Disciplina de Sistemas Distribuídos [INF 2545]
Biblioteca RPC - Trabalho 1

Rogério Carvalho Schneider

Abril 2014

Data dos testes: 14 de Abril de 2014
Professora: Noemi Rodriguez

Resumo

Relatório da implementação e testes realizados na biblioteca *luarpc* para a disciplina de Sistemas Distribuídos do Departamento de Informática da PUC-Rio. A linguagem Lua foi utilizada na implementação da biblioteca.

1 Introdução

Desenvolvemos uma biblioteca para auxílio a chamada remota de procedimento, Remote Procedure Calls (RPC), com o objetivo de enfrentar e resolver alguns dos problemas de comunicação de rede em sistemas distribuídos. A programação de servidor e cliente RPC também serviu de exercício na definição, entendimento e implementação de um protocolo de comunicação. A linguagem Lua foi escolhida para a implementação do trabalho.

A biblioteca consiste no arquivo *luarpc.lua*, que depende da definição de um arquivo de interface, normalmente chamado de *interface.lua*. A biblioteca deve poder ser reutilizada por diferentes implementações de servidor e cliente, para tanto, um contrato foi firmado entre os desenvolvedores (os alunos da disciplina) de maneira a seguir uma Application Programming Interface (API) comum a todas as implementações da biblioteca *luarpc*. Da mesma forma, foi necessário entrar em acordo quanto ao protocolo de comunicação de rede e de codificação/decodificação de mensagens a ser utilizado. A implementação desta biblioteca RPC segue as linhas gerais definidas no enunciado do trabalho[**?**].

A definição de API utilizada foi aquela do enunciado do trabalho, com os métodos *luarpc.createServant()*, *luarpc.waitIncoming()* e *luarpc.createProxy()* com exatamente o mesmo

número e tipo de parâmetros sugeridos pelo texto original. O protocolo sofreu pequenas alterações, principalmente no que diz respeito a forma de codificação e decodificação do conteúdo das mensagens enviadas.

Durante a evolução da definição do protocolo, uma lista de discussão por *e-mail* for arranjada de forma que as propostas fossem comentadas pelos participantes do consórcio que patrocinaria o desenvolvimento da biblioteca. Uma Request for Comments (RFC) foi informalmente definida na lista de discussão e após algumas rodadas de comentários o grupo chegou a um entendimento quanto ao protocolo a ser implementado.

2 Uma biblioteca RPC

Uma boa especificação de uma biblioteca de auxílio a chamada remota de procedimento funciona da seguinte maneira: A sua API sugerida deve ser clara e consistente, no sentido de que diferentes implementações possam se comportar de maneira idêntica quando estiverem se comunicando com recursos externos. Para clarificar, uma biblioteca deve manter um comportamento consistente ao interagir com o arquivo de interface e ao receber chamadas de criação e tratamento de *servant* e de *proxy* para procedimentos remotos. A biblioteca deve, portanto, fornecer um conjunto padronizado de métodos de criação e configuração de servidores e clientes, e deve interpretar de maneira correta um arquivo de interface, desde que, é claro, tenha sido escrito seguindo a API especificada.

Na prática, a proposta de especificação e implementação consistente de uma biblioteca RPC espera que os usuários/programadores possam intercambiar seus diferentes servidores e clientes, desde que todos sigam uma mesma definição de interface. Uma vez de posse da interface, um cliente qualquer pode, utilizando qualquer implementação que siga corretamente a definição da biblioteca *luarpc*, fazer requisições a um servidor qualquer, para tanto basta que ambos cliente e servidor respeitem e implementem o correto tratamento do arquivo de interface.

Um *servant* é um servidor instanciado pela biblioteca de RPC, o qual atende a requisições em rede e repassa as chamadas para a implementação local dos métodos. A biblioteca

trata de todos os detalhes da comunicação de rede, e o programador de um servidor precisa apenas se preocupar com a implementação dos métodos que deseja expôr via RPC, assim como precisa disponibilizar um arquivo de interface equivalente.

