MC833

Projeto 2 - Servidor iterativo sobre UDP

João Pedro Martins
176117

Introdução

Este projeto contempla a implementação de um servidor iterativo sobre UDP, num contexto de catálogo de filmes: o servidor deve permitir que clientes conectados a ele executam uma série de operações.

O objetivo é analisar o funcionamento do protocolo de comunicação UDP operando de forma iterativa, fazendo comparações com o protocolo TCP operando de forma concorrente, que também é um protocolo da camada de transporte no modelo OSI. O UDP é um protocolo simples que garante uma comunicação rápida, enquanto o TCP é mais sofisticado e garante uma comunicação confiável.

O código desenvolvido foca nos conceitos relevantes ao curso: comunicação em rede, no modelo cliente-servidor. Toda a parte de processamento das operações e armazenamento dos dados dos filmes foi implementada de forma a ser simples, sem depender de ferramentas de serialização e nem mesmo de *parsing* complexo de dados. Esses detalhes são esclarecidos nas seções a seguir.

Descrição geral e casos de uso

Ao enviar uma mensagem inicial ao servidor, um cliente é apresentado às seguintes operações:

- cadastrar um novo filme recebendo como resposta positiva o seu respectivo identificador;
- remover um filme a partir do seu identificador;
- listar o título e salas de exibição de todos os filmes;
- listar todos os títulos de filmes de um determinado gênero;
- dado o identificador de um filme, retornar o título do filme;
- dado o identificador de um filme, retornar todas as informações deste filme;
- listar todas as informações de todos os filmes.

O servidor processa e armazena os títulos, sinopses, gêneros, salas em exibição, e atribui um identificador único para cada filme armazenado.

O servidor consegue lidar com vários clientes enviando-o mensagems. Porém, se o servidor estiver se comunicando com um cliente específico, no meio do processamento de uma operação que exige várias etapas de troca de mensagem, como por exemplo a operação de cadastro de um novo filme, o servidor não irá diferenciar se as mensagens que está recebendo ainda são do mesmo cliente que iniciou a operação. Logo, outros clientes que enviarem mensagens nesse tempo causaram interferência na operação.

Para evitar ficar bloqueado indefinitivamente em *receives*, o servidor utiliza um *polling* com timeout sempre que precisar receber mensagens de forma não-bloqueante.

Armazenamento e estrutura de dados do servidor

Todos os dados dos filmes são armazenados em arquivos, em *plain text*. Como mencionado na introdução, para manipular e armazenar os dados dos filmes o servidor não faz uso de nenhuma ferramenta de serialização e nem mesmo precisa processar os dados ao lê-los dos arquivos.

Isso é possível graças a um mapeamento de um para um dos arquivos com cada campo de dado de cada filme, realizado da seguinte forma: são criadas pastas para cada campo de dados dos filmes e nessas pastas é criado um arquivo para cada filme (os arquivos são nomeados com o sufixo "_<*id*> ", onde *id* é o identificador do filme)

Assim, na pasta *titles/* são encontrados os arquivos com nome "*title*_<*id*>" e assim por diante para as pastas *synopses/, genres/* e *rooms/.*

Com isso, para obter o título do filme com id = 2, por exemplo, basta ler o arquivo *titles/title_2.*

Os identificadores dos filmes, por sua vez, são armazenados em um arquivo chamado *"ids"*. Esse arquivo guarda a informação de quais identificadores estão sendo utilizados e quais estão livres no servidor. Ele é o passo inicial quando o servidor quer obter as informações de um filme dado seu identificador.

Detalhes de implementação

O servidor lê e salva o arquivo dos identificadores após cada operação executada, para permitir que todos os clientes vejam sempre o estado mais atualizado do catálogo.

O servidor executa em loop e não pode ser encerrado por um cliente. Para encerrar o servidor de forma segura, mantendo os filmes modificados, é necessário digitar *Ctrl+C* no terminal para enviar um *SIGINT* e encerrar a aplicação quando nenhuma operação estiver em execução.

