Um Sistema *Peer-to-peer* de Compartilhamento de vídeo

Introdução

Este trabalho tem por objetivo implementar um sistema de *torrent* em uma arquitetura sem servidor centralizado, frequentemente denominada *peer-to-peer*. O objetivo do nosso sistema *torrent* é realizar o compartilhamento rápido de vídeos na Internet. Ao contrário da arquitetura cliente-servidor, onde o desempenho do *download* fica a cargo da capacidade do caminho entre eles, a arquitetura *peer-to-peer* permite o *download* distribuído e simultâneo de diferentes *pares*. A Figura 1 ilustra a diferença entre as duas arquiteturas. A rede *peer-to-peer*, Figura 1(b) consiste em uma arquitetura descentralizada na qual os *peers* estabelecem conexões virtuais sem a necessidade de um servidor central. Além disso, clientes podem se conectar nessa rede e solicitar diversos arquivos a partir de *peers* diferentes.

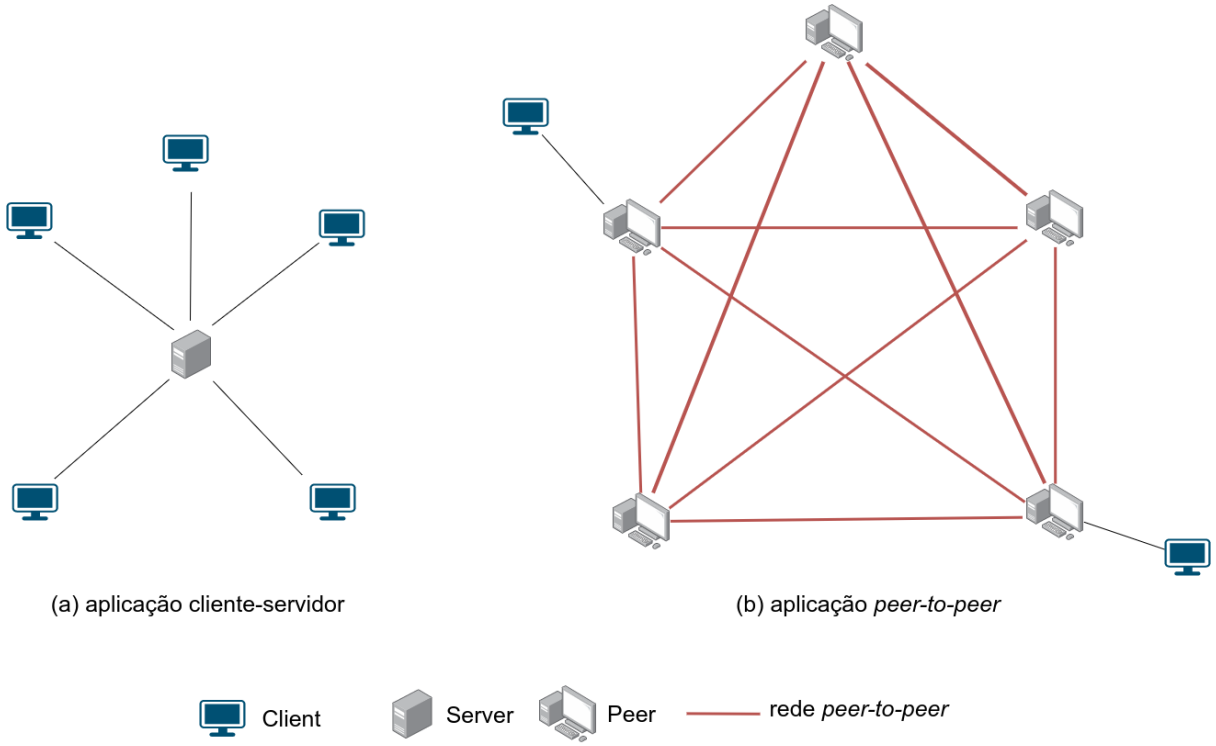


Figura 1 - Aplicações cliente-servidor e *peer-to-peer*

Para o compartilhamento de vídeo nas redes *peer-to-peer*, os arquivos de vídeo são codificados em pequenos pedaços, frequentemente denominados “chunks”, e ficam disponíveis em vários *peers* espalhados na Internet. Isso possibilita que os clientes baixem diferentes *chunks* a partir de vários *peers* simultâneamente. Já em uma arquitetura cliente-servidor (Figura 1(a)) os *chunks* ficam armazenados apenas no servidor, tornando ele um ponto crítico da rede.

