

Biblioteca RPC



PUC
RIO

Professora: Noemi Rodriguez
Aluno: Fernando Homem da Costa

INF2545 - Sistemas Distribuídos
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
4 de abril de 2021

Lista de Figuras

1	Executando o Servidor RPC	7
2	Porta Gerada pelo Servidor RPC	8
3	Cliente RPC	8
4	Código Cliente - Chamada a Função Inexiste - coo	9
5	Resultado no Terminal - Caso 1	9
6	Definição da Função foo	10
7	Código Cliente - Parâmetro extra	11
8	Resultado Terminal - Caso 2	11
9	Código Cliente - Parâmetro Faltando	12
10	Resultado Terminal - Caso 3	12
11	Código Cliente - Parâmetro Tipo Incorreto	13
12	Resultado Terminal - Caso 4	13
13	Nova Definição Interface	14
14	Código Cliente - Parâmetro “nil”	15
15	Resultado Terminal - Caso 5	15
16	Arquivo IDL - Definição Struct	16
17	Código Cliente - Função foo	17
18	Resultado Terminal - Caso 6	17

Conteúdo

1	Introdução	3
2	Executar	7
2.1	Bibliotecas	7
2.2	Servidor RPC	7
2.3	Cliente RPC	7
3	Simulações	9
3.1	Função Inexistente	9
3.2	Excesso de Parâmetros	9
3.3	Falta de Parâmetros	11
3.4	Tipo de Parâmetro Incorreto	12
3.5	Parâmetro Especial “nil”	13
3.6	<i>Struct</i> Com Tipos de Campos Inválidos	15
4	Conclusão	18

1 Introdução

Este trabalho é o primeiro projeto da disciplina de Sistemas Distribuídos (INF2545) do primeiro semestre de 2021, ministrada pela professora Noemi Rodriguez.

Nesta tarefa, deve-se construir um sistema de apoio a chamadas remotas de métodos utilizando Lua e LuaSocket. O objetivo é implementar uma biblioteca luarpc contendo métodos `lrpc.createServant`, `lrpc.waitIncoming` e `lrpc.createProxy`.

A idéia é partir de uma especificação do objeto remoto como abaixo:

```
struct {
    name = "minhaStruct",
    fields = {
        {name = "nome",
         type = "string"},
        {name = "peso",
         type = "double"},
        {name = "idade",
         type = "int"},
    }
}

interface {
    name = "minhaInt",
    methods = {
        foo = {
            resulttype = "double",
            args = {
                {direction = "in",
                 type = "double"},
                {direction = "in",
                 type = "string"},
                {direction = "in",
                 type = "minhaStruct"},
                {direction = "out",
                 type = "int"}
            }
        },
        boo = {
            resulttype = "void",
            args = {
```

```

        {direction = "in",
        type = "double"},
        {direction = "out",
        type = "minhaStruct"}
    }
}
}

```

Essa é uma especificação que pode ser lida diretamente pelo programa Lua, caso tenha uma função chamada `interface` tenha sido definida - a sintaxe de Lua permite que uma função com um único argumento do tipo tabela seja chamada sem os parênteses, como acima.

Além disso, o arquivo de interface conterá um texto como o do exemplo acima, sempre contendo apenas uma interface. Os tipos que podem aparecer nessa especificação de interface são **char**, **string**, **double** e **void**. Os parâmetros podem ser declarados como `in` ou `out`. O seguinte trecho de programa Lua criaria dois servidores com a interface acima:

```

myobj1 = { foo =
    function (a, s, st, n)
        return a*2, string.len(s) + st.idade
        + n
    end,
    boo =
    function (n)
        return n, { nome = "Bia", idade = 30,
            peso = 61.0}
    end
}
myobj2 = { foo =
    function (a, s, st, n)
        return 0.0, 1
    end,
    boo =
    function (n)
        return 1, { nome = "Teo", idade = 60,
            peso = 73.0}
    end
}
-- cria servidores:
serv1 = luarpc.createServant (myobj1, arq_interface)

```

```

serv2 = luarpc.createServant (myobj2, arq_interface)
-- usa as infos retornadas em serv1 e serv2 para
   divulgar contato
-- (IP e porta) dos servidores
...
-- vai para o estado passivo esperar chamadas:
luarpc.waitIncoming()

