

# Relatório INF2545 - Trabalho 2 - LuaRPC

Aluno André Mazal Krauss

22/04/2019 - PUC-Rio

Professora Noemi Rodriguez

## A biblioteca

Com o presente trabalho, propõe-se a implementação de uma simples biblioteca em lua que permita *chamadas de procedimento remoto* (ou *remote procedures calls*, vide a sigla em inglês RPC). Com esta biblioteca, um script *implementador* (ou *servente*) pode disponibilizar funções para chamada remota através de um *servidor*. Com um servidor pronto, *clientes* podem então abrir um *proxy* para efetuar as chamadas remotas e obter seu retorno. Assim, a comunicação ponta a ponta acontece sempre entre: implementador  $\longleftrightarrow$  servidor  $\longleftrightarrow$  proxy  $\longleftrightarrow$  cliente.

Para tal, a biblioteca `luarpc` disponibiliza as seguintes funções:

1. `luarpc.registerServant(idl, object, [p_ip], [p_port])` - A função que deve ser usadas pelos implementadores para se registrarem junto à biblioteca `luarpc`.

Parametros:

- a. `Idl`: uma string com a especificação das funções a serem disponibilizadas
- b. `Object` - uma table cujos campos contém as funções a serem disponibilizadas e chamadas por meio da `rpc`
- c. `[P_ip]` - O IP da porta onde se deseja disponibilizar o acesso
- d. `[P_port]` - O número da porta onde se deseja disponibilizar o acesso

Caso não seja fornecido os parametros `p_ip` e `p_port`, serão usados o `localhost` e uma porta qualquer

2. `luarpc.waitIncoming()` - Indica à biblioteca que todos os implementador já estão registrados e que ela deve iniciar o ciclo de espera por clientes
3. `luarpc.createProxy(idl, ip, port)` - Tenta criar um proxy para a `idl`, conectando a um servidor do `ip` e porta dados por parametro

## Da Implementação da biblioteca

Abaixo detalho como se realiza a comunicação entre as partes que citei acima:

## Implementador $\longleftrightarrow$ Servidor

Essa comunicação é realizada através da função `registerServant`. Não são realizados checagens se o implementador está de acordo com a *idl* (se supõe que isso é parte do protocolo e não deve ser violado). A função `registerServant` apenas adiciona mais um servente que estará *escutando* por acessos. Ela faz isso criando uma função `listen()`, que será então chamada no loop principal na função `waitIncoming`. Quando a `listen` conecta com um proxy de um cliente, ela então atende, disparando a função `receive_and_reply()`.

Vale agora descrever brevemente o ciclo de escuta implementado na função `waitIncoming()`. Ela é somente um loop infinito que, para cada socket registrado, pede que ele escute por clientes. Essa escuta pela parte do socket é feita com um `timeout`, de maneira que todos estejam sempre se revezando e seja possível responder a clientes de qualquer socket sem muito tempo de espera. Tentei fazer algo mais sofisticado usando a função `select` da biblioteca `luasocket`, porém não conseguia detectar uma nova requisição de um cliente que já estava conectado, o que impossibilitou essa abordagem

## Servidor $\longleftrightarrow$ Proxy

Aqui que de fato está a comunicação entre dois processos, possivelmente em máquinas distintas. Quando um proxy é criado, ele tenta se conectar ao servidor no ip e porta passados. Se houver um servidor aberto nessa porta, vai ser aberta uma conexão, e um novo proxy-client estará registrado para atendimento. Essa conexão permanece aberta até que o cliente encerre seu processo, até que haja um erro qualquer no servidor (que só acontece se o protocolo não for mantido por alguma das partes) ou se ela for removida para dar espaço a um proxy mais novo (o máximo são 3 conexões abertas, e, na chegada de uma quarta, a conexão mais antiga será fechada para dar lugar a ela).

A comunicação de fato se usa através de passagem de *parametros* do proxy pro servidor e de *retornos* do servidor pro proxy. Ambos são de/serializados da mesma forma (usando uma combinação da biblioteca `binser` com a `mime` do `luasocket`). Vale notar que optei por serializa-los sempre dentro de uma `table`, para que fosse possível contemplar o caso de retornos vazios. Não há requisição vazias porque, mesmo sem nenhum parametro, o proxy ainda deve informar ao servidor o nome da função que deve ser executada. Após serializados, os valores da requisição/reply são enviados/recebidos através das funções `send`/`receive` da biblioteca `luasocket`.

