2021_2 - FUNDAMENTOS DE SISTEMAS PARALELOS E DISTRIBUÍDOS - TM

Segundo exercício de programação: armazenamento chave-valor distribuído

Introdução

Neste exercício vamos praticar o desenvolvimento de aplicações baseadas em chamdas 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 de armazenamento de pares (chave,valor) usando dois tipos de servidores construir um serviço mais elaborado. Dessa forma, a implementação pode ser dividida em duas partes: o serviço de registro de chaves e o serviço de consulta propriamente dito.

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 ":" (p.ex., "localhost:555" ou "cristal.dcc.ufmg.br:6789").

Primeira parte: um servidor de pares (chave,valor)

Primeiramente, seu objetivo é criar um par cliente/servidor que se comunique por gRPC para criar um serviço de armazenamento de pares do tipo (chave,valor).

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

- inserção: recebe como parâmetros um inteiro positivo (chave) e um string, e armazena o string em um dicionário, associado à chave, caso ela ainda não exista, retornando zero; caso a chave já exista o conteúdo não deve ser alterado e o valor -1 deve ser retornado;
- consulta: recebe como parâmetros um inteiro positivo (chave) e retorna o conteúdo do string associado à chave, caso ela exista, ou um string nulo caso contrário;
- ativação: recebe como parâmetro um string identificador de um serviço e, **para esta primeira parte**, não executa nenhuma ação, apenas retornando o valor zero;
- 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 deve escrever na saída o valor de retorno do servidor e terminar em seguida.

O string de valor terá tamanho máximo de 4.096 caracteres (aproximadamente 4KB). Durante os testes, não será usado nenhum valor maior que esse limite.

O programa servidor pode receber dois parâmetros de linha de comando: o número do porto a ser usado pelo servidor (obrigatório), e um flag de controle (opcional). Se o flag não estiver presente o programa deve funcionar como descrito nesta primeira parte (método de ativação "vazio"). Caso um segundo parâmetro seja fornecido (qualquer que seja ele - basta verificar o número de parâmetros), o servidor deve implementar o método de ativação que será descrito na segunda parte.

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

O **cliente** desta etapa deve receber como parâmetro um string identificador de um serviço, 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):

- `I,ch, string de descrição` - insere no servidor a chave ch, associada ao string de descrição como seu valor, escreve na saída padrão o valor de retorno do procedimento (0 ou -1);
- `C,ch` - consulta o servidor pelo conteúdo associado à chave ch e escreve na saída o string definido como valor, que pode ser nulo, caso a chave não seja encontrada;
- `A,string identificador de um serviço` - aciona o método de ativação do servidor passando o string identificador como parâmetro e escrevendo na saída o valor inteiro devolvido pelo servidor (que por enquanto deve ser zero, mas mudará na segunda parte);

- `T` - termina a operação do servidor, que envia zero como valor de retorno 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 término do servidor).

Qualquer outro conteúdo que não comece com I, C, A ou T deve ser simplesmente ignorado; os comandos I e C usam vírgulas como separadores; o único lugar onde espaços são permitidos é como parte do string de descrição.

Segunda parte: um servidor que combina servidores

Nos sistemas par-a-par iniciais, como o Napster, todos os nós podiam armazenar qualquer tipo de conteúdo e realizar ou atender consultas com uma certa chave. Ao ser disparado, um par deveria se ligar à "rede Napster", identificando-se para um servidor central e informando quais chaves o par armazenava localmente. Como o valor associado a cada chave poderia ser bem grande (usualmente um arquivo contendo uma música ou um álbum, naquele tempo, já era bem grande), o servidor só guardava a informação de que "o par X disse que tem a chave C". Quando um par queria encontra um arquivo/música (C), ele enviava uma consulta ao servidor central, que basicamente respondia "pergunte ao fulano (X)". Esse tipo de integração é muito comum em serviços em nuvem e vamos criar um pequeno caso de uso nesta parte.

Seu objetivo final é implementar um serviço de consulta que centraliza as chaves de alguns servidores de armazenamento (no máximo 10) e permite que um cliente replique o processo de consulta do Napster, consultando o servidor centralizador e depois consultando o servidor que deve ter o conteúdo. O servidor de conteúdo é o mesmo definido na primeira parte, com a adição de funcionalidade para a operação de ativação, que será descrita mais à frente.

O **servidor centralizador** recebe como parâmetro da linha de comando o número do porto que ele deve utilizar para receber conexões. Internamente, ele manterá um dicionário associando chaves (inteiros) aos endereços dos servidores que as anunciaram (representados por strings identificadores). Ele deve aceitar/exportar três comandos:

- registro: recebe como parâmetro o string identificador de serviço que identifica um servidor de armazenamento de pares chave/valor e a lista de chaves (inteiros) nele armazenadas, armazena cada chave em seu diretório, associando-as ao identificador de serviço recebido, e retorna o número de chaves que foram processadas;
- mapeamento: recebe como parâmetro um inteiro positivo ch, consulta o seu diretório de chaves por servidor e retorna o string identificador de serviço associado ao servidor que contém um par com aquela chave, ou um string vazio, caso não encontre tal servidor;
- término: encerra a operação do servidor centralizador apenas, retorna o número de chaves que estavam registradas e termina. (ATENÇÃO: isso é diferente da especificação inicial, que determinava a terminação dos servidores de nomes também. Esse novo comportamento simplifica a implementação e reflete o comportamento real de uma rede P2P.)

