Trabalho Prático 3 - Redes de Computadores

Luiz A. D. Berto

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

luizberto@dcc.ufmg.br

Abstract. *This work aims to implement a video transfer through a peer-to-peer network. The video is divided in multiple parts - called chunks - and a client can request chunks to a specific peer, that broadcasts the request to other peers, allowing the client to receive the requested chunks even if the contact peer does not have them. In this document we discuss implementation details of these programs.*

Resumo. *Este trabalho visa implementar uma transferência de vídeo usando uma rede peer-to-peer. O vídeo é dividido em múltiplas partes - denominadas chunks - e um cliente pode solicitar chunks para um peer, que espalha a solicitação para outros peers, fazendo com que o cliente receba os chunks que solicitou mesmo que o nó de contato não os possua. Nesse documento discutiremos os detalhes da implementação desses programas.*

1. Introdução

Redes peer-to-peer são bastante interessantes do ponto de vista técnico. Pelo seu desenho distribuído, elas são uma boa escolha para aplicações que necessitam de larga escalabilidade: a escalabilidade horizontal é bastante fácil, bastando adicionar mais nós na rede, ao passo que uma solução com ponto central (como um servidor) nem sempre pode ser escalada com facilidade. Além disso, por não ter ponto de falha única, ela acaba se tornando bastante interessante para soluções que demandam alta resiliência. Nesse trabalho, a aplicação da rede peer-to-peer foi para a transferência de vídeos em partes, num contexto bastante parecido com o *torrent*.

2. Implementação

Os programas foram implementados na linguagem de programação *Rust*. Foram desenvolvidos três *crates* (módulos): *common*, para as funcionalidades em comum dos programas, *client* e *peer*, que são os programas executáveis em si. Todos os programas operam sobre IpV4, usando exclusivamente UDP para a comunicação.

2.1. Common

Na *crate common*, foram implementadas algumas funcionalidades em comum ao cliente e ao peer, como:

- O *enum Message*, que codifica os cinco tipos de mensagens definidos na especificação;
- Funções para criação de instâncias do enum *Message*, serialização e deserialização;
- Funções para converter números em array de bytes;
- Tipos de dados abstratos que encapsulam algumas operações usadas em ambos os programas.

2.2. Client

Na *crate client*, foi feita a implementação do cliente da rede peer-to-peer. O cliente recebe como argumentos o ip e a porta do peer com o qual fará o contato inicial, além dos identificadores dos chunks que serão solicitados à rede peer-to-peer.

Após validar os argumentos recebidos, o cliente cria um socket UDP numa porta aleatória, e o usa para enviar uma mensagem do tipo *HELLO* para o peer de contato (esse socket também é usado para receber mensagens dos peers). Isso dispara uma busca na rede pelos chunks solicitados, conforme detalhado na especificação. Após receber a mensagem do tipo *CHUNKS-INFO* de um peer p, o cliente envia uma mensagem *GET* para o peer p. Ao receber uma mensagem do tipo *RESPONSE*, o cliente atualiza a lista de chunks já recebidos, salva no arquivo *output-ip.log* qual o peer que entregou o chunk, e salva o chunk no sistema de arquivos.

Além do funcionamento básico, o cliente também garante que nunca enviará mais de uma mensagem *GET* que contenha o mesmo id de chunk. Para isso, um algoritmo bastante simples é utilizado, levando em consideração se um chunk já foi enviado ou não em uma mensagem *GET* antes de enviá-la novamente.

Outro aspecto interessante de se notar sobre o cliente é o controle de timeout. Ao enviar a mensagem *HELLO*, o cliente inicia um timer de 5 segundos. Esse timer é o tempo máximo que o cliente esperará por mensagens de peers. Caso o timer se esgote e algum dos chunks solicitados não tenha sido recebido, o cliente finaliza a execução, e salva quais foram os chunks não recebidos no arquivo *output-ip.log*. Isso é necessário pois não existe garantia que todos os chunks requisitados pelo cliente existirão em peers de distância menor ou igual a três unidades do cliente. Caso não houvesse timeout, o cliente ficaria numa espera infinita. Para viabilizar o controle de timeout, o cliente utiliza um socket UDP não bloqueante: as chamadas *recv_from* não bloqueiam caso nenhum datagrama tenha sido recebido no momento da chamada. Isso impede que as verificações de timeout só ocorram após receber mensagens, evento que pode não acontecer em casos que nenhum dos peers possuam algum dos chunks solicitados.

2.3. Peer

Na *crate peer*, foi feita a implementação do peer. O peer recebe como argumentos o ip e a porta que fará o bind para esperar por mensagens, além do path para o arquivo de chave-valor que representa quais os chunks que o peer possui (e suas localizações no sistema de arquivos), além dos ips e portas dos peers conhecidos. Após validar os argumentos, o peer lê o arquivo chave-valor, e carrega para a memória o conteúdo dos chunks, salvando-os num *HashMap*. Esse carregamento inicial evita com que o peer tenha que ler os arquivos várias vezes, resultando em ganhos de performance.

O peer trata das mensagens *HELLO* e *QUERY* de maneira bastante parecida: ao receber mensagens de algum desses dois tipos, o peer verifica se possui algum dos chunks solicitados, e caso possua, envia os identificadores destes numa mensagem do tipo *CHUNKS-INFO* para o cliente solicitante. A diferença é que ao receber uma mensagem do tipo *HELLO*, o peer envia a mensagem *CHUNKS-INFO* diretamente para o endereço remetente de *HELLO* (obtido via recv_from), ao passo que ao receber uma mensagem do tipo *QUERY*, o peer envia a mensagem *CHUNKS-INFO* para o endereço descrito pelos

campos IP e PORTO da própria mensagem *QUERY*. Isso é necessário pois a mensagem *QUERY* é trafegada apenas entre peers.

Além de enviar mensagens do tipo *QUERY*, o peer também executa a lógica de alagamento: o peer decrementa o *TTL* da mensagem e verifica se o valor resultante é maior que 0. Caso for, o cliente espalha a mensagem (com o *TTL* decrementado) para seus vizinhos na rede peer-to-peer. No caso de mensagens do tipo *HELLO*, o próprio peer cria uma mensagem *QUERY* equivalente, e inicia o alagamento, com valor inicial *TTL=3*.

Já ao receber uma mensagem do tipo *GET*, o peer verifica se de fato possui todos os chunks solicitados, e para cada chunk, busca seu conteúdo no *HashMap* criado inicialmente, e envia uma mensagem do tipo *RESPONSE* para o cliente solicitante com o conteúdo do chunk em questão.

Um aspecto interessante de se notar é que o peer é *single-threaded*, pois funciona num esquema de requisição e resposta sobre um socket UDP.

3. Execução

Para compilar o programa, basta rodar o comando *cargo build –release*. Esse comando gerará os executáveis *cliente* e *peer* na pasta *target/release*. Os programas podem ser executados normalmente, recebendo os argumentos denotados na especificação.

4. Conclusão

Conclui-se que redes peer-to-peer, apesar de complexas, são bastante interessantes para aplicações que envolvem transferência de arquivos (frequentemente grandes). Devido à sua natureza distribuída e tolerante a falhas, elas resolvem o problema com bastante eficiência, e podem ser aplicadas para diversos outros tipos de problemas.

References

(2021). Linux man pages. `https://linux.die.net/man/`. Acessado em: 2021-03-17.