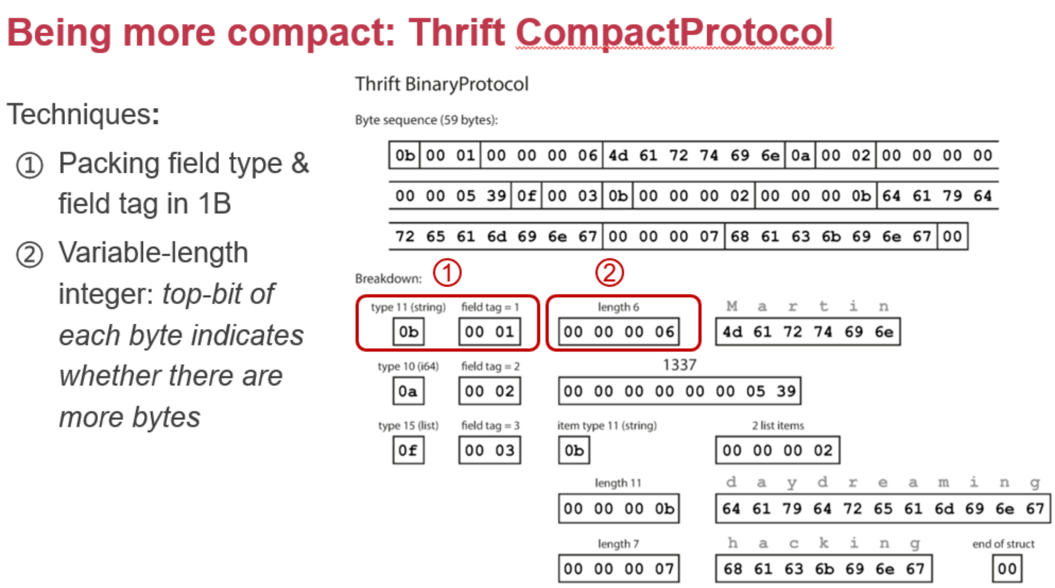
```
{ "userName": "Martin", "favoriteNumber":
1337, "interests": ["daydreaming", "hacking"] }
```

JSON representation takes 81B in total (w/o spaces)

JSON需要81个字节；但是我们如果用binary format，我们可以狠狠压缩，甚至让压缩比超过一半。这个协议的名字叫做Thrift。

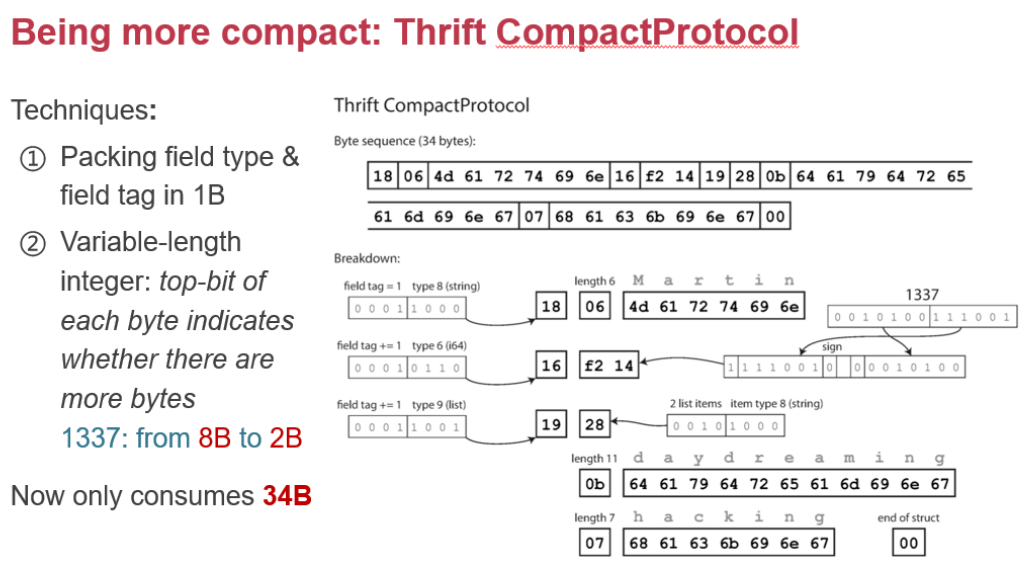


我们可以进一步去压缩，为什么要把type放在最头上呢，这个顺序是很关键的，当我们在做一个server收到一个request，一开始什么都不知道，读到的第一个byte就很关键，告诉我们后面是一个string，这个string是一个1号（username），接下来串一个length=6，后来是传6个byte。然后是i64类型……，真正有用的数据都是data，前面这些数据都是metadata，为了组织这些数据，metadata是必需的。



1. 把3个byte变成一个byte，因为tag不会太大
2. Length很浪费，我们可以用一个变长的8位list，也就是每8位如果第一位为1，说明剩余7位中有数字。

最后结果如下：



Failure

当RPC遇到FAILURE怎么办?