```

e, por outro lado, o seguinte trecho de programa Lua deve conseguir acessar esse servidor:

```

...
local p1 = luarpc.createproxy (IP, porta1,
    arq_interface)
local p2 = luarpc.createproxy (IP, porta2,
    arq_interface)
local r, s = p1:foo(3, "alo", {nome = "Aaa", idade =
    20, peso = 55.0})
local t, p = p2:boo(10)

```

Como sugerido nesse exemplo, um parâmetro out deve ser tratado como um resultado a mais da função.

Vale a pena destacar que o cliente como o servidor conhecem o arquivo de interface. O código cliente deve tentar fazer a conversão dos argumentos que foram enviados para tipos especificados na interface, e gerar erros nos casos em que isso não é possível: por exemplo, se o programa fornece um **string** com letras onde se espera um parâmetro **double**.

O cliente deve verificar se estão sendo passados todos os parâmetros esperados pela função chamadas. A biblioteca deve tratar erros de forma educada. Por exemplo, se o cliente chamar uma função inexistente na interface do servente, nem cliente nem servidor devem “voar”.

A função **luarpc.waitIncoming** pode ser executada depois de diversas chamadas a **luarpc.createServant**, como indicado no exemplo, e deve fazer com que o processo servidor entre em um loop onde ele espera pedidos de execução de chamadas a qualquer um dos objetos serventes criados anteriormente, atende esse pedido, e volta a esperar o próximo pedido (ou seja, não há concorrência no servidor!). Provavelmente você terá que usar a chamada **select** para programá-la.

O protocolo é baseado na troca de strings ascii. Cada chamada é realizada pelo nome do método seguido da lista de parâmetros **in**. Entre o nome do método e o primeiro argumento, assim como depois de cada argumento, deve vir um fim de linha. A resposta deve conter o valor resultante seguido dos valores dos argumentos de saída, cada um em uma linha. Caso ocorra

algum erro na execução da chamada, o servidor deve responder com uma **string** iniciada com "**___ERRORPC:**", possivelmente seguida de uma descrição mais específica do erro, por exemplo, "função inexistente".

O protocolo descrito acima em linhas bastante gerais. Cabe à turma combinar entre si o protocolo exato, pois na entrega final do trabalho o cliente de cada aluno deve conseguir se comunicar com o servidor dos demais.

Vários dos objetos passados a **luarpc.createServant** podem ter a mesma interface, então não está correto distinguir qual deles deve ser chamado pela interface. Cada objeto servant deve ser associado a uma porta diferente.

Coloque as funções da biblioteca pedida em um arquivo **luarpc.lua**, e crie outros arquivos separados para **clientes** e **servidores**.

A entrega deve incluir código e um pequeno texto relatando as maiores dificuldades encontradas.

2 Executar

2.1 Bibliotecas

Para executar são necessárias algumas bibliotecas, todas elas já estão disponível no arquivo zip:

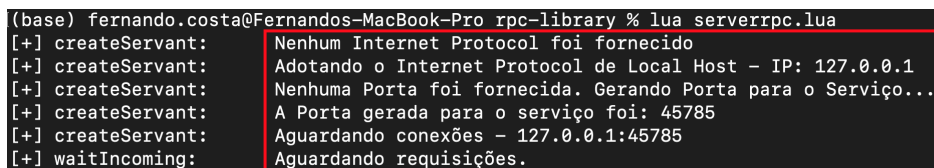
- Lua Socket
- Lua Penlight
- Lua JSON

2.2 Servidor RPC

Para inicializar o servidor, execute a seguinte instrução na linha de comando/terminal:

```
lua serverrpc.lua [IP] [PORT]
```

IP e PORT são parâmetros opcionais, quando não definidos pelo usuário, o programa irá adotar o IP de local host e gerará uma porta entre 12000 e 65535. A Figura 1 ilustra esse cenário.



```
(base) fernando.costa@Fernandos-MacBook-Pro rpc-library % lua serverrpc.lua
[+] createServant: Nenhum Internet Protocol foi fornecido
[+] createServant: Adotando o Internet Protocol de Local Host - IP: 127.0.0.1
[+] createServant: Nenhuma Porta foi fornecida. Gerando Porta para o Serviço...
[+] createServant: A Porta gerada para o serviço foi: 45785
[+] createServant: Aguardando conexões - 127.0.0.1:45785
[+] waitIncoming: Aguardando requisições.
```

