2022_2 - FUNDAMENTOS DE SISTEMAS PARALELOS E DISTRIBUÍDOS - TM

Segundo exercício de programação: chamadas de procedimentos remotos

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 o gRPC, por ser das opções populares que não estão presas a um ambiente de desenvolvimento específico (como Flask ou node.js, por exemplo).

Objetivo

Neste exercício você deve desenvolver, usando gRPC, um sistema que inclui um serviço de armazenamento do tipo chave/valor e um serviço de autenticação de usuários (ambos bem simples, obviamente) que funcionam de forma integrada. Dessa forma, a implementação pode ser dividida em duas partes: o serviço de autenticação e controle de acesso e o serviço armazenamento chave/valor.

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:5555", "150.164.6.45:12345" ou "cristal.dcc.ufmg.br:6789").

Primeira parte: um servidor de identidade

Primeiramente, seu objetivo é criar um par cliente/servidor que se comunique por gRPC para criar um serviço de controle de usuários, para autenticação e controle de acesso. Cada usuário é identificado por um string único e uma senha numérica (isto é, um inteiro) o qual é considerado secreto e conhecido apenas pelo servidor pelo usuário. Por simplicidade, aqui esse string pode ser qualquer sequência de caracteres legíveis, sem espaços com até 10 caracteres. O programa servidor manterá um container com associações de strings com valores inteiros não negativos que representam as senhas, além de uma chave binária que pode ser usada para identificar cada usuário em outros serviços (algo como um serviço de login único) .

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 um número que será usado como a senha do administrador do serviço (o único que pode criar outros usuários. O administrador será identificado pelo string "super".

Para fins de implementação, serão usados, no máximo, 100 usuários diferentes em uma execução.

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

- autenticação: recebe como parâmetros o string de identificação e um valor inteiro positivo representando a senha associada e retorna um inteiro e um vetor de 32 bytes sendo que, se o usuário não existir ou a senha estiver incorreta, retorna -1 e um vetor de 32 bytes zero; caso contrário, retorna 0, cria um segredo usando a função token_bytes(32) da biblioteca secrets, armazena o segredo gerado junto ao string do usuário em um dicionário de usuários (substituindo um valor anterior, se ele existir), retorna 0 e o vetor gerado;
- criação de usuários: recebe como parâmetros um string com a identificação de um usuário, um inteiro não negativo com a senha para o mesmo, um string de permissões que deve ser "RW" ou "RO" e um vetor de 32 bytes que deve ter sido fornecido em uma autenticação anterior. Se o vetor de bytes não for igual ao vetor associado ao usuário "super", o usuário não é criado o valor -1 é retornado; caso contrário, o par (identificação,senha) é inserido em um dicionário de usuários cadastrados e o valor 0 é retornado;
- verificação de acesso: recebe como parâmetro um vetor de bytes retornado anteriormente em uma operação de autenticação e retorna o string de permissões do usuário associado ao vetor de bytes, ou "NE" se o vetor não existir no dicionário associado, ou "SP" se o usuário for "super";
- fim da execução: um procedimento sem parâmetros que indica que o servidor deve terminar sua execução; nesse caso o servidor deve responder com o número de usuários criados (sem incluir o "super") e terminar sua execução depois da resposta (isso é mais complicado do que pode parecer, veja mais detalhes ao final).

Certamente um servidor de identidades usaria recursos de criptografia na comunicação e 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 **cliente de identificação** deve receber como parâmetro 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):

- `A string valor` - faz um pedido de autenticação do usuário identificado por string, com senha dada por valor, e escreve o valor inteiro de retorno na tela; se o valor for zero, o vetor de 32 bytes retornado deve ser armazenado internamente ao cliente para ser usado em comandos posteriores (mais detalhes a seguir); se o valor for -1 obviamente o vetor de retorno não deve ser armazenado e se já existir um vetor de bytes anterior, o mesmo deve ser removido;
- `C string1 valor string2` - aciona o método de criação de usuário do servidor passando o string1 como identificador do usuário, o valor como senha e o string2 como o string de permissões e o último vetor obtido por um comando A. Escreve -9 se um vetor de bytes não for encontrado, ou o valor inteiro devolvido pelo servidor ao executar o comando;
- `V` - aciona o método de verificação de acesso no servidor passando o vetor de bytes obtido no último comando A bem sucedido; escreve -9 se tal vetor não existe, ou o string de retorno do comando, caso contrário;
- `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 F, o cliente deve terminar sem comandar o fim do servidor).

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

Sobre a identificação do usuário nos comandos C e V: exibir e escrever as sequências de bytes que vão ser geradas pela biblioteca secrets para representar o segredo de autenticação seria muito trabalhoso. Ao invés disso, o cliente deve armazenar internamente o último vetor obtido por um comando A bem sucedido e incluí-lo nas chamadas que dele necessitem. Se um comando de autenticação retornar sem sucesso, entretanto, qualquer vetor recebido anteriormente deve ser removido.

Segunda parte: um serviço de armazenamento do tipo chave/valor

Seu objetivo final é implementar um serviço de armazenamento de pares com controle de acesso baseado no servidor de identificação.