Um *proxy* para métodos remotos é o lado cliente, também disponibilizado pela biblioteca. Trata-se de uma visão local do cliente, um objeto, que não tem acesso direto à implementação dos métodos, acesso este que se dá remotamente com o uso da comunicação em rede com o servidor. Da mesma forma que no servidor, toda a comunicação, protocolo, codificação de mensagens e demais detalhes são transparentes para o programador. Basta que utilize no cliente um arquivo de interface equivalente e, sem ter conhecimento local prévio da implementação, o cliente pode acessar os métodos remotos.

Dadas as definições de *servant* e *proxy* é interessante observar que o cliente não precisa ser alterado para que uma modificação na implementação dos métodos seja feita. Como os métodos são remotos, a única alteração de código se daria no servidor, mantendo o cliente agnóstico à implementação. Não há, portanto, necessidade de alteração local de código no cliente para que o novo código do método tenha efeito. Claro que para não haver quebras no funcionamento o contrato deve ser mantido, a API e o protocolo de rede devem ser preservados e a interface não deve ser alterada.

2.1 O protocolo

Particularmente, a regra definida para o protocolo e codificação de mensagens neste trabalho foi a seguinte:

- Nomes de métodos chamados pelo cliente são enviados sem delimitadores;
- Parâmetros do tipo *char* ou *string* são enviados com delimitadores *"..."* envolvendo o texto transmitido;
- Parâmetros do tipo *double* são enviados sem transformação.
- Parâmetros do tipo *void* são transformados em *nil* e são enviados com delimitadores *"..."* envolvendo o texto transmitido. Resulta no envio de *"nil"*;

Ao transmitir conteúdo é necessário que ele seja serializado, codificando conforme o seguinte acordo, no transmissor:

- Usar sequência de escape para \, transformando em \\;
- Usar sequência de escape para $\backslash n$, transformando em $\backslash\backslash n$;
- Usar sequência de escape para ", transformando em \";
- O processo inverso, de decodificação, deve ser feito no receptor.

Os tipos válidos *char*, *string*, *double* e *void* tem o seguinte protocolo de codificação/decodificação:

- *char*: restrito a um caractere
  codificação no transmissor: envolto em *"..."*
  exemplo: *"a"*
  decodificação no receptor: remove *"..."*
  exemplo: *a*
- *string*: sem restrição
  codificação no transmissor: envolto em *"..."*
  exemplo: *"abc"*
  decodificação no receptor: remove *"..."*
  exemplo: *abc*
- *double*: apenas números inteiros ou de ponto flutuante
  codificação no transmissor: não há
  exemplo: *3.1415*
  decodificação no receptor: não há
  exemplo: *3.1415*
- *void*: transformado em nil, envolto em *"..."*
  codificação no transmissor: nil e *"..."*
  exemplo: *"nil"*

decodificação no receptor: remove *"..."*

exemplo: *nil*

- Todos os tipos devem ser enviados em apenas uma chamada de *socket:send()* e recebidos em apenas uma chamada *socket:receive()* e, portanto, o correto escape de *string* com quebras de linha ($\backslash n$) é muito importante.

O papel do receptor é desfazer a codificação aplicada, seguindo a sequência inversa de passos, de forma a obter valor e tipo originais transmitidos na mensagem. Uma saída adotada por alguns desenvolvedores para fazer o escape inverso de *string*, por exemplo, foi o uso de uma chamada "*loadstring("return ".. content)*"no conteúdo recebido, o que faz com que alguns escapes sejam decodificados transparentemente, evitando a necessidade de tratar manualmente, por exemplo, a remoção de delimitadores *"..."* no início e fim da linha recebida. Da mesma forma as contra barras de sequência de escape também são removidas automaticamente. De forma contrária, como outras implementações sugeriram, utilizar esse tipo de artifício pode facilitar a vida do programador mas ao mesmo tempo introduz uma falha de segurança que permite que uma mensagem trocada entre cliente e servidor, quando decodificada, tenha seu *payload* executado como código Lua, abrindo uma brecha para execução remota de código não desejado (alheio ao definido na interface). Para evitar este problema, foi utilizada a alternativa de tratamento manual do conteúdo, fazendo uso de técnica de *search and replace* dos caracteres codificados $\backslash"$, $\backslash\backslash n$ e delimitadores *"..."*. As funções *string.gsub()* e *string.sub(2,-2)* foram utilizadas para decodificar $\backslash"$ em *"*, $\backslash\backslash n$ em $\backslash n$ e remover os delimitadores *"..."*.