Figura 1: Executando o Servidor RPC

2.3 Cliente RPC

Para executar o server, rodar na linha de comando/terminal:

```
lua clientrpc.lua [IP] [PORT]
```

IP é um parâmetro opcional, que quando não definido pelo usuário, o programa irá adotar o IP de local host. O parâmetro PORT deverá ser fornecido pelo usuário, de acordo com o valor gerado pelo servidor. As Figuras 2 e 3 ilustram esse cenário.


```
(base) fernando.costa@Fernandos-MacBook-Pro rpc-library % lua serverrpc.lua
[+] createServant: Nenhum Internet Protocol foi fornecido
[+] createServant: Adotando o Internet Protocol de Local Host - IP: 127.0.0.1
[+] createServant: Nenhuma Porta foi fornecida. Gerando Porta para o Serviço...
[+] createServant: A Porta gerada para o serviço foi: 61392
[+] createServant: Aguardando conexões - 127.0.0.1:61392
[+] waitIncoming: Aguardando requisições.
[+] decodeMSG: Mensagem decodificada.
```

Figura 2: Porta Gerada pelo Servidor RPC

```
(base) fernando.costa@Fernandos-MacBook-Pro rpc-library % lua clientrpc.lua "" 61392
[+] struct: Todos os parametros estão corretos.
[+] createProxy: Nenhum Internet Protocol foi fornecido
[+] createProxy: Conectando ao Internet Protocol de Local Host - IP: 127.0.0.1
[+] createProxy: A Porta fornecida - Porta: 61392
[+] createProxy: Conectando ao Servidor - IP: 127.0.0.1 - Porta: 61392
[+] interface: Todos os parametros estão corretos.
[+] encodeMSG: Mensagem codificada.
{"type":"REQUEST","parameters":[0,"void",{"name":"Fernando","age":60},1],"func":"foo"}
[+] decodeMSG: Mensagem decodificada.
```

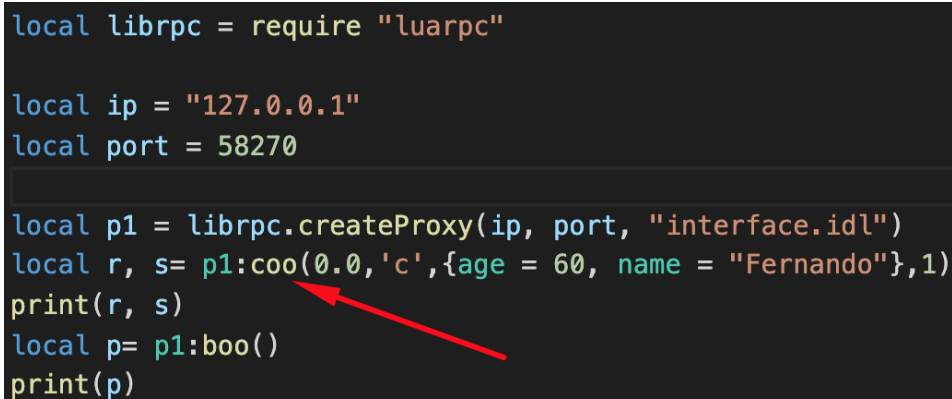
Figura 3: Cliente RPC

3 Simulações

Para validar o funcionamento de recebimento e envio de mensagem de acordo com a IDL, foram criados casos de testes.

3.1 Função Inexistente

Este caso teste é a chamada de uma função inexistente por parte do cliente. A Figura 4 ilustra o cenário em que o cliente irá chamar uma função inexistente no arquivo de interface, a função `coo`.