Tratamento de colisões: em serviços "reais", normalmente colisões de chave são documentadas e tratadas de forma mais sofisticada. Neste trabalho, para simplificar, se o servidor centralizador recebe uma chave de um servidor de pares que já estava associada a um outro servidor, ele simplesmente sobrescreve a associação - isto é, o último servidor a registrar uma chave se torna responsável por ela.

O **servidor de pares** (criado na primeira parte) agora deve ser iniciado com um segundo parâmetro de linha de comando para ativar o seguinte comportamento no seu método de ativação:

- ativação: recebe como parâmetro um string identificador de um serviço, conecta-se ao servidor centralizador identificado por aquele string e faz uma chamada do procedimento de registro deste último, passando como primeiro parâmetro o string de identificação dele mesmo (servidor de pares), e como segundo parâmetro uma lista com todas as chaves que já foram inseridas nele até aquele momento (sem os valores associados, apenas as chaves). Retorna para o cliente o número de chaves informadas ao servidor centralizador.

O **cliente que se contecta ao servidor centralizador** recebe como parâmetro um string identificador de um servidor centralizador. 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):
- `T` - dispara a operação de término do servidor, escreve na saída o valor de retorno recebido e termina;
- `C,ch` - executa o métododo de mapeamento do servidor em busca da chave ch; caso a resposta seja um string vazio, não escreve nada; caso contrário, escreve o string de resposta, seguido de ":" , executa uma RPC do tipo consulta para o servidor que foi identificado na resposta do mapeamento e escreve o valor de retorno (que pode ser um string vazio).

Qualquer outro conteúdo que não comece com C ou T deve ser simplesmente ignorado; o comando C usa uma vírgula como separador.

Requisitos não funcionais:

O código deve usar apenas C/C++ padrão ou 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.

O material desenvolvido por você deve executar sem erros nas máquinas linux do laboratório de graduação. A correção será feita naquelas máquinas e 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 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
- `run_cli_pares` - executa o programa cliente da primeira parte
- `run_serv_pares_1` - executa o programa servidor de pares com o comportamento da primeira parte (ativação não faz nada)
- `run_serv_pares_2` - executa o programa servidor de pares com o comportamento da segunda parte (ativação funciona com o servidor central identificado nos argumentos)
- `run_serv_central` - executa o programa servidor da segunda parte
- `run_cli_central` - 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 e gerar códigos executáveis, no caso de C/C++.

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

```
make run_cli_pares arg=nome_do_host_do_serv_pares:5555
make run_serv_pares_1 arg=5555  #para disparar o servidor com o comportamento da primeira parte
make run_serv_pares_2 arg=5555  #para disparar o servidor com o comportamento da segunda parte
make run_serv_central arg=6666
make run_cli_central arg=nome_do_host_do_serv_central: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 máquians diferentes.

Para poder executar os comandos, no makefile, supondo que os programas tenham nomes "svc_par" e "cln_par", "svc_cen" e "cln_cen", as regras seriam:

```
run_serv_pares_1:
     ./svc_par $(arg)
run_serv_pares_2:
     ./svc_par $(arg) qqcoisa  # como descrito, o segundo argumento só tem que existir
run_cli_pares:
     ./cln_par $(arg)
run_serv_central:
     ./svc_cen $(arg)
run_cli_central:
     ./cln_cen $(arg)
```

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:

- **Manipulação de listas em uma chamada de procedimento**

  Cada framework de RPC tem sua forma de lidar com isso. Em Sun RPC/XDR, listas eram representadas como vetores e deviam vir acompanhadas de um outro campo de parâmetro que indicaria quantas posições realmente seriam usadas no vetor. Em gRPC isso é relativamente mais simples, especialmente em Python: a definição do parâmetro usa a palavra chave **repeated** e ele é manipulado exatamente como uma lista. Esse link explica como a lista pode ser manipulada no cliente e no servidor (em Python).

- **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.

- **Uso de streaming ao invés de listas**

  O framework gRPC permite também tratar chamadas onde uma sequência muito grande de parâmetros precisa ser chamada como uma RPC em stream. Streams e repeated têm algumas semelhanças, mas essa página explica quando é melhor usar um ou outro. No nosso caso, usar repeated é mais simples, mas se alguém quiser experimentar streams, não há restrição ao seu uso (desde que funcione, óbvio).

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.

◄ Primeira prova

Seguir para...

Dúvidas sobre o segundo exercício de programação ►