Este trabalho deve expor os alunos a pelo menos dois aspectos de projeto em redes de computadores: os aspectos de implementação de um algoritmo distribuído sobre um ambiente composto por um número desconhecido de participantes, e uma solução para roteamento em uma topologia não conhecida *a-priori*.

As seções a seguir descrevem o projeto em linhas gerais.

O Problema

A ideia deste trabalho é implementar a funcionalidade básica de um sistema de *torrent*, cuja função é baixar arquivos grandes via Internet. No entanto, vamos utilizar uma rede *peer-to-peer* para obter o arquivo do vídeo em partes, de diferentes *peers* da rede. Utilizaremos como exemplo o vídeo Big Buck Bunny[1], na qual os *chunks* já estão codificados em 2 segundos de vídeo cada. Os *chunks* que serão utilizados neste trabalho estão disponíveis aqui.

Vocês devem implementar um protocolo em nível de aplicação, utilizando interface de *sockets* UDP. Dois programas devem ser desenvolvidos:

1. Um programa do sistema *peer-to-peer* que atuará como servidor e cliente simultaneamente, denominado ***peer***. O ***peer*** será responsável pelo armazenamento dos *chunks* do vídeo (servidor) e pelo controle da troca de mensagens com os outros *peers*.
2. Um programa ***cliente***, que receberá uma lista de identificadores correspondente aos *chunks* do vídeo que ele deseja receber, bem como armazená-los. Pense nesse *cliente* como um player de vídeo, que faz requisições dos *chunks* para construir seu *buffer* e reproduzir o vídeo para o usuário final (não precisa de interface gráfica).

Um programa *peer*, ao ser iniciado, deverá receber um IP local e um número de porto onde escutará por mensagens (`local-port`), o nome de um arquivo contendo um conjunto de chaves associadas com valores (`key-values-files`, mais detalhes à frente) e uma lista de endereços de outras instâncias desse mesmo programa que estarão executando no sistema (em formato `IP:porto`). A linha abaixo mostra um exemplo de invocação ao programa, onde [id] é referente ao número do *peer* (veja a Figura 2).

```
peer <ip-local>:<local-port> <key-values-files_peer[id]> <ip1:port1> ...
<ipN:portN>
```

O programa *peer* deve então ler o arquivo `key-values-files` e abrir um socket UDP no porto local indicado e ficar esperando por mensagens.

A lista de pares `IP:porto` recebida na linha de comando identifica os pares que são *vizinhos* daquele nó. Cada nó pode trocar mensagem com seus vizinhos, e a rede *peer-to-peer* é formada pelas vizinhanças formadas entre os nós da rede, criando uma rede sobreposta (*overlay*). Veja a Figura 2.

[1] https://peach.blender.org/

O Arquivo de Dados Chave-valor

O arquivo `key-values-files` armazena quais *chunks* cada servidor tem. Ele deve ser representado em formato <*key*:*value*> , onde a *key* determina o ID do *chunk* e o *value* é uma *string* do nome do *chunk*. A *string* será definida pelo padrão BigBuckBunny_Nm4s, onde N denota o ID do *chunk*. Por exemplo, em um determinado *peer* estão armazenados os *chunks* 5, 6, 7 e 8. Portanto, o arquivo key-values-files desse servidor terá o seguinte formato.

```
5: path/to/file/BigBuckBunny_5.m4s
6: path/to/file/BigBuckBunny_6.m4s
7: path/to/file/BigBuckBunny_7.m4s
8: path/to/file/BigBuckBunny_8.m4s
```

O programa ***cliente*** deve ser disparado com o endereço e porto de um *peer* da rede sobreposta que será seu ponto de contato com o sistema distribuído. Além disso, deve receber uma lista de identificadores (IDs) daqueles *chunks* que deseja receber.

```
cliente <IP:port> <5,6,7>
```

O ***cliente***, deve então, construir uma mensagem de consulta e enviá-la para o ponto de contato (veja no protocolo abaixo).