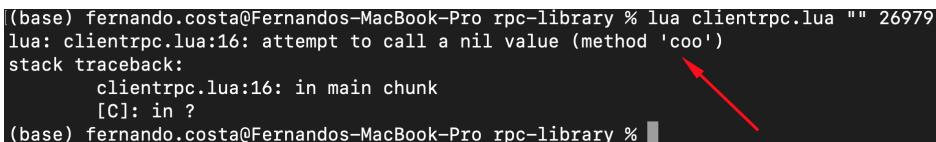
```
local librpc = require "luarpc"

local ip = "127.0.0.1"
local port = 58270

local p1 = librpc.createProxy(ip, port, "interface.idl")
local r, s= p1:coo(0.0, 'c', {age = 60, name = "Fernando"}, 1)
print(r, s)
local p= p1:boo()
print(p)
```

Figura 4: Código Cliente - Chamada a Função Inexistente - coo

O comportamento do programa para cenário não está de acordo com as especificações do enunciado do trabalho. A Figura 5 apresenta o resultado da execução de uma função inexistente.



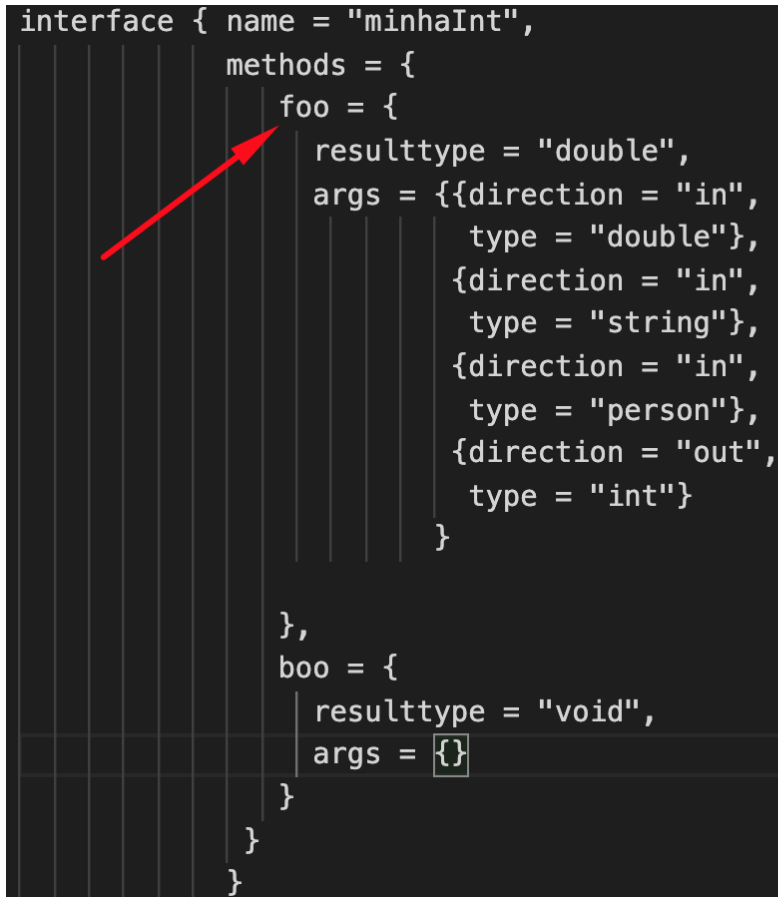
```
(base) fernando.costa@Fernandos-MacBook-Pro rpc-library % lua clientrpc.lua "" 26979
lua: clientrpc.lua:16: attempt to call a nil value (method 'coo')
stack traceback:
  clientrpc.lua:16: in main chunk
  [C]: in ?
(base) fernando.costa@Fernandos-MacBook-Pro rpc-library %
```

Figura 5: Resultado no Terminal - Caso 1

3.2 Excesso de Parâmetros

Neste cenário testaremos chamar uma função válida na interface, passando mais parâmetros do que o estipulado em sua definição. Para isso,

utilizaremos a função foo, que tem quatro parâmetros definidos em estrutura, conforme apresenta a Figura 6.



```
interface { name = "minhaInt",  
    methods = {  
        foo = {  
            resulttype = "double",  
            args = {{direction = "in",  
                    type = "double"},  
                    {direction = "in",  
                    type = "string"},  
                    {direction = "in",  
                    type = "person"},  
                    {direction = "out",  
                    type = "int"}  
            }  
        },  
        boo = {  
            resulttype = "void",  
            args = {}  
        }  
    }  
}
```

