

Trabalho Prático 2: FSPD

Lucas Fonseca Mundim - 2015042134 2022/1

1. Introdução

Esse trabalho prático de FSPD tem como objetivo exercitar os conceitos e a prática de **RPC**, mais especificamente **gRPC**, implementando múltiplos servidores e clientes que devem se intercomunicar através da rede seguindo o protocolo. É importante frisar a expectativa de funcionamento em redes distintas, pois o comportamento de **RPCs** é frequentemente associado a chamadas em servidores remotos que não necessariamente estão na rede do cliente.

2. Detalhes da implementação

Conforme a especificação já havia mencionado, o funcionamento da aplicação é razoavelmente simples. Nessa documentação vou tentar explicar um pouco do código e dos desafios.

2.1. **wallet_server.py**

Para o armazenamento das carteiras foi improvisado uma estrutura de "banco" utilizando dicionários onde a chave é o **id** da carteira e o valor é o saldo. O programa inicia com a importação dos dados do arquivo passado para o banco e inicia o **server**.

2.1.1. **GetBalance**

Para recuperar o saldo da carteira o servidor recebe o **id** do cliente. Se o **id** não se encontra no dicionário, **balance** é definido como **None** e, nesse caso, o saldo **-1** é retornado para o cliente. Se o **id** se encontra no dicionário o valor do saldo "cropado" em 2 casas decimais é retornado.

```
if balance is None:
    return wallet_routes_pb2.Wallet(
        id=request.id,
        balance='-1'
    )
else:
    return wallet_routes_pb2.Wallet(
        id=request.id,
        balance='{:.2f}'.format(balance)
    )
```

2.1.2. **GeneratePaymentOrder**

Para gerar a ordem de pagamento o servidor recebe o valor da ordem e o **id** da carteira pagadora. Conforme especificação o servidor retorna **status=-1** em caso de a carteira pagadora não existir no

banco, `status=-2` em caso de saldo insuficiente e `status=0` em caso de tudo estar certo.

Caso tudo esteja certo, é gerado um objeto `bytes` com `secrets.token_bytes(32)` e armazenado em um dicionário no servidor onde a chave é o objeto e o valor é o valor da ordem

```
secret = token_bytes(32)
wallet_balance = wallet_balance - request.value
self.wallet_db[request.wallet_id] = wallet_balance
self.secret_db[secret] = request.value
return wallet_routes_pb2.PaymentOrderResponse(
    status = 0,
    secret = secret
)
```

2.1.3. GenerateTransfer

A última função desse servidor é finalizar a transferência de dinheiro. Para isso o servidor recebe um valor a ser creditado, o `id` da carteira de destino e os `bytes` gerados por `GeneratePaymentOrder` de forma a garantir que a transação existe e pode ser completada. Novamente conforme a especificação, o estado `status=-1` é retornado em caso da carteira de destino não existir, `status=-2` em caso da transação original `bytes` não existir e `status=-9` em caso dos valores debitado e creditado divergirem. Em caso de "tudo certo", `status=0` é retornado bem como o saldo resultante da carteira de destino.


2.2. wallet_client.py

O cliente do servidor de carteiras tem uma implementação muito simples onde sua maior responsabilidade é definir o comando e os parâmetros a serem enviados ao servidor e a impressão da resposta de interesse recebida, então não será muito aprofundada a sua explicação, somente alguns pontos chave.

2.2.1. Recepção de bytes como parâmetro

Para receber parâmetros no formato `b'\x82\xc0\x028\x11\x92\xc1\x9d'\x9cL*\x8e\xca\x95?\x82"\xf8P\x83hR\x03\xb4\xde\x80i\xf6\x1a\x19i'` recuperado do retorno de `GeneratePaymentOrder` e utilizado como parâmetro de `GenerateTransfer` foi necessário o emprego de alguma solução alternativa. Devido ao escopo simples e de demonstração desse trabalho foi utilizada a função `eval()` do `python` junto da conversão para `bytes` para obtermos o array de bytes sem problemas. Essa solução só deve ser aplicada nesse escopo visto os grandes perigos do uso de `eval()` em sistemas

de produção, já que resultaria na execução de um possível comando enviado.



```
In [3]: def foo(s):
...:     eval(s)
...:

In [4]: foo('print("bar")')
bar
```

2.3. `store_server.py`

O servidor da loja precisa de alguns detalhes a mais que o servidor de carteiras. Especificamente, esse programa faz o papel de *servidor* para a loja mas também o papel de *cliente* para a carteira. Portanto, na inicialização do server, é passado um `stub` da carteira para o servidor da loja. Isso se dá pelo fato de que algumas operações da loja dependem da carteira (ex: recuperar saldo, efetuar venda).

Para o armazenamento do saldo da loja, é feita uma chamada inicial em `GetBalance` e o valor é passado na inicialização do servidor da loja. Ele é atualizado a cada chamada de venda.

2.3.1. `GetPrice`

A função `GetPrice` não tem segredos ou complicações: simplesmente retorna o preço do produto/serviço da loja recebido no seu start-up.

2.3.2. `Sell`

Conforme mencionado anteriormente, a função `Sell` faz uso do servidor de carteiras para seu funcionamento. É feita uma chamada para gerar a ordem de pagamento e, caso essa chamada falhe, seu status é retornado. Caso a ordem de pagamento seja um sucesso, é feita a chamada para gerar a transferência e o saldo da loja é atualizado localmente caso a segunda chamada seja um sucesso.

2.3.3. CloseUp

Essa última função realiza o papel de encerrar o servidor. Na inicialização do servidor é utilizada a função `stop_event.wait()` em vez de `server.wait_for_termination()` presente no servidor da carteira para aguardar algum evento de terminação. Quando o comando de terminação é recebido pelo servidor o evento é lançado com `self._stop_event.set()` e o servidor é encerrado, retornando o saldo final da loja armazenado localmente.

3. Conclusão

O trabalho prático 2 foi muito mais simples e de fácil execução que o trabalho prático 1, que exigia manipulação de primitivas de sincronização, um feito muito mais complexo que a comunicação entre servidor e cliente em RPC.

Os principais desafios encontrados foram a definição dos endpoints e objetos a serem trafegados, a recepção de `bytes` como input do usuário e o começo da implementação do servidor e cliente. Felizmente [a documentação da API de gRPC em Python](#) é bem redigida e com exemplos razoáveis de forma que esse último ponto foi apenas um "ramp up" inicial, que se tornou muito mais simples de repetir em seguida. A [documentação de protobuf](#) para definir os tipos a serem utilizados também foi de fácil compreensão.