Trabalho 1 - Sistemas Distribuídos

Data de entrega: 27/3

Biblioteca RPC

Construa um sistema de apoio a chamadas remotas de métodos utilizando Lua e LuaSocket. Implemente 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 (se você tiver definido uma função chamada *interface* - a sintaxe de Lua permite que uma função com um único argumento do tipo tabela seja chamada sem os parênteses, como acima). Um 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.

Observe que tanto 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).

- *atenção:* 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.

Protocolo de Comunicação

O protocolo de comunicação entre cliente e servidor deve respeitar a descrição a seguir.

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").

atenção: Esse protocolo está descrito 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.

- *atenção:* 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.

Entrega

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.