Figura 6: Definição da Função foo

A Figura 7 ilustra o código do cliente realizando a chamada com cinco parâmetros, um parâmetro além do necessário.

```

local librpc = require "luarpc"

local ip = "127.0.0.1"
local port = 58270

local p1 = librpc.createProxy(ip, port, "interface.idl")
local r, s= p1:foo(0.0,'c',{age = 60, name = "Fernando"},1, "extra")
print(r, s)
local p= p1:boo()
print(p)

```

Figura 7: Código Cliente - Parâmetro extra

Por sua vez, a Figura 8 apresenta o resultado desta chamada. Como é possível observar, o resultado mostra que a chamada não está de acordo com a definição do arquivo IDL. Após isso, o programa é finalizado.

```

(base) fernando.costa@Fernandos-MacBook-Pro rpc-library % lua clientrpc.lua
[+] interface: Numero de parametros nao esta de acordo com a IDL.
(base) fernando.costa@Fernandos-MacBook-Pro rpc-library %

```

Figura 8: Resultado Terminal - Caso 2

3.3 Falta de Parâmetros

Similar ao caso anterior, testaremos chamar uma função válida na interface, passando menos parâmetros do que o estipulado em sua definição. Para isso, utilizaremos a função foo, que tem quatro parâmetros definidos em estrutura, conforme apresenta a Figura 6.

A Figura 9 ilustra o código do cliente realizando a chamada com quatro parâmetros, um parâmetro a menos do necessário.

```

local librpc = require "luarpc"

local ip = "127.0.0.1"
local port = 58270

local p1 = librpc.createProxy(ip, port, "interface.idl")
local r, s= p1:foo(0.0, 'c', {age = 60, name = "Fernando"})
print(r, s)
local p= p1:boo(
print(p)

```

Figura 9: Código Cliente - Parâmetro Faltando

Por sua vez, a Figura 10 apresenta o resultado desta chamada. Como é possível observar, o resultado mostra que a chamada não está de acordo com a definição do arquivo IDL. Após isso, o programa é finalizado.

```

(base) fernando.costa@Fernandos-MacBook-Pro rpc-library % lua clientrpc.lua
[+] interface: Numero de parametros nao esta de acordo com a IDL.
(base) fernando.costa@Fernandos-MacBook-Pro rpc-library %

```

Figura 10: Resultado Terminal - Caso 3

3.4 Tipo de Parâmetro Incorreto

Neste caso teste faremos uma chamada a uma função válida da interface, porém passando um parâmetro de um tipo diferente do que esperado na definição da interface. Para isso, utilizaremos a função foo, que tem quatro parâmetros definidos em estrutura, conforme apresenta a Figura 6.

A Figura 11 ilustra o código do cliente realizando a chamada com quatro parâmetros, porém substituindo o primeiro parâmetro, que é esperado um **double**, por uma **string**.

```

local librpc = require "luarpc"

local ip = "127.0.0.1"
local port = 58270

local p1 = librpc.createProxy(ip, port, "interface.idl")
local r, s= p1:foo(["incorreto",'c',{age = 60, name = "Fernando"}, 1])
print(r, s)
local p= p1:boo()
print(p)

```

Figura 11: Código Cliente - Parâmetro Tipo Incorreto

A Figura 12 apresenta o resultado desta chamada. Como é possível observar, o resultado mostra que a chamada não está de acordo com a definição do arquivo IDL. Após isso, o programa é finalizado.

```

(base) fernando.costa@Fernandos-MacBook-Pro rpc-library % lua clientrpc.lua
[+] interface: Tipo de parametro nao esta de acordo com a IDL.
(base) fernando.costa@Fernandos-MacBook-Pro rpc-library %

