2022_1 - FUNDAMENTOS DE SISTEMAS PARALELOS E DISTRIBUÍDOS - TM

Segundo exercício de programação: lojas e bancos digitais

Introdução

Neste exercício vamos praticar o desenvolvimento de aplicações baseadas em chamadas de procedimento remotos (RPC). Na área de desenvolvimento de aplicações em nuvem, RPCs são provavelmente a técnica mais utilizada hoje para comunicação entre partes de um serviço em rede. Mesmo quando a comunicação se dá por sistemas de filas de mensagens, normalmente a interface de programação é construída sobre RPCs. Muitas vezes, diversos serviços baseados em RPC são interligados entre si formando serviços mais complexos, como no caso do que se costuma chamar de arquiteturas de microsserviços. Neste exercício vamos exercitar esses conceitos em um mini-sistema.

Entre os frameworks de programação com RPC existem diversas opções. Neste exercício usaremos gRPC, por ser das opções populares que não estão presas a ambiente de desenvolvimento específicos (como Flask ou node.js, por exemplo).

Objetivo

Neste exercício você deve desenvolver, usando gRPC, um serviço simples com um "banco" e depois uma "loja" para, usando esses dois tipos de servidores, construir um serviço mais elaborado. Dessa forma, a implementação pode ser dividida em duas partes: o serviço do banco e o serviço de compra na loja usando contas no banco.

Observação: para a descrição a seguir, um "string identificador de um serviço" é um string com o nome ou endereço IP de uma máquina onde um servidor executa e o número do porto usado, separados por ":", sem espaços (p.ex., "localhost:555", "150.164.6.45:12345" ou "cristal.dcc.ufmg.br:6789").

Primeira parte: um servidor de contas (quase uma carteira digital)

Primeiramente, seu objetivo é criar um par cliente/servidor que se comunique por gRPC para criar um serviço de controle de contas bancárias, ou carteiras digitais. Cada carteira é identificada por um string, que é considerado secreto e conhecido apenas pelo servidor (o banco) e cada cliente - algo como uma assinatura única. Por simplicidade, aqui esse string pode ser qualquer sequência de caracteres legíveis, sem espaços com até 80 caracteres. O programa servidor manterá um container com associações de strings com valores interios não negativos. O valor inteiro será o valor do saldo associado com a conta/carteira identificada pelo string.

Seu **servidor de carteiras** deve exportar o seguintes procedimentos:

- saldo: recebe como parâmetros um string, identificando uma carteira; caso ela ainda não exista, deve retornar -1; caso a chave exista associada a uma carteira válida, deve retornar o valor associado à carteira no momento;
- ordem de pagamento: recebe como parâmetros o string da carteira e um valor inteiro positivo a ser pago e retorna um inteiro e um vetor de 32 bytes; se a carteira não existe, retorna -1 e um vetor qualquer; se o valor a ser pago é maior que o valor na carteira, retorna -2 e um vetor qualquer; caso contrário, subtrai o valor a ser pago da carteira, cria um segredo usando a função token_bytes(32) da biblioteca secrets, armazena o vetor e o valor em um dicionário para referência futura, retorna 0 e o vetor gerado;
- transferência: recebe como parâmetros um valor inteiro positivo, um vetor de 32 bytes que deve ter sido gerado anteriormente como uma ordem de pagamento e um string identificador da carteira onde o valor deve ser creditado. Se a string não corresponde a uma carteira, retorna -1; se o vetor identificador da ordem de pagamento não existe, retorna -2; se o valor associado à ordem de pagamento é diferente do valor passado como parâmetro, retorna -3; caso contrário, incrementa o valor da carteira com o valor indicado e retorna o valor resultante;
- fim da execução: um procedimento sem parâmetros que indica que o servidor deve terminar sua execução; nesse caso o servidor

deve salvar uma nova versão do arquivo de entrada, com os valores atualizados das carteiras, responder com o número de contas salvas e terminar sua execução depois da resposta (isso é mais complicado do que pode parecer, veja mais detalhes ao final).