O servidor executa exatamente 1 *recvfrom* de forma bloqueante: o *recvfrom* inicial, que espera um datagrama com o nome de uma operação para executar. Esperar por uma operação é tudo o que o servidor pode fazer, então não faria sentido querer que isso ocorra de forma não-bloqueante. Todos os demais *recvfrom* (que são aqueles que executam dentro de uma operação) são realizados de forma não-bloqueante, utilizando a função *poll* com um timeout de 1 min (pode ser alterado na diretriz *POLL_TIMEOUT_MS*).

Quando um timeout ocorre durante uma operação, essa operação é cancelada e o servidor fica esperando novos datagramas para iniciar uma nova operação. Um detalhe interessante é que, se um timeout ocorrer durante uma operação mas a mensagem do cliente ainda sim chegar até o servidor (talvez porque o cliente demorou demais para enviar a mensagem, mas ela não foi perdida), o servidor não conseguirá prever a chegada dessa mensagem e a lerá como sendo uma requisição de uma nova operação, quando, na verdade, é uma mensagem atrasada que pode conter, por exemplo, o nome de um filme. Isso causará um desalinhamento do cliente com o servidor, que pode ser consertado apenas reiniciando ambos. O timeout serve e funciona, portanto, apenas para casos de mensagens perdidas, e não para casos de mensagens demasiadamente atrasadas.

Para deixar a comunicação mais clara para o usuário, o cliente, ao ser executado, inicia a comunicação com o servidor ao enviar uma mensagem inicial automaticamente. A partir daí, o servidor consegue enviar ao cliente uma mensagem contendo as operações disponíveis.

Para execução do código

Para executar o servidor, compile:

gcc server.c -o server

E execute:

./server

Para executar o cliente, compile:

gcc client.c -o client

E execute:

./client <hostname>

onde *hostname* é o output do comando Linux *hostname.*

A partir daí basta seguir o que aparece no terminal.

Comparação: UDP iterativo vs TCP concorrente

Para lidar com o fato de o UDP não usar conexões e para implementar uma comunicação com recebimento não-bloqueante de mensagens, o código resultante do projeto com o UDP ficou um pouco maior e mais complexo do que o código com o TCP. Afinal, é necessário aqui ficar dizendo a quem você quer enviar uma mensagem. Com o TCP, é necessário um setup inicial para fazer a conexão, mas depois toda troca de mensagem fica mais simples de executar e de saber de quem a mensagem está sendo recebida.

Porém, em contraposição, o *recvfrom* e o *sendto* utilizados para o UDP não requerem ficar checando se uma mensagem foi enviada por completa ou se apenas uma parte foi enviada, já que nesse caso a unidade enviada é um datagrama completo, e não bytes separados como no TCP. Isso deixou o código das funções de recebimento e envio de mensagens mais simples.

Outro ponto é a confiabilidade na comunicação. Com o TCP, tem-se uma transferência de dados confiável: o cliente e o servidor tem a garantia de que uma mensagem enviada vai chegar ao destinatário. Isso tira a necessidade de implementar *fail-safes* para lidar com casos de mensagens perdidas, que podem ocorrer quando se utiliza o UDP. O cenário de comunicação entre cliente e servidor apresentados nestes projetos, em específico, se beneficia mais do uso do TCP do que do UDP, já que não é um cenário onde é essencial uma comunicação mais rápida com um protocolo mais lightweight, mas sim um cenário onde perda de mensagens não são desejadas.

Por fim, como mencionado, o código produzido utilizando o protocolo TCP é mais legível por possuir um nível de abstração maior do que o código com UDP: ao se estabelecer uma conexão inicial, fica mais fácil saber para quem uma mensagem está sendo enviada e de quem uma mensagem está sendo recebida.

Conclusão

Foram exploradas nos projetos a utilização dos dois principais protocolos de comunicação da camada de transportes na stack TCP/IP. Um é straightforward e lightweight, mas não garante uma comunicação confiável. O outro, é mais sofisticado e requer um pouco de overhead antes de mensagens poderem ser trocadas, mas garante uma comunicação confiável. Cada um tem suas aplicações. No cenário cliente-servidor apresentado no projeto, o TCP seria uma melhor escolha, já que não é um cenário onde é absolutamente necessário uma comunicação mais rápida com um protocolo mais lightweight, mas sim um cenário onde perda de mensagens não são desejadas. A utilização do TCP proporcionou também um código mais abstrato e legível.