```

Figura 12: Resultado Terminal - Caso 4

3.5 Parâmetro Especial “nil”

Neste caso teste faremos uma chamada a uma função válida da interface, porém passando um parâmetro de um tipo especial, o nil. Para isso, alteraremos a definição da função foo, a Figura 13 apresenta a definição que será utilizada.

```

interface { name = "minhaInt",
    methods = {
        foo = {
            resulttype = "double",
            args = {{direction = "in",
                    type = "double"},
                    {direction = "in",
                    type = "void"}},
            {direction = "in",
            type = "person"},
            {direction = "out",
            type = "int"}
        },
        boo = {
            resulttype = "void",
            args = {}
        }
    }
}

```

Figura 13: Nova Definição Interface

A Figura 14 ilustra o código do cliente realizando a chamada com quatro parâmetros, porém substituindo o segundo parâmetro, que é esperado um **string**, por um **void**.

```

local librpc = require "luarpc"

local ip = "127.0.0.1"
local port = 64580

local p1 = librpc.createProxy(ip, port, "interface.idl")
local r, s= p1:foo(0.0, nil, {age = 60, name = "Fernando"}, 1)
print(r, s)
local p= p1:boo()
print(p)

```

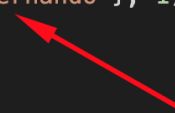


Figura 14: Código Cliente - Parâmetro “nil”

A Figura 15 apresenta o resultado desta chamada. Como é possível observar, o resultado mostra que a chamada não está de acordo com a definição do arquivo IDL. Após isso, o programa é finalizado.

```

(base) fernando.costa@Fernandos-MacBook-Pro rpc-library % lua clientrpc.lua
[+] interface: Tipo de parametro nao esta de acordo com a IDL.
(base) fernando.costa@Fernandos-MacBook-Pro rpc-library %

```

Figura 15: Resultado Terminal - Caso 5

3.6 *Struct* Com Tipos de Campos Inválidos

Para validarmos este caso de teste, precisamos alterar o arquivo IDL que contém as definições. A Figura 16 ilustra a *struct person*, que tem dois campos definidos em estrutura.


```

struct { name = "person",
        fields = {
            {name = "age",
             type = "int"},
            {name = "name",
             type = "string"},
        }
}

interface { name = "minhaInt",
            methods = {
                foo = {
                    resulttype = "double",
                    args = {{direction = "in",
                             type = "double"},
                           {direction = "in",
                             type = "void"}},
                    {direction = "in",
                     type = "person"},
                    {direction = "out",
                     type = "int"}
                },
                boo = {
                    resulttype = "void",
                    args = {}
                }
            }
}

```

Figura 16: Arquivo IDL - Definição Struct

Para validarmos esse cenário, o código do cliente irá realizar uma chamada

a função `foo`, passando uma *struct* inválida, conforme mostra a Figura 17.

```
local librpc = require "luarpc"

local ip = "127.0.0.1"
local port = 64580

local p1 = librpc.createProxy(ip, port, "interface.idl")
local r, s= p1:foo(0.0, nil, {age = 60, name = 0}, 1)
print(r, s)
local p= p1:boo()
print([p])
```

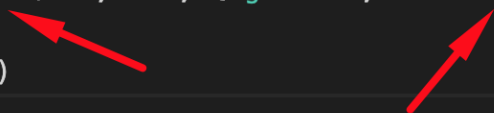


Figura 17: Código Cliente - Função `foo`

A Figura 18 mostra que o biblioteca não aceita tipo inválidos para a *struct*.

```
(base) fernando.costa@Fernandos-MacBook-Pro rpc-library % lua clientrpc.lua
[+] struct:      Tipo de parametro nao esta de acordo com a IDL.
(base) fernando.costa@Fernandos-MacBook-Pro rpc-library %
```

Figura 18: Resultado Terminal - Caso 6

4 Conclusão

Durante o projeto foram encontradas algumas dificuldades, tais como: extrair, armazenar e validar as informações do arquivo ID; e “*debuggar*” o código para encontrar os erros.

Para solucionar a questão de extrair as estruturas do arquivo IDL foram criadas duas funções: `struct` e `inteface`. Em relação ao processo de “*debuggar*”, utilizei uma biblioteca para Lua chamada Penlight Lua Libraries, permitindo visualizar o conteúdo de uma tabela e contabilizar o número de parâmetros dentro da tabela.

O código fonte do projeto pode ser encontrado em: <https://github.com/nandohdc/INF2545/tree/main/rpc-library>