Certamente um banco/gerente de carteiras teria diversas outras funcionalidades, mas vamos ficar apenas com essas. Elas já serão suficientes para exercitar a comunicação por RPC.

Nenhuma mensagem deve ser escrita na saída padrão durante a execução normal (mensagens de erro, obviamente, não devem ocorrer durante uma execução normal).

O programa servidor deve receber dois parâmetros de linha de comando: o número do porto a ser usado pelo servidor (inteiro, entre 2048 e 65535), e o nome de um arquivo contendo as carteiras a serem servidas. Esse arquivo deve conter apenas linhas com um string identificador de conta e um valor inteiro não negativo correspondente ao saldo inicial de cada conta, separados por um espaço.

O **cliente de carteira** deve receber como parâmetros dois strings: o primeiro será o identificador da carteira do cliente e o segundo será um "string identificador de um serviço" (como descrito anteriormente) indicando onde o servidor executa. Ele deve ler comandos da entrada padrão, um por linha, segundo a seguinte forma (os programas devem poder funcionar com a entrada redirecionada a partir de um arquivo):

- `S string` - consulta o saldo no servidor de carterias para a carteira do cliente e escreve o valor retornado;
- `O valor` - faz um pedido de geração de uma ordem de pagamento, a partir da carteira do cliente, para o valor indicado e escreve o valor de retorno, se for um erro, ou um valor maior que zero em caso de sucesso, que será usado para identificar a ordem em comandos anteriores (mais detalhes a seguir);
- `X valor op string` - aciona o método de ordem de pagamento do servidor passando o string como identificador da carteira, o valor como o valor a ser pago e usa o inteiro op para resgatar o vetor de bytes que deve ter sido armazenado internamente em um comando O anterior. Escreve -9 se o valor de op não for encontrado, ou o valor inteiro devolvido pelo servidor ao executar o comando;
- `F` - dispara o procedimento de fim de execução do servidor, escreve o valor retornado e termina (somente nesse caso o cliente deve terminar a execução do servidor; se a entrada terminar sem um comando T, o cliente deve terminar sem acionar o fim do servidor).

Qualquer outro conteúdo que não comece com S, O, X ou F deve ser simplesmente ignorado; os comandos usam espaços como separadores; os strings de identificação de contas não podem ter espaços. Pode-se assumir que se uma linha começa com uma das quatro letras (maiúsculas) elas conterão comandos bem formados.

Segunda parte: um serviço de pagamentos

Seu objetivo final é implementar um serviço de pagamentos que inclui uma "loja" que oferece um certo serviço/produto e que deve ser pago através de um servidor de carteiras. Continuando com as simplificações, a "loja" simplesmente informa o preço de seu (único) serviço/produto e o cliente deve então fazer o pagamento com uma ordem de pagamento.

O **servidor da "loja"** recebe quatro parâmetros da linha de comando: um inteiro que será o valor do seu serviço/produto, o número do porto que ele deve utilizar para receber conexões dos clientes, um string com o identificador da sua conta no servidor de carteiras e um "string identificador de um serviço" (como descrito anteriormente) indicando onde o servidor de carteiras executa.

Internamente ele deve manter o saldo da conta da loja. Esse saldo obtido no início da execução, acionando o RPC de saldo do servidor de carteira e deve ser mantido internamente. Depois disso ele deve aceitar/exportar três comandos:

- preço: não utiliza nenhum parâmetro e retorna o preço do serviço/produto, recebido como o primeiro parâmetro da linha de comando e o string identificador de serviço que identifica o servidor de carteiras associado (último parâmetro da linha de comando);
- venda: recebe como parâmetro o vetor de bytes que identifica uma ordem de pagamento, aciona o servidor de carteiras para fazer a transferência do valor do preço do serviço, usando a ordem de pagamento, para a carteira da loja, atualiza o saldo da loja, em caso de sucesso, e escreve o valor retornado pelo servidor de carteiras ou -9 em caso de erro na comunicação com o outro servidor;
- fim da execução: um procedimento sem parâmetros que indica que o servidor da loja deve terminar sua execução; nesse caso o servidor deve responder com o saldo calculado internamente para a conta da loja (novamente, isso é mais complicado do que pode parecer, veja mais detalhes ao final); o servidor de carteiras não deve ser acionado (deve continuar executando).