Ao perceber um erro no lado do cliente ou do servidor, o protocolo deve enviar uma mensagem iniciada por "___ERRORPC:"contendo, possivelmente, a causa do erro. Em alguns casos especiais de erro no modo de uso, a biblioteca deve tentar evitar o envio da mensagem de erro e deve procurar corrigir o número e tipo de parâmetros chamados no cliente ou respondidos pelo servidor. Para tanto, a biblioteca utiliza valores default como alternativa à falta de parâmetros nas chamadas do cliente e nas respostas do servidor. O objetivo é prover uma construção o mais limpa possível da sequência de mensagens a

trafegar no canal de comunicação de rede, tentando sempre respeitar o protocolo definido quanto a tipos, quantidade, codificação e sequência de parâmetros esperados no transporte entre cliente e servidor, ida e volta.

3 Testes de performance realizados

Foram executadas baterias de teste segundo a sugestão do enunciado do trabalho. Para cada configuração de cenário segue abaixo um parecer e algumas informações adicionais. Um cliente especial foi criado para a execução dos testes. Este cliente tem capacidade de medir o tempo que leva para executar uma série de chamadas em sequência e também consegue medir o quanto consumiu de tráfego de rede para a série de chamadas. Em todas as baterias foram executadas 10000 chamadas a cada método. Os métodos chamados no laço de teste também foram especialmente implementados e escolhidos para terem diferentes comportamentos quanto ao custo de tráfego de rede (mensagens pequenas, mensagens grandes) e custo de processamento (tempo de CPU). Também foram realizadas medições isoladas para determinar o custo de tempo imposto pela biblioteca *luarpc* e o custo de tempo imposto pela implementação presente no cliente e no servidor.

3.1 Tamanho de string muito pequeno e muito grande

Neste teste foi utilizado o método chamado *min* 1, que recebe um *string* e devolve apenas um *double*. Este perfil de teste demonstra o impacto do aumento do tráfego de dados no desempenho da aplicação.

Listing 1: Interface min

```
min = {
  resulttype = "double",
  args = {
    {direction = "in", type = "string"},
  },
},
```

Testes de cliente e servidor				
	Conexão	Enviados	Recebidos	Tempo
String 1B	Persistente	1120KB	1060KB	6s
String 10KB	Persistente	104MB	1060KB	34s
String 1B	Close	1120KB	1060KB	8s
String 10KB	Close	104MB	1060KB	34s
Serialização	Persistente	21MB	1060KB	16s
Serialização / deserialização	Persistente	21MB	1060KB	19s
Serialização	Close	21MB	1060KB	20s
Serialização / deserialização	Close	21MB	1060KB	21s
Serialização	Local	-	-	8s
Deserialização	Local	-	-	2s
Serialização / deserialização	Local	-	-	10s

Tabela 1: Tabela com custos de execução para 10000 chamadas em *loop*

Como resultado obtivemos um total de 1120KB enviados e 1060KB recebidos pelo cliente com um tempo total de duração de 6 segundos para 10000 requisições. O *string* enviado era mínimo no primeiro teste, apenas um *byte*.

No segundo teste, usando o mesmo método mas agora com um *string* muito grande, como sugerido no roteiro de testes do enunciado do trabalho, obtivemos um total de 104MB enviados e 1060KB recebidos pelo cliente com um tempo total de duração de 34 segundos para 10000 requisições. O *string* enviado neste teste era grande, continha 10KB. Os dados de performance se encontram na tabela 1.

3.2 Conexão persistente contra conexão sem reuso

Foram realizados dois outros testes interessantes, que demonstram o custo da conexão do cliente com o servidor. O método utilizado também foi o *min* 1 por oferecer duas visões

diferentes, uma delas com pouco tráfego de dados e outra com grande tráfego de dados (e portanto com conexões que duram mais tempo, até que os dados todos sejam trafegados de um lado para o outro).

