Trabalho 1 - Redes de Computadores 2023/2 - Turma 2

Lucas de ALmeida Abreu Faria
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Brasil
lucasaafaria@gmail.com

***Abstract*—This document is a report of the first practical project of the computer networks subject from Universidade de Brasília, which had the goal of implementing an RPC (Remote Procedure Call) using native sockets. This goal was achieved by developing a client that implements function stubs which call remote functions from a server. The remote functions are executed at the server and return a response to the client. A messaging protocol was created to regulate the communication between client and server.**

***Index Terms*—RPC, sockets, computer networks**

I. Introdução

Este relatório visa documentar o desenvolvimento do primeiro projeto prático da disciplina de Redes de Computadores da Universidade de Brasília, ministrada pelo Professor Gabriel Ferreira. O projeto se trata da criação de uma aplicação que presta e consome serviços através de *sockets* nativos dos sistemas operacionais. Como parte do projeto, desenvolveu-se também o próprio protocolo de comunicação entre cliente e servidor, especificando o tamanho, formato, conteúdo e sequência de troca de mensagens.

O tipo de aplicação escolhida pelo autor para implementar esses conceitos foi a RPC (*Remote Procedure Call* na linguagem Python, na qual existe um servidor capaz de implementar funções que serão chamadas remotamente por um cliente. A aplicação desenvolvida precisa atender aos seguintes critérios:

1) O cliente é capaz de listar as funções oferecidas pelo servidor;
2) O cliente implementa funções stub locais que reproduzem a assinatura das funções remotas;
3) Cada função stub do cliente encapsula os parâmetros passados em uma mensagem RPC, que é enviada ao servidor quando essa função é chamada durante a execução do programa;
4) O servidor escuta por mensagens de RPC, e ao receber uma mensagem válida, executa a função solicitada localmente e retorna ao cliente o resultado da execução;
5) O cliente recebe o resultado da RPC e retoma a execução do programa local sem realizar nenhuma operação de rede ou captura de exceções fora das funções stub;
6) Caso o servidor não responda num intervalo de tempo especificado, o cliente deve retransm;itir a requisição de RPC e aguardar pela resposta;
7) Caso o cliente receba duas respostas para uma mesma requisição de RPC, deverá escolher o primeiro valor recebido.

Para exemplificar o funcionamento desses requisitos, o servidor implementa 4 funções simples de cálculos: adição(*add*), subtração(*sub*), multiplicação(*mult*) e divisão(*div*). O cliente é capaz de chamar essas funções remotamente e receber a resposta do servidor, possibilitando ao usuário do programa controlar esse fluxo.

II. Protocolo de Mensagens

O protocolo de mensagens estabelecido entre o cliente e o servidor é baseado na troca de mensagens estruturadas em JSON (JavaScript Object Notation) e na utilização de um *length prefix* para garantir a integridade e a interpretação correta das mensagens transmitidas. Esta abordagem permite a comunicação eficiente e confiável entre as partes envolvidas no sistema RPC.

A. Tamanho das Mensagens

As mensagens trocadas possuem tamanho variável. Contudo, um *length prefix* de 4 bytes é adicionado ao início de toda mensagem, informando o tamanho em bytes da mensagem em questão. Por este fato, o tamanho máximo do corpo de uma mensagem que pode ser enviada ou recebida é de $2^{32} - 1 = 4,294,967,295$ bytes. Esse prefixo é adicionado pela função SEND_MESSAGE [Fig. 1], que recebe a mensagem JSON a ser enviada:

```python
# prefix the message with its length and send it to the server
def send_message(self, message):
    message_length = len(message).to_bytes(LENGTH_PREFIX_SIZE, byteorder='big')
    self.__sock.sendall(message_length + message)
```

Fig. 1. Função no cliente que envia a mensagem

Há também um limite máximo de 1024 bytes para serem recebidos por vez. Então, caso a mensagem possua um tamanho menor do que 1024 bytes, o socket escuta pelos próximos N bytes, onde N é o tamanho da mensagem estipulado no prefixo. Caso a mensagem possua pelo menos 1024 bytes, quebra-se então o recebimento em *chunks* de 1024 bytes cada um, que posteriormente são concatenados para formar a mensagem completa. Para receber uma mensagem, o socket