O **cliente que se conecta ao servidor da "loja"** recebe como parâmetros um string identificador da sua conta no servidor de carteiras e um string identificador de um servidor da "loja". Ele deve então ler comandos da entrada padrão, de um dos tipos a seguir (o programa deve poder funcionar com a entrada redirecionada a partir de um arquivo):

- `F` - dispara a operação de término do servidor da loja, escreve na saída o valor de retorno recebido, então dispara a operação de término do servidor de carteiras (se for conhecido), escreve na saída o valor de retorno e termina a execução;
- `P` - executa o método de consulta de preço da loja e usa o identificador do servidor de carteiras para fazer uma consulta RPC pelo saldo do cliente no servidor de carteiras (usando o identificador da conta recebido como parâmetro); escreve na saída o preço do serviço e o saldo da carteira separados por espaço (não há erros previstos);
- `C` - faz a compra de um produto/serviço, usando uma ordem de pagamento; para isso, executa o RPC de emissão de ordem de pagamento com o valor do produto/serviço no servidor de carteiras e se não há erros, executa a RPC de venda no servidor da loja; escreve na saída o valor de retorno da ordem de serviço e, se não houver erro na primeira operação, na linha seguinte escreve o valor de retorno da operação de venda da loja.

Qualquer outro conteúdo que não comece com C, P ou T deve ser simplesmente ignorado; os comandos nunca serão mal-formados.

Requisitos não funcionais:

O código deve usar apenas Python, sem bibliotecas além das consideradas padrão. **Não serão aceitas outras bibliotecas, nem o uso de recursos como E/S assíncrona em Python**. A ideia é que os programas sejam simples, tanto quanto possível. O código deve observar exatamente o formato de saída descrito, para garantir a correção automática. Programas que funcionem mas produzam saída fora do esperado serão penalizados.

A correção será feita nas máquinas linux do laboratório de graduação. Você deve ser certificar de que seus programas executam corretamente naquele ambiente. Programas que não compilarem, não seguirem as determinações quanto a nomes, parâmetros de entrada e formato da saída, ou apresentarem erros durante a execução serão desconsiderados.

- O laboratório de graduação é onde eu pedi ao CRC para instalar o grpc . Não sei se ele está instalado também em outras máquinas, como a tigre. Recomendo testar sempre no laboratório: https://www.crc.dcc.ufmg.br/infraestrutura/laboratorios/linux.
- A login.dcc.ufmg.br é só o gateway de entrada na rede do DCC, o CRC não instala nada lá. Na verdade, recomenda-se que não se execute nada naquela máquina. De lá, vocês podem fazer ssh para as máquinas do laboratório.
- Vocês podem também instalar a VPN do DCC nas suas máquinas. Com ela ligada, vocês conseguem fazer ssh direto para as máquinas do laboratório, sem ter que passar pela login: https://www.crc.dcc.ufmg.br/servicos/conectividade/remoto/vpn/openvpn/start

O que deve ser entregue:

Você deve entregar um arquivo .zip incluindo todo o código desenvolvido por você, com um makefile como descrito a seguir. Considerando a simplicidade do sistema, um relatório final em PDF é opcional, caso você ache importante documentar decisões de projeto especiais. Entretanto, especialmente na ausência do relatório, todo o código deve ser adequadamente comentado.

Preste atenção nos prazos: entregas com atraso não serão aceitas.

