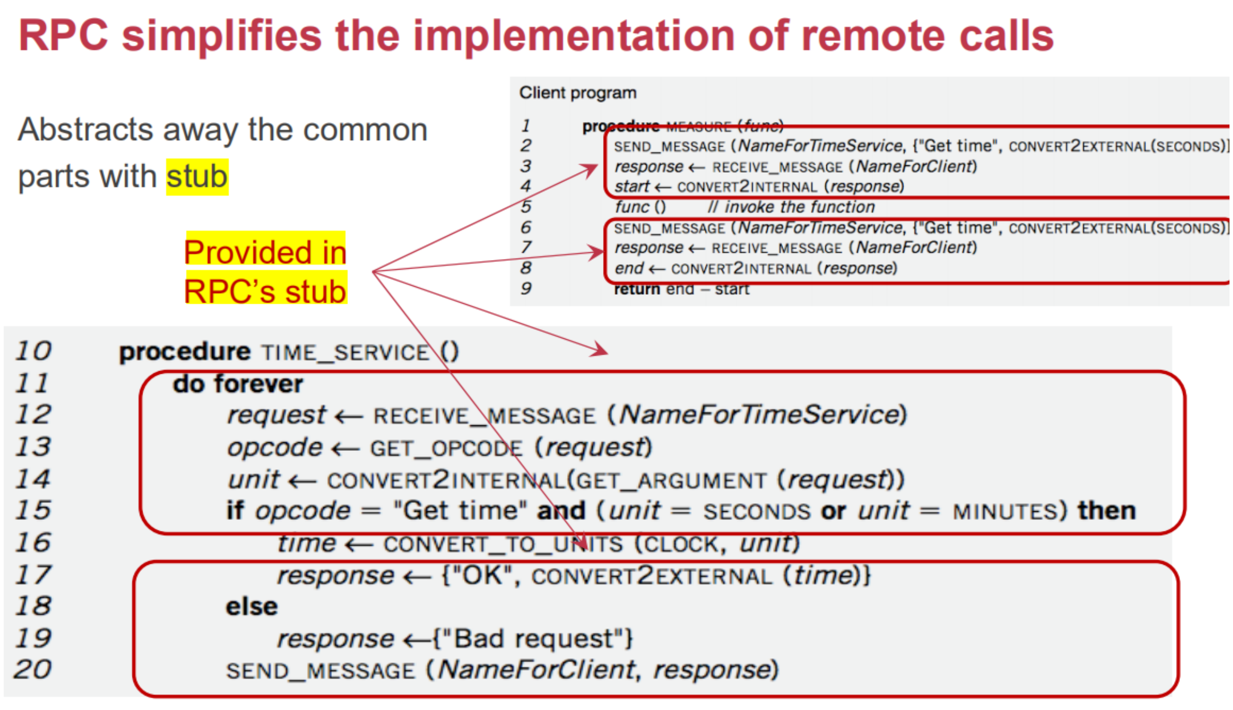
RPC: Remote Procedure Call

stub提供包装和抽象：



Client stub

- Put the arguments into a request
- Send the request to the server
- Wait for a message

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

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

inside message:

- Service ID (e.g., function ID)
- Service parameter (e.g., function parameter)
- Using marshal / unmarshal(序列化，消除对于操作系统和语言的依赖)

RPC request message

RPC request:

- **Xid** → X is short for "transaction"
  Client reply dispatch uses xid
  Client remembers the xid 用来解析 of each call
- call/reply
- rpc version → 保证client和server的rpc version是一致的
- **program #**
- program version
- **procedure #** → program中的一个过程（函数）
  Server dispatch uses prog#, proc#
- auth stuff → 验证权限
- arguments

*

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**

两个分开，可以提高效率

paramater passing

- Pass by **value**?   Easy: just copy data to network message
- not Pass by **reference** Makes no sense without shared memory
- Process
  - Client converts data structure into **pointerless** representation
  - Client transmits data to the server
  - Server reconstructs structure with local pointers

Evolvability: we should built systems that are easy to adapt to changes

- **Backward compatibility**: newer code can read data that was written by older code
- **Forward compatibility**: older code can read that was written by newer code

encoding:

- P26-P30: standardized encoding
- **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;
}
```

- **The BinaryProtocol of Thrift**

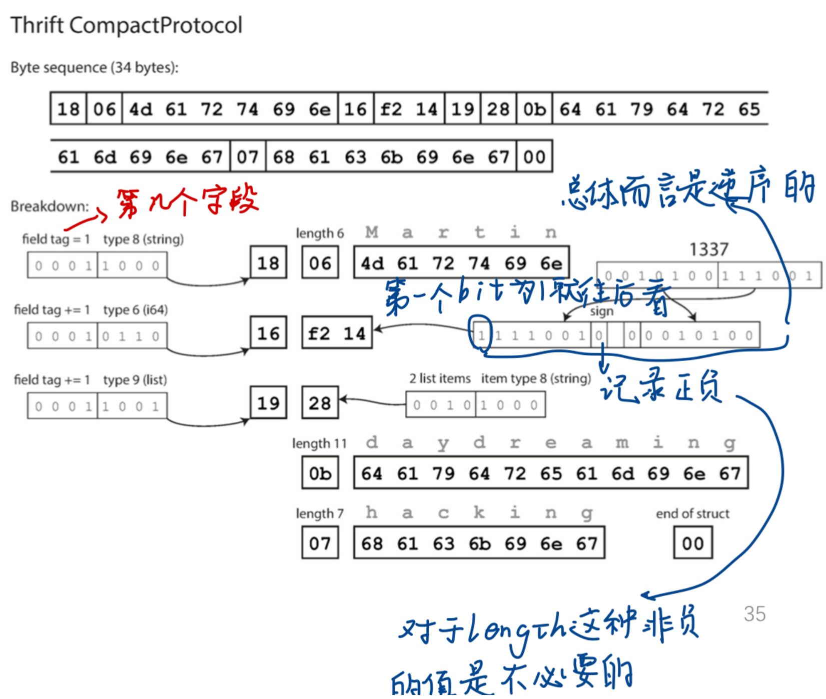
Techniques:

① Packing field type & field tag in 1B

② Variable-length integer: *top-bit of each byte indicates whether there are more bytes*

1337: from 8B to 2B

Now only consumes **34B**



- **Schema simplifies supporting compatibility(P36)**
  - **Forward compatible**
  - **Backward compatible**
- **Transport protocol of RPC(P39)**

RPC优点：

- RPC simplifies programming w/ an interface similar to local function call
- RPC uses stubs to avoid handling argument **encoding/decoding** and send/receiving messages, etc. – Ensure correctness & efficiency

A user sends an RPC but the server does not reply, possible reasons（P44、45）

RPC semantic

Most RPC systems will offer either:

- **At-least-once** semantics 没有成功不断重试
- **At-most-once** semantics 没有成功,也不会重试,成功1次或0次

Simple **retransmission** leads to "**at-least-once**"

Birrell's RPC semantics :

- server says **OK**: executed once
- server says **CRASH**: zero or one time

much **easier** than exactly once, more **useful** than at-least-once

Understand application: → 幂等操作,每次访问结果是一致的

- **Idempotent**: may be run any number of times without harm (e.g., i = 1)
- **Non-idempotent**: those with side-effects (e.g., i++) → 做不同次结果不同

When **at-least-once** is **OK**?

- if no side effects (e.g., read-only operation)
- if app has its own plan for detecting duplication

Ideal RPC Semantics: exactly-once

Like single-machine function call

Implement exactly-once semantics:

– Server remembers the requests it has seen and replies to executed RPCs (need to across reboots)

– **Detect duplicates**, requests need unique IDs (XIDs)

summary

Put it all together: RPC system components

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"
   – For client: marshal arguments, call, wait, unmarshal reply
   – For server: unmarshal arguments, call real function, marshal reply

4. Server framework:

– Dispatch each call message to correct server stub

– Recall each called functions ,if provide **at-most-once** semantic or **exactly-once semantic**

5. Client framework:

– Give each reply to correct waiting thread / callback

– Retry if timeout or server cache

6. Binding: how does client find the right server?