对于一个local的procedure call的情况下，failure的情况下是很少的。Call就是把地址压栈跳转到对应的函数地址去执行，如果这种情况下都出错了，说明计算机有很大的问题了，这个程序基本上会被kill掉，一般来说我们不去考虑call带来的问题。

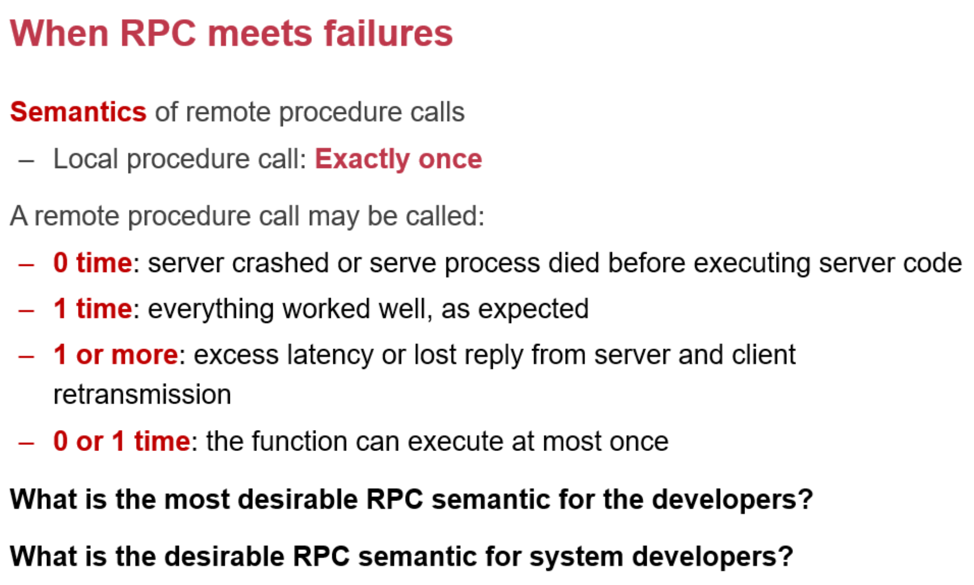
那么如果变成了RPC，遇到问题的概率远远大于本地的调用。序列化出错、RPC版本出错、对方server crash、网络出错，当一个RPC调用没有得到任何返回的时候，有以下情况：

网络问题：

1. 包没有到server（比如挖掘机把线挖断了）
2. Server的response没有回到client。

我们没有办法区分出上面两者。

1. 远程crash（比如server没电了）
2. Request队列（比如服务器过载了，请求还在队列中等待）
3. 远程暂时停止响应了（比如服务器正在进行一段耗时较长的垃圾回收）
4. 远程执行了，但是你超时了（比如你自己机器的网络过载了）

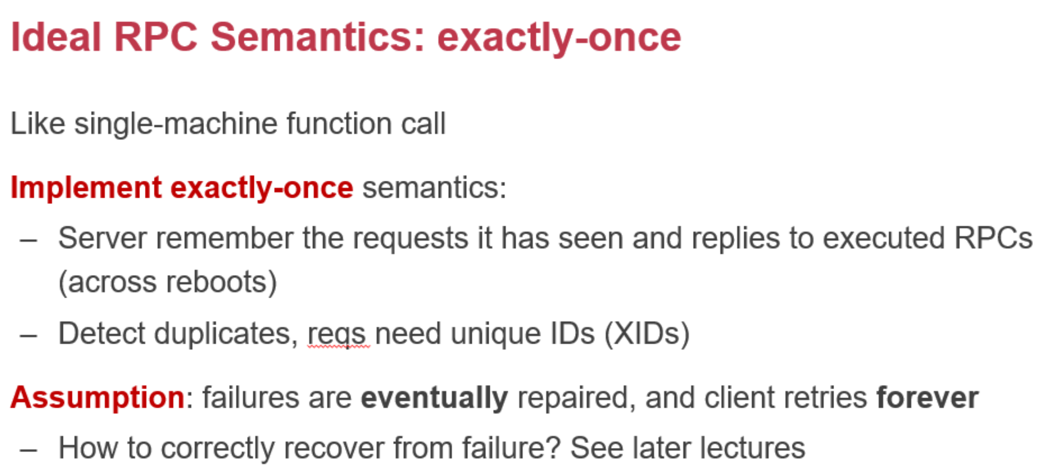


RPC调用以后等多久？这个时间就很tricky，只能根据经验来设置。对于RPC来说，会return一个错误status。RPC的期望结果，exactly once，我们会遇到：

1. 0次，server没执行
2. 1次，正常
3. 1次或多次，client可能等不及了再发一次。

所以RPC一般提供at least once（没收到response就重试，直到ok）和at most once（只发送一次）。

幂等性：我们希望实现at least once，我们能不能让一个操作执行一次和执行两次是一样的。比如按电梯：按三次和按一次是一样的，但是存钱不是幂等的。如果实现不了幂等性，怎么实现at least once呢，我们要有别的机制去记录。Server可以把执行成功的xid（还记得X是什么吗？transaction！）记录下来，如果发现已经做过了那就返回OK。



Take away message

1. Standards for wire format of RPC message and data types

2. Library of routines to **marshal / unmarshal** data
3. Stub generator, or RPC compiler, to produce "stubs"
    a. For client: marshal arguments, call, wait, unmarshal reply
    b. For server: unmarshal arguments, call real function, marshal reply
4. Server framework:
    a. Dispatch each call message to correct server stub
    b. Recall each called functions ,if provide **at-most-once** semantic or **exactly-once semantic**
5. Client framework:
    a. Give each reply to correct waiting thread / callback
    b. Retry if timeout or server cache
6. Binding: how does client find the right server?