T2 – RPC

Aluno: Marcelo Costalonga Cardoso
Matricula: 1421229

Instruções:

Para executar o programa é preciso primeiro levantar o server em “server.lua”, as versões 1 e 2 estão nos arquivos “rpc_without_pool.lua” (V1) e “rpc_with_pool.lua” (V2), respectivamente. O arquivo “luarpc.lua” está com o código da V2, caso queira rodar a versão sem pool é necessário fazer um:

```
$ cp  rpc_without_pool.lua luarpc.lua
```

Há um script em bash para executar o cliente (“client.lua”) e seus testes, basta executar o comando “bash run-tests.sh” e passsar os seguintes parametros:

```
# Parameters:
#   $1 - Version (1 or 2)
#        1 = close and reconnect at each request
#        2 = maintaining connection open
#   $2 - Number of clients (must be 1, 5 or 50)
#   $3 - port 1
#   $4 - port 2
#   $5 - path to interface file
```

Ao final de cada teste será gerado um arquivo txt com os tempos para executar cada teste.
Ex: “test-V2-N1-time-results.txt” corresponde ao teste V2 (com pool de conexões) e 1 cliente

Para análise de experimento, foram considerados testes de 3 cargas, com 1 cliente (N1), 5 clientes (N5) e 50 clientes (N50). Para cada uma dessas cargas foram executados 5 testes para cada versão. Esses resultados foram ordenados pelo seu tempo total (considerando todos os clientes em cada caso) e o arquivo que continha a mediana dentre cada um desses 5 tempo totais foi o considerado para as comparações.

Funcionamento do código:

V1 – Sem pool de conexões:

- Fecha e abre conexão a cada request.
- No momento em que é feito um request como p.boo() ou p.foo() é aberta uma nova conexão e logo antes de retornar o resultado essa conexão é fechada
- No select(), dentro da “waitIncomming” só são verificados os sockets de server, pois logo após o “accept()” é feito o “receive()” da mensagem do cliente e após o término da mensagem a conexão será fechada

V2 – Com pool de conexões:

- Mantem a conexão aberta até que o limite do buffer seja ultrapassado (foi considerado um limite de 4 clientes por buffer de cada server)
- No momento em que é feito um request como p.boo() ou p.foo(), é verificado se já foi criada uma conexão anteriormente caso contrário, é criada uma nova conexão. Após isso, é feito o send(), caso receba uma mensagem de erro “closed” o cliente fica tentando se reconectar continuamente com o server, até conseguir e então é feito o send(), novamente.
- No select(), são verificados os sockets de server e de clientes:
  - Caso seja um socket de cliente, é feito somente um receive, (com certeza não é o primeiro request feito por esse cliente)
  - Caso seja um socket de server, primeiro é atualizado o buffer da pool (podendo expulsar ou não um cliente) e logo em seguida é feito o receive(), dessa forma é possível garantir que todo o cliente vai ter seu pelo menos um request processado antes de ser descartado pelo do buffer pelo server.

Em ambas as versões, as conexões só são criadas no momento em que um cliente faz um request, ou seja nunca é feito na createProxy() (para não deixar um server com uma conexão ociosa).
Além disso, quando um request é feito pelo cliente no momento em que o server dá o primeiro receive() ele fica bloqueado até terminar de processar aquele request.
E é feita também uma validação de cada pedido dos clientes, caso algo não esteja de acordo com o esperado pela interface é gerada uma mensagem informando os erros.

Testes:

Foram utilizados dois arquivos para teste: “client.lua” e “client1.lua”, ambos são iguais:

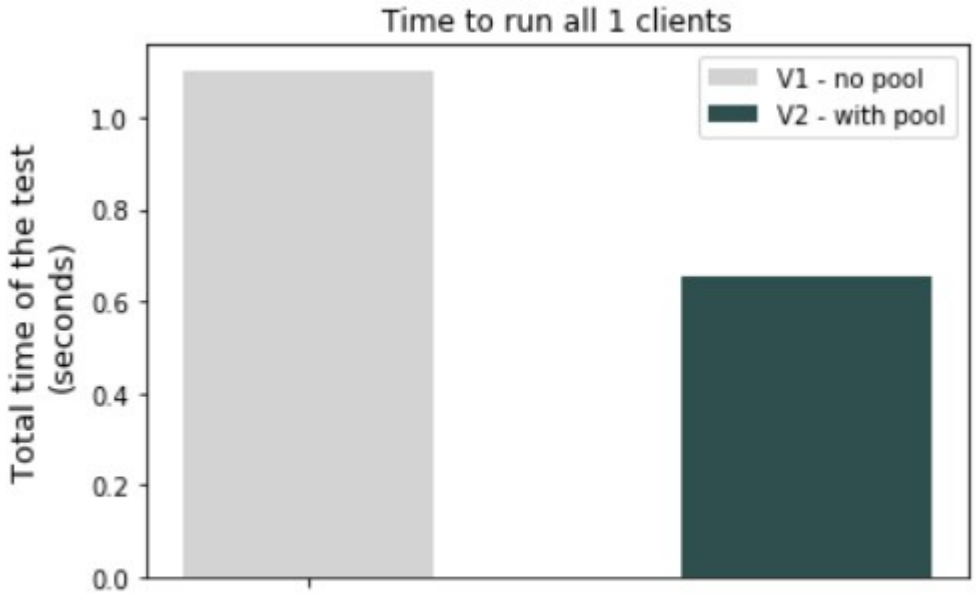
1. São criados dois proxies (p1 e p2) para dois servers diferentes (um na porta 8000 e outro na 8001)
2. O tempo do teste começa a ser marcado aqui
3. Dentro de loop, ocorrem as seguintes chamadas:
    1. p1.foo()
    2. p2.foo()
    3. p1.boo()
    4. p2.boo()
4. O tempo de término do teste ocorre logo após o fim do loop.

Após cada request é verificado se o resultado está correto, caso contrário é exibida uma mensagem de erro. Portanto num caso normal, não deve ser printada nenhuma mensagem no terminal entre requests.

A única diferença entre os arquivos é que o “client.lua” foi usado para os testes com 1 e 5 clientes, e seu loop tem 2500 iterações. E o “client1.lua” foi usado para o caso de 50 clientes, com um loop menor de 500 iterações (pois a minha máquina não suportou bem o caso de 50 clientes com um loop de 2500 e os resultados ficaram estranhos).

Análise dos Resultados:

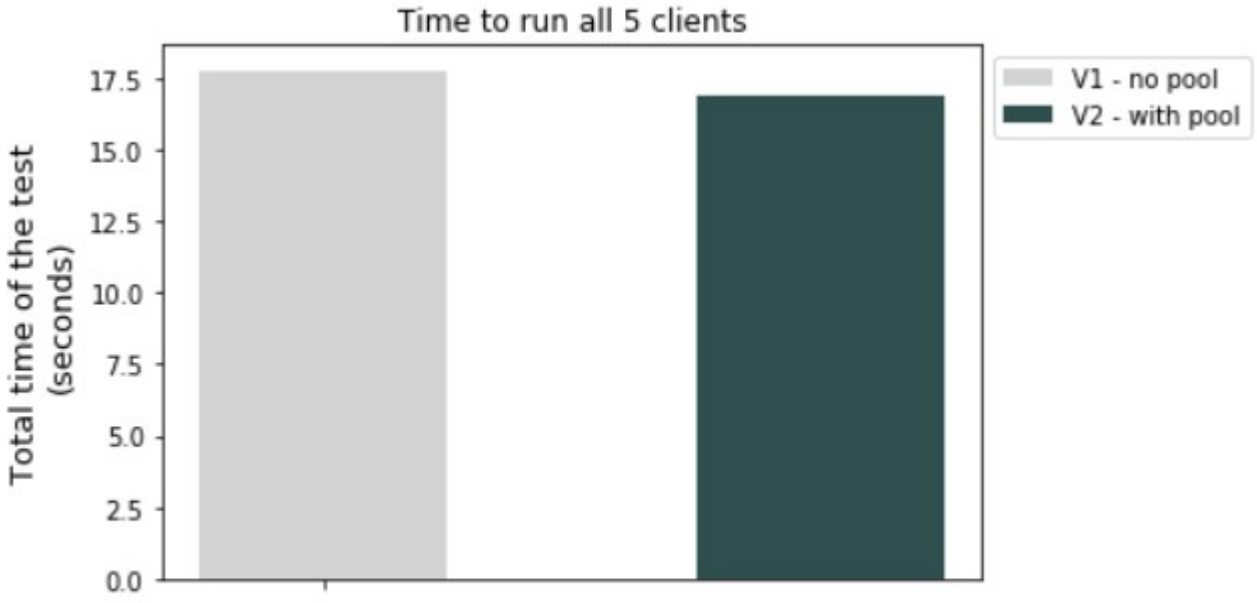
Tempo total para rodar 1 único cliente (cada cliente faz 2.5k chamadas boo() e 2.5k chamadas foo() aqui)




```
V1 - Time to run one client (median): 1.1024520397186
V2 - Time to run one client (median): 0.65135192871094
```

Aqui, como esperado, vemos que o pool de conxões é bem melhor do que ter que arcar com o overhead de abrir uma conexão a cada request.