## Proxy $\longleftrightarrow$ Cliente

O cliente é aquele que de fato deseja os valores de retorno oferecidos pelo servente, então é ele quem dispara toda a cadeia da execução remota. Depois de criar um proxy usando a função `createProxy`, ele está livre para realizar quantas chamadas quiser, sendo sempre responsável por capturar (com `pcall`) um possível erro levantado pelo proxy (porque a conexão fechou, ou por algum erro na passagem dos parametros). O papel do proxy é então, somente: avaliar se a chamada feita pelo cliente está de acordo com os parametros definidos na idl, de maneira a proteger o server de uma execução errônea; mandar a requisição ao server; ouvir a resposta; desempacotar a resposta e, por último repassá-la ao cliente. Esse último repasse é feito retornando todos os valores de retorno consecutivamente, começando pelo valor de retorno propriamente dito (que será nil, caso a função seja definida void), seguido de quaisquer parâmetros que tenham sido definidos como inout.

## Da Organização dos Arquivos

O arquivo `luarpc.lua`: implementa a biblioteca `luarpc`, ou seja, aqui está a parte mais importante do trabalho.

No arquivo `idl.txt` está a idl usada para todos os exemplos. Todos os exemplos usam o mesmo servente, que disponibiliza várias funções, implementado em `server.lua`. Há 4 clientes exemplo, cada um implementado num `.lua` diferente. Os `.luas` são denominados `clientX.lua`, (com X de 1 a 4, para cada exemplo) e o output obtido na minha máquina para a execução de cada cliente exemplo pode ser lido em `outputX.lua` (o `client4.lua` não tem um arquivo output, porque não se encaixava ao intuito do exemplo). Abaixo descrevo como usar cada cliente, e o que eles testam. Para todos eles já deve haver um servidor aberto.

## Server

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

```
lua server.lua [ip] [port]
```

IP e port são parâmetros que, opcionalmente, definem em qual ip e porta deve ser aberto o servidor. Se não forem passados, o server será aberto no localhost em uma porta qualquer, que será impressa na linha de comando.

## Cliente 1

O cliente abre um proxy, e chama n vezes seguidas a função 'inc' do servidor. Por último, imprime o retorno obtido da última função, demonstrando o valor incrementado

Testa a capacidade do servidor de atender diversas vezes seguidas à mesma conexão, sem o processo de fechar e abrir de novo.

Se N não for passado, o default é 1

uso: com o servidor aberto no ip 'ip' e na porta 'port' fazer:

```
lua client1.lua ip port [n]
```

## Cliente 2

O cliente abre um proxy, e chama várias funções que utilizam paramtros diversos, de tipos diversos, com args in e inout, funções void e sem retorno

Testa a capacidade do servidor de atender chamadas corretas de vários tipos

usage: com o servidor aberto no ip 'ip' e na porta 'port' fazer:

```
lua client1.lua ip port
```

## Cliente 3

O cliente abre três proxies na mesma porta, e utiliza os três com sucesso. Ao abrir o quarto proxy, nota-se que o primeiro foi desconectado pelo servidor

Testa o limite do servidor de atender proxies distintos na mesma porta

usage: com o servidor aberto no ip 'ip' e na porta 'port' fazer:

```
lua client1.lua ip port
```

## Cliente 4

exemplo de client 4:

O cliente abre um proxy, seta a string global com um numero aleatorio, e imprime [n] vezes seguidas a string global que está no servidor.

A ideia é que seja usada para demonstrar que é possível abrir mais de um servidor, em portas diferentes, usando a biblioteca implementada. Se forem abertos dois pares server-client4, cada cliente deve imprimir sua propria string, quando um botão for *inputado* no terminal.

Por outro lado, se você conectar um segundo client4 no mesmo servidor, será possível averiguar no primeiro client4 que a string global original foi sobrescrita.

Esse é o único exemplo para o qual eu não forneci o output obtido, já que, além de ser aleatório, o ponto de interesse só é reproduzível com dois terminais.

Se N não for passado, o default é 1

usage: com o servidor aberto no ip 'ip' e na porta 'port' fazer:

```
lua client1.lua ip port [n]
```

## Faulty Server

Para além dos arquivos e exemplos já descritos, há o faulty\_server.lua, usado para experimentar os efeitos de um objeto implementador que não respeita o protocolo delimitado pela comunicação server-proxy e pela idl. Como isso é uma quebra explícita do protocolo, que não deveria ser violado, tem efeitos adversos relativamente imprevisíveis: alguns podem ser detectados pelo servidor, que deve encerrar a comunicação com o proxy e derrubar o servidor; outros podem ser detectados pelo proxy, que então encerra a comunicação com o server defeituoso e levanta um erro; outros ainda podem não ser detectados, e propagarem uma informação faltosa até o script cliente, ou mesmo derrubar o servidor por levantar um erro. Podem ser testados com qualquer cliente acima, e resultará em erros distintos para cada um deles.