é instruído a escutar pelos primeiros 16 bits para identificar o tamanho da mensagem que será lida. Isso é feito pela função RECEIVE_MESSAGE [Fig. 2]:

```python
# deal with length prefix and break large messages into chunks of BUFFER_SIZE bytes maximum
def receive_message(self):
    length_prefix = self.__sock.recv(LENGTH_PREFIX_SIZE)
    if not length_prefix:
        return None

    # unpack the length prefix to determine the message size
    message_size = int.from_bytes(length_prefix, byteorder='big')

    # receive the complete message based on the determined size
    received_message = b''
    while len(received_message) < message_size:
        chunk = self.__sock.recv(min(message_size - len(received_message), BUFFER_SIZE))
        if not chunk:
            return None  # connection closed unexpectedly while reading data
        received_message += chunk

    return received_message
```

Fig. 2. Função no cliente que recebe a mensagem

B. Formato e Conteúdo das Mensagens

Como mencionado anteriormente, as mensagens devem ser necessariamente formadas por um prefixo de 4 bytes, seguido por uma mensagem em formato JSON, que por sua vez precisa ter chaves específicas para ser aceita pelo cliente ou pelo servidor.

1) Cliente: Do lado do cliente, os campos obrigatórios (mensagem é recusada pelo servidor caso qualquer campo esteja faltando) para as mensagens enviadas são:

- **sequenceNumber**: Identifica a ordem das mensagens para permitir a correspondência entre as solicitações e respostas. A primeira mensagem possui *sequenceNumber* igual 0, e este número é incrementado em 1 a cada mensagem enviada subsequentemente.
- **methodName**: Nome do método remoto a ser invocado no servidor.
- **args**: Argumentos a serem passados para o método remoto.
- **kwargs**: Argumentos do tipo "*key word*" a serem passados para o método remoto.

```python
# build the message based on the protocol standards
def build_request_data(self, method_name, args, kwargs):
    request_data = {
        "sequenceNumber": self.__seq,
        "methodName": method_name,
        "args": args,
        "kwargs": kwargs
    }
    self.__seq += 1

    return json.dumps(request_data).encode()
```

Fig. 3. Função no cliente que constrói o JSON

2) Servidor: Já do lado do servidor, caso seja possível interpretar corretamente a mensagem do cliente e executar localmente a função solicitada, ele deve enviar a resposta ao cliente com as seguintes chaves:

- **sequenceNumber**: Deve ser exatamente o mesmo número recebido na *request* do cliente.
- **result**: Resultado da execução da função solicitada.

Caso ocorra algum erro no processo de recebimento/interpretação da mensagem ou da execução da função, o servidor deve retornar a resposta ao cliente com no formato:

- **sequenceNumber**: Deve ser exatamente o mesmo número recebido na *request* do cliente.
- **error**: Contém a mensagem indicando qual erro ocorreu

```python
# build the successfull response message that will be returned to the client
def build_response(self, sequence_number, result):
    response = {
        "sequenceNumber": sequence_number,
        "result": result
    }
    print('Successfull response built: ', json.dumps(response))
    return json.dumps(response).encode()

# build the response message reporting an error during the server execution
def build_error_response(self, sequence_number, error_message):
    response = {
        "sequenceNumber": sequence_number,
        "error": error_message
    }
    print('Error response built: ', json.dumps(response))
    return json.dumps(response).encode()
```

Fig. 4. Funções no servidor que constroem a resposta

C. Ordem das Mensagens

A ordem das mensagens é bem simples. Após ser inicializado, o servidor instrui seu socket a escutar por requisições.

A conexão deve ser iniciada pelo cliente, que envia uma requisição ao servidor com um determinado *sequence number*. O servidor processa essa requisição e, caso a mensagem esteja fora do formato estabelecido, devolve uma mensagem de erro com o *sequence number* igual a -1. Caso contrário, envia uma resposta ao cliente com o mesmo *sequence number* recebido, e salva qual *sequence number* é esperado daquele cliente na próxima requisição.

Se o cliente recebe uma resposta fora do padrão ou com *sequence number* diferente da requisição enviada, ele descarta essa mensagem e envia a requisição novamente. Caso passadas 3 tentativas de requisição sem sucesso, o cliente encerra a conexão e a execução do programa por múltiplas falhas consecutivas.