O perfil mencionado no teste de *string* pequeno e grande, na seção 3.1, foi realizado usando conexão persistente e tinha duração de 6 segundos para *string* pequeno e 34 segundos para *string* grande. O mesmo perfil de teste aplicado ao servidor que fecha a conexão do cliente assim que ele termina de ser atendido fez com que o tempo total de atendimento de 6 segundos passasse para 8 segundos com o *string* pequeno. Para o *string* grande não houve modificação no tempo, o que pode ser explicado pela menor importância do custo de conexão perto do custo de tráfego de rede para o perfil de *string* grande. Para o perfil de *string* pequeno é possível perceber que o tempo de conexão representa uma parte significativa do tempo total da execução do teste. Os dados de performance se encontram na tabela 1.

3.3 Serializando e deserializando uma tabela grande

Outro perfil executado foi o sugerido de serializar uma tabela com 100 *doubles*. A serialização transforma uma tabela em um *string* e envia este *string* para o servidor em uma chamada de tipo nativo simples, chamada aqui de *tbl* 2, e que recebe um *string* e devolve um *double*. Para este teste foram usados dois servidores, um que apenas responde "0"e que não faz deserialização da tabela, e outro que responde "1"logo após fazer a deserialização da tabela.

Foi possível observar que neste perfil não houve muita diferença entre o tempo do servidor que deserializa a tabela e o servidor que não trata da decodificação da tabela.

Como resultado obtivemos um total de 21MB enviados e 1060KB recebidos pelo cliente com um tempo total de duração de 16 segundos sem deserialização e 19 segundos com deserialização no lado do servidor para 10000 requisições.

Listing 2: Interface tbl

```
tbl = {
```

```
  resulttype = "double",
  args = {
    {direction = "in", type = "string"},
  },
},
```

Ainda sobre a tabela, medimos a execução isolada de serialização e deserialização sem o envolvimento de cliente e servidor RPC. As chamadas em laço local levaram 8 segundos para serializar e 2 segundos para deserializar a tabela com 100 doubles em 10000 iterações, totalizando 10 segundos de tempo total de execução local. Os dados de performance se encontram na tabela 1.

3.4 Diferente número de clientes para pool de conexões

Um teste final foi executado configurando um servidor para atender a até 3 clientes com conexão persistente simultânea. O *pool* (a fila circular) do servidor foi configurada com tamanho 3.

Em um primeiro perfil apenas um cliente se conectou ao servidor, sem gerar concorrência com outros clientes. Neste caso o atendimento foi satisfatório para o método *foo* 3, levando apenas 7 segundos para executar 10000 operações de soma de dois *doubles* e de devolução da resposta em conjunto com um *string* pequeno.

Listing 3: Interface foo

```
foo = {
  resulttype = "double",
  args = {
    {direction = "in", type = "double"},
    {direction = "in", type = "double"},
    {direction = "out", type = "string"},
  },
},
```

Número variado de clientes e pool fixo em 3 no servidor		
	Conexão	Tempo
1 cliente	Persistente	7s
3 clientes	Persistente	16s
10 clientes	Persistente	N/A

Tabela 2: Tabela com custos de conexão para 10000 chamadas em *loop*

Na segunda execução, com 3 clientes, a concorrência já causou impacto no tempo de execução, mas mesmo assim todos os clientes ainda estavam podendo manter as suas conexões abertas o tempo todo, sem necessidade de reconexão. Neste perfil o tempo subiu para 16 segundos em cada cliente.

O último perfil sugerido, com 10 clientes e *pool* de 3 conexões, para forçar a reconexão sequencial de todos de tempos em tempos, não pode ser executado até o final. A causa é um provável erro na implementação da biblioteca *luarpc*. O consumo de memória, possivelmente devido ao grande número de reconexões, impediu que o perfil completasse a execução antes de travar o *hardware* dos testes. Mais detalhes sobre o problema são informados na seção 3.5. A suspeita é que, de alguma forma, o *garbage collector* ficou impedido de coletar os *sockets* antigos. Possivelmente, devido a maneira de manter uma lista de *sockets* em uma tabela (uma referência, portanto), a memória ocupada pelos *sockets* e por suas entradas na tabela de controle não deve estar sendo corretamente desreferenciada, impedindo que as entradas sejam coletadas para liberação de memória.