Tempo total para rodar 5 clientes (cada cliente faz 2.5k chamadas boo() e 2.5k chamadas foo() aqui)

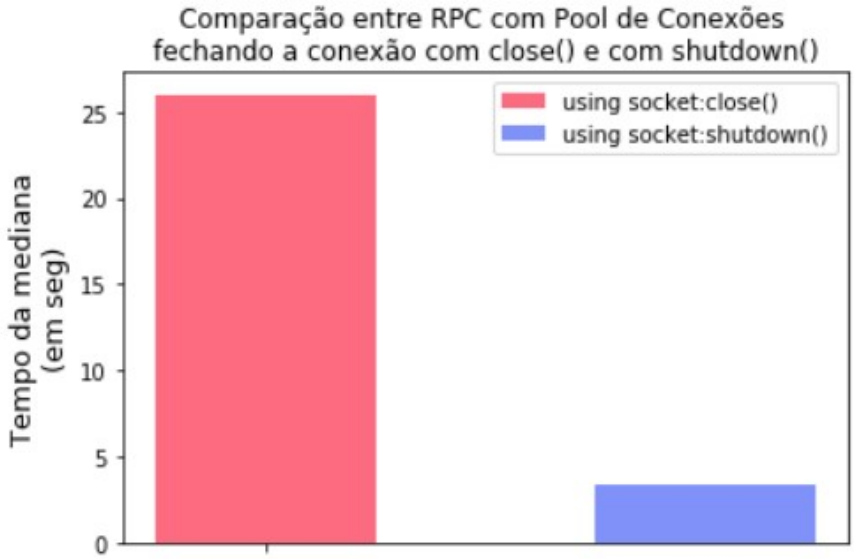



```
V1 - Time to run one client (median): 3.5563170909882
V2 - Time to run one client (median): 3.3846218585968
```

Nesse caso a vantagem entre o pool de conexões não ficou tão evidente, já que o limite do buffer é de 4 clientes por vez, o server com pool de conexões tem que arcar com o tempo de descarte de alguns clientes.

OBS: Na V2 (com pool de conxões) percebi um comportamento inesperado, inicialmente estava usando o socket:close() para fechar uma conexão de cliente, no momento de descarte e estava obtendo resultados muito piores que o da V1, após começar a utilizar socket:shutdown() ao invés de socket:close() percebi uma grande melhora. Mas não consegui perceber uma justificativa muito plausivel para isso.

Tempo total para rodar 5 clientes com pool de conexões usando close() e shutdown() para descarte (cada cliente faz 2.5k de foo() e 2.5k de boo())

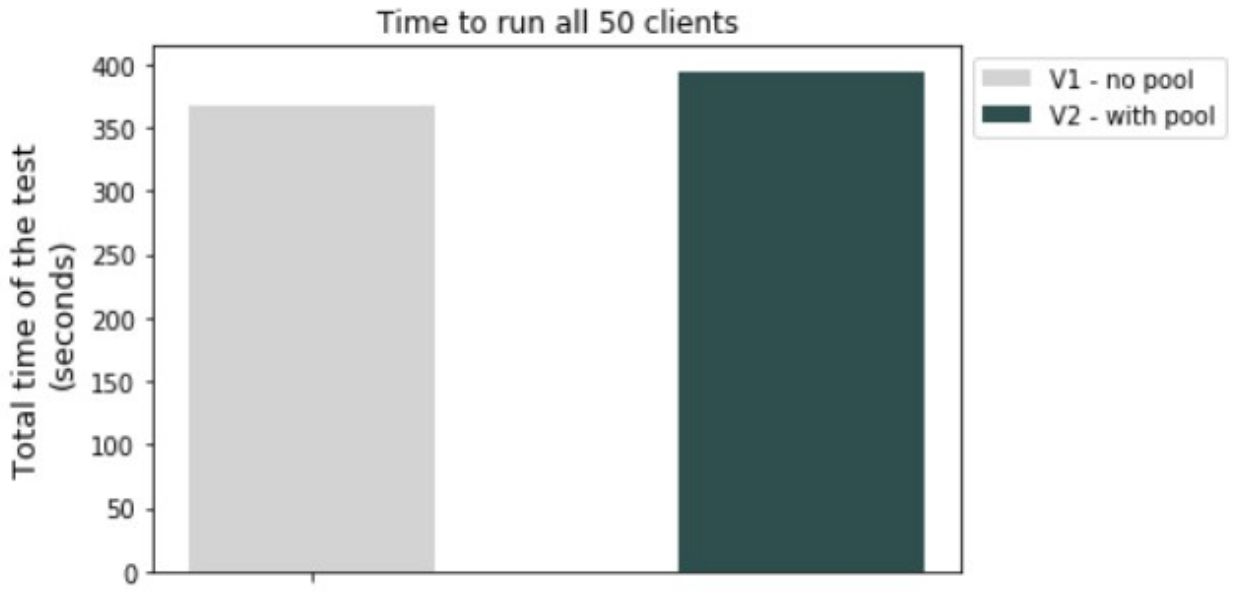



```
V1 - Time to run one client (median): 26.004794836044
V2 - Time to run one client (median): 3.3846218585968
```