O **servidor de pares** recebe dois parâmetros da linha de comando: o número do porto que ele deve utilizar para receber conexões dos clientes e um "string identificador de um serviço" (como descrito anteriormente) indicando onde o servidor de identificação executa.

Internamente ele deve manter um dicionário de pares chave/valor. Chaves serão inteiros positivos e os valores associados serão strings com até 4 KB, podendo conter qualquer caractere válido para exibição na tela. Esses pares serão incluídos/removidos através de chamadas de procedimento remoto. O servidor deve aceitar/exportar os seguintes comandos:

- inserção: recebe como parâmetros um inteiro positivo (chave), um string representando seu valor, e um vetor de bytes que identifica o usuário para o servidor de identificação. O servidor de pares deve primeiramente executar um comando de verificação de acesso no servidor de identificação, usando o vetor de bytes de segredo; caso o string retornado seja "RW", o servidor de pares armazena o string em um dicionário, associado à chave, substituindo um valor anterior, caso ela já exista, e retornando zero; caso o string de permissão não seja "RW", o par não deve ser incluído e o valor de retorno deve ser -1, -2 ou -3, caso o string retornado pelo servidor de identificação seja "RO", "NE" ou "SP", respectivamente;
- consulta: recebe como parâmetros um inteiro positivo (chave) e o vetor de bytes de segredo. O servidor de pares deve primeiramente executar um comando de verificação de acesso no servidor de identificação, usando o vetor de bytes de segredo; caso o string retornado seja "RO" ou "RW" o servidor de pares retorna o conteúdo do string associado à chave, caso ela exista, ou um string nulo em qualquer outro caso;
- término: um procedimento sem parâmetros que indica que o servidor deve terminar sua execução; nesse caso o servidor deve responder com zero e terminar sua execução depois da resposta (isso é mais complicado do que parece, mas veja mais detalhes ao final).

O cliente do servidor de pares recebe como parâmetros um string identificador do servidor de carteiras e um string identificador de um servidor de pares. 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):

- `A string valor` - executa um procedimento de autenticação do usuário identificado por string, com senha dada por valor, no servidor de autenticação associado, e escreve o valor inteiro de retorno na tela; se o valor for zero, o vetor de 32 bytes retornado deve ser armazenado internamente ao cliente para ser usado em comandos posteriores (mais detalhes a seguir); se o valor for -1 obviamente o vetor de retorno não deve ser armazenado e se já existir um vetor de bytes anterior, o mesmo deve ser removido;
- `I ch string de descrição` - executa no servidor de pares o procedimento de inserção da chave ch (inteiro positivo), associada ao string de descrição (o restante da linha) como seu valor, usando o vetor de segredo obtido em uma autenticação anterior e escreve na saída padrão o valor de retorno do procedimento; caso não haja um vetor de autenticação válido, simplesmente retorna -9;
- `C ch` - consulta o servidor de pares pelo conteúdo associado à chave ch, incluindo o vetor de segredo ativo, e escreve na saída o string retornado como valor, que pode ser nulo - o mesmo resultado deve ser exibido caso não haja um segredo válido;
- `T` - termina a operação do servidor de pares, que envia zero como valor de retorno e termina (somente nesse caso o cliente deve terminar a execução do servidor de pares; se a entrada terminar sem um comando T, o cliente deve terminar sem acionar o término do servidor);
- `F` - dispara o procedimento de fim de execução do servidor de identificação, escreve o valor retornado e termina (somente nesse caso o cliente deve terminar a execução do servidor; se a entrada terminar sem um comando F, o cliente deve terminar sem comandar o fim do servidor de identificação).

Qualquer outro conteúdo que não comece com A, I, C, F ou T deve ser simplesmente ignorado. Pode-se assumir que os comandos nunca serão mal-formados (não é necessário incluir código de validação).

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.

Cada par cliente servidor define uma interface (API) diferente. Por isso, devem haver dois arquivos do tipo .proto e dois conjuntos de stubs.

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, caso sejam aceitas, serão penalizadas.

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_ident` - executa o programa servidor de identificação
- `run_cli_ident` - executa o programa cliente da primeira parte
- `run_serv_pares` - executa o programa servidor de pares
- `run_cli_pares` - 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. Entretanto, não é aceitável que os stubs sejam recompilados a cada execução (use a regra stubs com dependências corretas).

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

```
make run_cli_ident arg1=nome_do_host_do_serv_ident:5555
make run_serv_ident arg1=5555 arg2=12345
make run_serv_pares arg1=6666 arg2=nome_do_host_do_serv_ident:5555
make run_cli_pares arg1=nome_do_host_do_serv_ident:5555 arg2=nome_do_host_do_serv_pares: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_ident.py" e "cln_ban.py", "svc_pares.py" e "cln_pares.py", as regras seriam:

```
run_serv_ident:
	python3 svc_ident.py $(arg1) $(arg2)
run_cli_ident:
	python3 cln_ident.py $(arg1)
run_serv_pares:
	python3 svc_pares.py $(arg1) $(arg2)
run_cli_pares:
	python3 cln_pares.py $(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.

◄ Dúvidas e curiosidades sobre a disciplina

Seguir para...

Dúvidas sobre o segundo exercício de programação ►