O mesmo cliente foi usado em todos os testes. A implementação do cliente e do servidor é resistente a desconexão da outra ponta e pode tratar da reconexão quando necessário. O servidor tem o tamanho do *pool* configurável por linha de comando 4, no *start* do processo, e quando configurado para *pool* de tamanho 0 ou menor passa a atuar sem persistência de conexão, forçando o fechamento após atender cada requisição. Os dados de performance se encontram na tabela 3.

Listing 4: Linha de comando

```
$ ./rpc_server.lua
Usage: ./rpc_server.lua <iface_file> [port1] [port2] [pool_size]
```

3.5 Gargalos identificados

Na configuração de teste que determina o uso de mais clientes do que o *pool* de conexões do servidor comporta, com valores sugeridos de 3 conexões para o *pool* e 10 clientes, foi possível observar um comportamento que evitou a execução da bateria de testes até o final. Na implementação atual do *servant*, que é a parte da biblioteca *luarpc* que instancia um servidor, deve existir algum erro de programação ainda não identificado que faz com que o uso de memória do servidor cresça rapidamente, impedindo a execução da bateria até o final.

Aparentemente a memória não está sendo liberada após o fechamento e descarte dos *sockets* antigos. Talvez a manipulação da tabela que memoriza a lista de clientes não esteja liberando corretamente as entradas antigas para que o *garbage collector* possa de fato fazer a limpeza das mesmas. Esta tabela que memoriza as conexões dos clientes dentro do intervalo definido para o *pool* de conexões funciona como uma lista circular limitada, a qual tem o seu limite definido pelo tamanho do *pool* e, ao atingir o limite, tem as entradas excedentes (e mais antigas) descartadas. Logo antes do descarte de uma entrada uma chamada *socket:close()* é feita na conexão.

O problema fica mais evidente quando o servidor é configurado para não persistir conexão e sempre desconectar o cliente assim que o pedido for atendido. Isto dá a entender que o ciclo completo e repetido nas listas circulares de conexões consome, de alguma forma, bastante memória, a qual não é reclamada em tempo pelo *garbage collector*.

Outro gargalo interessante foi identificado no laço principal de execução do servidor, no que diz respeito à configuração de *socket:settimeout()* nos *sockets* para as *syscalls* *socket:accept()*, *socket:select()* e *socket:receive()*. No laço principal do servidor são feitas chamadas de *socket:accept()* a cada iteração para aceitar conexões de novos clientes, bem como são feitas chamadas *socket:select()* para determinar o conjunto de *sockets* ativos de

clientes que está pronto para ser consumido (lido) sem travar. A primeira versão do servidor implementada na biblioteca *luarpc* fazia as chamadas mencionadas a cada iteração do laço principal e tinha um *timeout* configurado de 100ms para cada chamada. O tempo de 100ms pode parecer pouco, mas quando somado a cada iteração do laço principal, ele tem um efeito muito grande na performance geral do servidor. Como comparação de desempenho, vale notar que usando valores baixos de *timeout* o servidor conseguia atender até 3 requisições por segundo. Na versão final, sem *timeout* configurado (ou seja, sem bloqueio) a capacidade de atendimento do servidor passou para 2000 requisições por segundo. A principal mudança foi fazer um *select* nos *sockets* dos servidores antes do chamar *accept*, assim a chamada de *accept* só será feita se já houver algum cliente interessado em abrir nova conexão com o servidor, caso contrário o servidor nem precisa perder tempo fazendo a chamada de *accept*. Este *select* a mais e o uso de *timeout* zero para todas as chamadas de rede do sistema fizeram com que o servidor ficasse bem mais interessante do ponto de vista da performance, mantendo a estabilidade. Para as etapas de envio e consumo de dados (*send* e *receive*) ainda existe um *timeout* grande configurado, de 10 segundos, para permitir interação humana em console de depuração (*telnet*). A tabela **??** mostra um comparativo de vazão de requisições por segundo (*throughput*) para cada cenário, com e sem *timeout*.