Esse experimento foi executado apenas para o caso de 5 clientes com pool de conexões variando (comentando e descomentando) apenas as linhas 136 e 137 do código da “rpc_with_pool.lua” (indicadas na foto abaixo)

```
127  function luarpc.update_pool_queue(servant, client)
128    table.insert(servants_lst[servant]["pool_queue"], client)
129    local pool_size = #servants_lst[servant]["pool_queue"]
130    if pool_size > MAX_POOL_SIZE then
131      -- print("CONNECTION EXCEED!! CLOSING CLIENTS TO MAKE ROOM FOR CLIENT: ", client)
132      -- print("Pool size = ", pool_size)
133      local diff = pool_size - MAX_POOL_SIZE
134      for i=1,diff do
135        local oldest_client = table.remove(servants_lst[servant]["pool_queue"], 1) -- pop index 1
136        -- oldest_client:close() -- NOTE: WAY WORSE
137        oldest_client:shutdown() -- NOTE: SO MUCH IMPROVMENT (BUT WHY?)
138        -- print(string.format("\t >>> Client %s closed!", oldest_client))
139      end
140      -- print("\n")
141    end
142  end
```

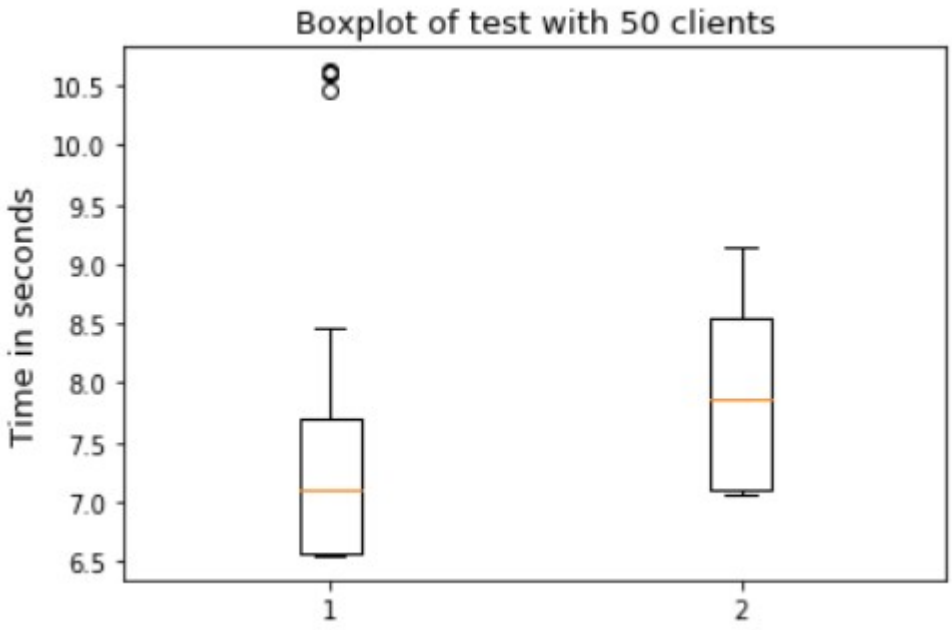
Tempo total para rodar 50 clientes (cada cliente faz 500 chamadas boo() e 500 chamadas foo() aqui)




```
V1 - Time to run one client (median): 7.09844958782195
V2 - Time to run one client (median): 7.867704987525951
```

Nesse último caso, é possível ver que o server sem pool de conexões passa a ser mais vantajoso, isso quer dizer que o overhead pelo custo abertura de conexão a cada request em V1 passa a ter um impacto menor em comparação ao custo de descarte em V2

Como temos 50 amostrars de tempo para cada versão nesse caso, podemos fazer uma análise com boxplots também:



Não sei explicar esses outliers na V1. Esperava esse cenário numa situação de estresse do server de pool de conexões. Entretanto, ainda assim aqui continua mais evidente as vantagens de se usar a V1 (sem pool de conexões) num cenário com um número de clientes, simultaneos, muito maior do que o tamanho do buffer.