O protocolo de comunicação entre os pares já está pré-definido e será um protocolo de alagamento (*flooding*) confiável, como o utilizado pelo OSPF, que é a opção mais simples para esse tipo de problema. Um protocolo de alagamento caracteriza-se pela disseminação de mensagens por um dispositivo aos dispositivos vizinhos, que sucessivamente repetem o mesmo procedimento, levando a inundação da rede com um todo, e portanto alcançando todos os dispositivos que pertencem à rede.

Arquitetura do Sistema

A arquitetura do sistema é composta por P *peers* e um número arbitrário de *clientes*.

Peers:

A aplicação *peers* precisa ser capaz de lidar com várias consultas (ver mais à frente) simultaneamente, bem como transmitir vários *chunks* a diferentes *clientes* concorrentemente.

Cliente:

A aplicação ***cliente*** também deve ser capaz de receber vários *chunks* de diferentes *peers* simultaneamente.

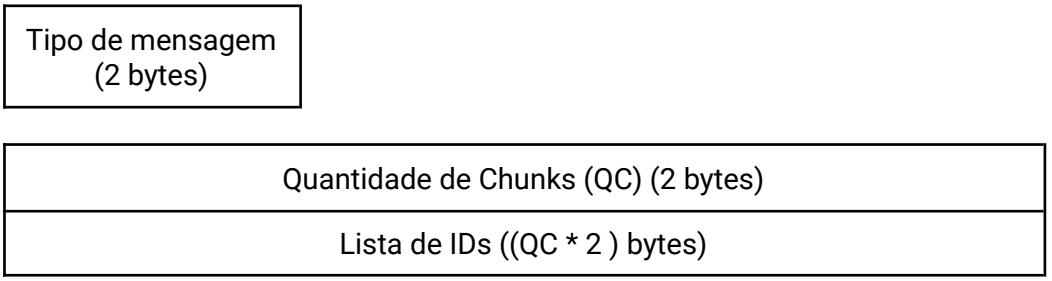
Mensagens do Protocolo

A aplicação utiliza apenas três trocas de mensagens. A primeira refere-se à comunicação do *cliente* com o seu ponto de contato. A segunda é entre os *peers*, referente às consultas e a terceira é referente a comunicação entre os *peers* e os clientes.

Cliente

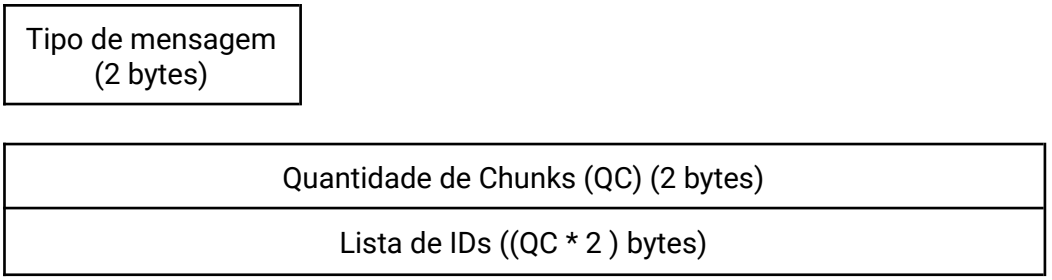
O *cliente* envia uma mensagem UDP para o seu ponto de contato contendo apenas um campo de tipo de mensagem com valor 1 (HELLO) a quantidade de *chunks* e a lista de identificadores (IDs) dos *chunks* que deseja fazer o download.