Se a resposta recebida pelo cliente é válida, ele interpreta essa mensagem e segue com a execução do programa. Para a requisição seguinte, o número de sequência é incrementado em 1.

III. Listagem de Funções Disponíveis

A. Registro de Funções

O servidor possui um método para registrar funções à sua lista de funções a serem disponibilizadas para chamadas remotas, que fica salva num atributo da instância do servidor. Após inicializar o servidor, o programa registra as 4 funções (add, sub, mult e div).

No início da execução do cliente, este solicita ao servidor a lista de funções disponíveis:

```python
# request to the server the list of available remote methods
def get_available_methods(self):
    request_data = self.build_request_data('get_available_methods', [], {})
    response = self.send_request_with_timeout(request_data)
    return response['result']
```

Fig. 5. Função no cliente que solicita métodos disponíveis

O servidor então recebe essa requisição, e responde com a lista salva em seu atributo *_methods*. O cliente recebe de volta essa lista e mostra ao usuário.

```
dores$ python3 client.py
Available methods: ['add', 'sub', 'mult', 'div']

Choose a method to call (or 'exit' to quit):
1. add
2. sub
3. mult
4. div
Enter the number of the method to call:
```

Fig. 6. Lista de métodos disponíveis na interface

IV. Funções Stub

Recebida a lista de funções disponíveis, o cliente então cria uma função stub para cada função remota, reproduzindo sua assinatura. A função stub criada é responsável por construir e enviar a requisição RPC ao servidor, assim como processar a resposta recebida, e só então retornar um valor à execução do programa.

```python
# encapsulate function signature to call the corresponding remote function
def __create_stub_function(self, method_name):
    def stub(*args, **kwargs):
        request_data = self.build_request_data(method_name, args, kwargs)
        response = self.send_request_with_timeout(request_data)
        return response['result']

    return stub

# create all stub functions based on the list provided by the server
def create_stub_functions(self, available_methods):
    for method_name in available_methods:
        setattr(self, method_name, self.__create_stub_function(method_name))
```

Fig. 7. Funções que criam as stubs no cliente

As stubs são chamadas quando o usuário seleciona na interface qual método ele quer executar, e provê os argumentos corretamente. A requisição é então enviada ao servidor.

V. Recebimento da Requisição

Após instanciar o servidor com o endereço 0.0.0.0:8080, o programa inicializa o servidor reservando um socket para escutar por mensagens RPC. Para cada cliente que envia uma requisição, o servidor cria uma nova *thread* para gerenciar aquele processo. Quando o cliente encerra a conexão, sua respectiva *thread* também é encerrada.

```python
def run(self):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.bind(self.address)
        sock.listen()

        print(f'Server {self.address} running')
        while True:
            try:
                client, address = sock.accept()

                Thread(target=self.__handle__, args=[client, address]).start()

            except KeyboardInterrupt:
                print(f'Server {self.address} interrupted')
                break
```

Fig. 8. Função que configura o socket do servidor

VI. Política de Timeouts

Foi implementada no cliente uma política de timeout para as requisições feitas ao servidor. Foi estipulado um limiar de 5 segundos para receber a reposta de cada requisição, e um total de 3 tentativas de reenviar a mensagem. Passados os 5 segundos sem resposta, o cliente lança uma exceção de timeout e parte para a próxima tentativa. Após as 3 tentativas, a conexão é encerrada e a execução do cliente também, informando ao usuário que houveram múltiplas tentativas de requisição sem sucesso.

VII. Conclusão

O projeto foi uma excelente oportunidade para exercitar os conceitos de camada de aplicação e programação com sockets, e a aplicação desenvolvida funcionou corretamente para os requisitos propostos, obtendo um nível de robustez satisfatório para o contexto do projeto. Um vídeo explicando em maiores detalhes o código produzido e a execução do programa pode ser acessado em https://youtu.be/85AdpaKOf5Q

References

[1] J. F. Kurose, K. W. Ross, Redes de computadores e a internet: uma abordagem top-down, 8a ed., 2021.

[2] G. C. Ferreira, repositório de código "Redes-de-Computadores-UnB", disponível em https://github.com/Gabrielcarvfer/Redes-de-Computadores-UnB/tree/master, acessado em 15/11/2023.