As chamadas com *timeout* e em modo não persistente são mais rápidas do que as persistentes pois elas geram evento mais rapidamente no *select* e *accept* de entrada de conexão, consumindo portanto menos tempo de espera. O evento de nova conexão ocorre antes de vencer o *timeout*, por isso é mais rápido do que o cenário de conexão persistente. Interessante observar que normalmente se poderia pensar que um cenário de conexão persistente sempre deveria ser mais rápido do que um cenário onde as conexões são fechadas a cada iteração.

Vazão de requisições		
	Conexão	Req/s
Sem timeout	Persistente	2000
Sem timeout	Close	1400
accept(0.1)	Persistente	2000
accept(0.1)	Close	1400
select(0.1)	Persistente	3
select(0.1)	Close	5
accept(0.1) e select(0.1)	Persistente	3
accept(0.1) e select(0.1)	Close	5

Tabela 3: Tabela com *throughput* para chamadas em *loop* ao método *min*

4 Melhorias futuras

Para ficar mais idiomática a implementação em Lua, seria interessante substituir no código o trecho que trata de forma mais tradicional as listas circulares por algo mais poderoso fornecido pela linguagem Lua. A proposta é alterar o tratamento de *pool* de conexões 5 utilizando uma *metatable* [**?**] que pode, no momento da inserção de um novo *socket* de cliente na tabela de conexões, avaliar se o limite de conexões foi atingido e poderia então liberar as conexões mais antigas. A implementação atual tem exatamente o mesmo efeito, mas não usa totalmente o idioma da linguagem Lua. A sugestão é utilizar o *metamethod* *__newindex* para tratar novas entradas em forma de lista circular na tabela que agrupa os clientes de um servidor.

Listing 5: Pool de conexões

```
-- Pool size limit.
if #servant.client_list > servant.pool_size then
  while #servant.client_list > servant.pool_size do
    old_client = table.remove(servant.client_list, 1)
```

```
      old_client:close()
    end
end
```

Para tentar resolver o problema da utilização de memória no servidor poderia ser investigada a utilização de *weak tables* [**?**]. Com este recurso o funcionamento do *garbage collector* pode ser viabilizado, fazendo com que a tabela mantenha referências fracas aos objetos (*sockets*) agrupados por ela, de maneira que estes objetos poderiam ser liberados com maior probabilidade.

5 Conclusão

Este trabalho proporcionou uma boa oportunidade de aprimorar o conhecimento da linguagem Lua. A escolha por esta linguagem para a implementação do trabalho se deu por apresentar bom suporte a comunicação em rede e por ter uma curva pequena de aprendizado, afinal trata-se de uma linguagem enxuta. Apesar de enxuta, ela oferece a possibilidade de construções elegantes e poderosas com recursos como *metatables* e *metamethods*. Ainda, por ser tratar de uma linguagem dinâmica ela permite construções adaptáveis em tempo de execução, o que é ideal quando se precisa obter flexibilidade na implementação.

Foi possível entender como pode ser difícil definir um padrão de protocolo e ter convergência rápida entre diferentes grupos de desenvolvimento. Entrar em acordo é um processo que pode levar tempo de discussão até que esteja satisfatório para todos os envolvidos.

Também entendemos que pequenas escolhas de projeto podem fazer grande diferença no resultado da criação de um servidor. Pequenos detalhes como um vazamento de memória ou um *timeout* mal configurado podem render um servidor que não atende a requisitos práticos de performance e estabilidade.

Referências

[1] RODRIGUEZ, N.. **Trabalho 1 - sistemas distribuídos - biblioteca rpc**. `http://www.inf.puc-rio.br/~noemi/sd-14/trab1.html`, 2014. [Online; acessado em 15-Abril-2014].

[2] IERUSALIMSCHY, R.. **Programming in lua**. `http://www.lua.org/pil/13.html`, 2004. [Online; acessado em 15-Abril-2014].

[3] IERUSALIMSCHY, R.. **Programming in lua**. `http://www.lua.org/pil/17.html#weaktables`, 2004. [Online; acessado em 15-Abril-2014].