Cabeçalho da mensagem HELLO (do cliente para o peer ponto de contato)



A partir disso, o cliente fica aguardando respostas (CHUNKS INFO) dos *peers* que contém os *chunks* consultados (ver na descrição do *peer* abaixo). Conforme o cliente vai recebendo as respostas, o mesmo fica ciente de quais *peers* tem os *chunks* desejados. Em sequência, o *cliente* envia uma mensagem (GET), com valor 4, para aqueles *peers* que detém os *chunks*.

Cabeçalho da mensagem GET
(de cliente para peers que possuem os chunks requisitados)



O *cliente* pode receber várias respostas simultâneas (CHUNKS INFO) para o mesmo *chunk*, pois vários *peers* podem conter chunks iguais. Neste caso, você deve estabelecer uma forma de escolher para qual *peer* você vai enviar a mensagem (GET). Em um cenário real, um método interessante seria comparar a qualidade de serviço (QoS) do caminho entre o cliente e os *peers*. Desta forma, poderíamos escolher aquele *peer* com melhor vazão, por exemplo. Mas, para facilitar, vocês podem implementar algo mais simples, como uma escolha *pseudo*-aleatória. Além disso, você deve garantir que não enviará uma mensagem GET mais de uma vez para o mesmo *chunk*. Vocês devem fazer o controle dos *chunks* consultados e os já recebidos.

Após o envio da mensagem GET, o cliente começa então a receber os *chunks* daqueles *peers* selecionados por ele. À medida que as respostas (RESPONSE) cheguem ele deve salvar em um arquivo de log com o nome output-IP.log (definido abaixo), indicando que *peer*

respondeu (definido pelo par `IP:porto`) e ao final do tempo de espera, ele deve salvar que não há mais respostas. Isso significa que o cliente deve implementar um *timeout* para que a mensagem de inundação seja completada.

O arquivo de log deve ter nome **output-IP.log**, onde o parâmetro IP corresponde ao IP do *cliente*. O arquivo deve conter M linhas contendo em cada uma, **peerIP:peerPort - chunkID**, onde peerIP é o IP do *peer* que tem o *chunk,* peerPort é o porto do peer que tem o *chunk*, e o ID do *chunk* é o ID do chunk que o *peer* transmitiu. Para *chunks* que não estão disponíveis, isto é, *chunks* pros quais o cliente não recebeu uma mensagem CHUNKS INFO (abaixo), os valores peerIP e peerPort devem ser preenchidos com 0.0.0.0 e 0, respectivamente. Em C, o formato de cada linha deve ser "%s:%d - %d\n".

Peers

Os *peers* trocam mensagens entre si e com os clientes. As trocas de mensagens entre os *peers* se dão pelo tipo de mensagem, com valor 2. Esse tipo de mensagem identifica uma consulta (QUERY).

Cabeçalho da mensagem de consulta (QUERY, de peer para peers vizinhos))

Tipo de mensagem (2 bytes)

IP:PORTO IP (4 bytes) PORTO (2 bytes)
Peer-TTL (2 bytes)
Quantidade de Chunks (QC) (2 bytes)
Lista de IDs ((QC * 2) bytes)

Nota: O campo IP da mensagem QUERY tem apenas 4 bytes para o endereço IP por que este trabalho utiliza apenas o protocolo IPv4.

O *peer* que recebe a consulta do *cliente* (HELLO) gera a primeira mensagem QUERY associada, preenchendo o IP e o número de PORTO do *cliente* (que podem ser obtidos do `recvfrom`), inicializa o campo Peer-TTL com o valor 3. Além disso, envia a quantidade de *chunks* e a lista dos *chunks* que o *cliente* deseja. Essa mensagem é então repassada a todos os seus vizinhos. Em sequência, cada *peer* que receber a mensagem de consulta deve verificar se possui os *chunk*s e responder ao *cliente*.

A primeira resposta para o cliente é do tipo CHUNK INFO, com valor 3. Essa mensagem informa ao cliente quais (Lista de IDs dos *chunks*) e quantos (QC) *chunks* o *peer* tem armazenado. A partir disso, os *peers* ficam aguardando a mensagem GET.