O makefile a ser entregue:

Junto com o código deve ser entregue um makefile que inclua, pelo menos, as seguintes regras:

- `clean` - remove todos os arquivos intermediários, deixando apenas os arquivos produzidos por você para e entrega
- `stubs` - faz a geração dos stubs em Python
- `run_serv_banco` - executa o programa servidor de carteiras
- `run_cli_banco` - executa o programa cliente da primeira parte
- `run_serv_loja` - executa o programa servidor da loja
- `run_cli_loja` - executa o programa cliente da segunda parte

As regras do tipo "run_*" devem se certificar de disparar todas as regras intermediárias que podem ser necessárias para se obter um programa executável, como executar o compilador de stubs.

Para o make run funcionar, você pode considerar que os comandos serão executados da seguinte forma (possivelmente, em diferentes terminais):

```
make run_cli_banco arg1=carteira_cliente arg2=nome_do_host_do_serv_banco:5555
make run_serv_banco arg1=5555  arg2=contas.txt
make run_serv_loja arg1=10 arg2=6666 arg3=carteira_loja arg4=nome_do_host_do_serv_banco:5555
make run_cli_central arg1=carteira_cliente arg2=nome_do_host_do_serv_loja:6666
```

Obviamente, o valor dos argumentos pode variar. Se todos os programas forem executados na mesma máquina, o nome do servidor pode ser "localhost" em todos os casos - mas os programas devem funcionar corretamente se disparados em máquinas diferentes.

Para poder executar os comandos, no makefile, supondo que os programas tenham nomes "svc_banco" e "cln_ban", "svc_loja" e "cln_loja", as regras seriam:

```
run_serv_banco:
     ./svc_banco $(arg1) $(arg2)
run_cli_banco:
     ./cln_banco $(arg1) $(arg2)
run_serv_loja:
     ./svc_loja $(arg1) $(arg2) $(arg3) $(arg4)
run_cli_loja:
     ./cln_loja $(arg1) $(arg2)
```

Referências úteis

Em um primeiro uso de gRPC, pode ser que vocês encontrem diversos pontos que vão exigir um pouco mais de atenção no estudo da documentação para conseguir fazer a implementação correta. Eu considero que os pontos que podem dar mais trabalho e que merecem algumas dicas são os seguintes:

- **Desligar um servidor através de um RPC**

  Como mencionado anteriormente, fazer um servidor de RPC parar de funcionar usando uma chamada de procedimento dele mesmo tem uma pegadinha: não basta chamar um exit() enquanto se executa o código do procedimento, ou ele vai terminar a execução antes de retornar da chamada, deixando o cliente sem resposta. E normalmente a gente só pode escrever código dentro das chamadas, já que não devemos alterar o código do stub. Cada framework de RPC tem uma solução diferente para esse problema e a solução do gRPC é bastante elegante, exigindo pouco código. Usa-se a geração de um evento dentro do código da RPC, que é capturado pelo servidor. Pode parecer complicado, mas o código para se fazer isso já está descrito no stackexchange.

Dúvidas?

Usem o fórum criado especialmente para esse exercício de programação para enviar suas dúvidas. Entretanto, **não é permitido publicar código no fórum**! Se você tem uma dúvida que envolve explicitamente um trecho de código, envie mensagem por e-mail diretamente para o professor.

Certamente, o mundo é mais complicado...

Como a carga horária da disciplina é limitada, como mencionado antes, este é um problema extremamente simplificado e certas práticas de desenvolvimento que são muito importantes no ambiente profissional estão sendo ignoradas/descartadas: seu código não precisa se preocupar com verificação de entradas incorretas, erros de operação, ações maliciosas. Não considerem que isso é um argumento contra essas práticas, mas em prol do foco principal da disciplina, em função do tempo disponível, temos que simplificar.

Na sua vida profissional, tenham sempre em mente que testes exaustivos, programação defensiva (testar todos os tipos de entradas possíveis, etc.) e cuidados de segurança devem estar sempre entre suas preocupações durante qualquer desenvolvimento.

◄ Primeiro exercício de programação (entrega PRORROGADA até as 23:59 do dia 23/05/2022)

Seguir para...

Dúvidas sobre o segundo exercício de programação ▶