Cabeçalho da mensagem de consulta
(CHUNK INFO, do peer com chunk requisitado para o cliente)

Tipo de mensagem (2 bytes)

Quantidade de Chunks (QC) (2 bytes)
Lista de IDs ((QC * 2) bytes)

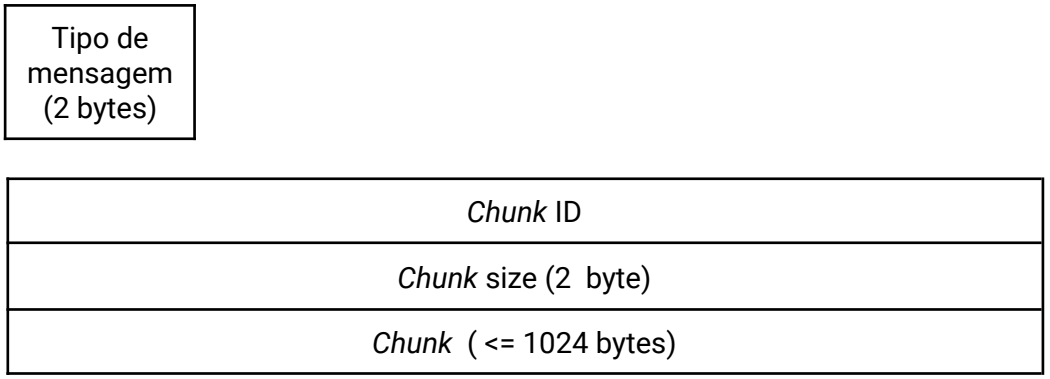
Caso o *peer* receba a mensagem GET, ele envia uma mensagem RESPONSE (definido abaixo) e começa a transmitir o(s) *chunk(s)* para o *cliente* requisitante. Para permitir interoperabilidade, todos os campos inteiros da estrutura acima devem estar em *network byte order*.

Cada *peer* deve implementar um protocolo de alagamento confiável (*reliable flooding*), semelhante ao usado pelo OSPF para disseminar as mensagens de QUERY. O *peer* deve procurar pelo *chunk* no seu repositório local e enviar uma primeira resposta (CHUNK INFO) para o *cliente* caso encontre o *chunk* requisitado. Além disso, o *peer* deve decrementar o valor do Peer-TTL e, se o valor resultante for maior que zero, ele deve retransmitir a mensagem para seus vizinhos, menos aquele do qual recebeu a mensagem.

Note que cada *peer* altera apenas o Peer-TTL da mensagem de QUERY. Todos os demais campos permanecem com os valores preenchidos pelo *peer* que criou a mensagem (o contato com o cliente). Como o valor inicial do Peer-TTL é 3, se houver cinco *peers* conectados em sequência (p.ex., A-B-C-D-E) e o primeiro da cadeia (A) receber uma mensagem de um cliente, a QUERY gerada por ele atingirá apenas 3 *peers* depois dele (B, C e D). Em sistemas reais o valor do TTL é um compromisso entre o diâmetro da rede, a maior distância entre quaisquer dois nós na rede, e da sobrecarga imposta devido à inundação (*flooding*) das mensagens de QUERY. Quanto maior o TTL, maior o número de nós atingidos durante a inundação e maior a chance de encontrarmos um resultado para a busca, porém o custo de inundação aumenta.

Como mencionado, qualquer *peer* que receber uma mensagem GET deve enviar uma mensagem RESPONSE diretamente para o *cliente* que fez a consulta, que pode ser identificado pelos campos de IP e PORT na mensagem QUERY/GET. A mensagem RESPONSE também é simples, contendo apenas um campo de tipo de mensagem (`2 bytes`) com valor 5 (RESPONSE), o tamanho do *chunk* e o *chunk*.

Cabeçalho da mensagem RESPONSE
(de peer com chunk requisitado para o cliente)



Como descrito, há mensagens trocadas entre cliente->*peer, peer*->cliente e *peer->peer*. Abaixo descrevemos os tipos de mensagens entres os elementos deste trabalho.

- cliente -> *peer* (HELLO (1))
- cliente -> *peer* (GET (4))
- *peer* -> *peer* (QUERY (2))
- *peer* -> cliente (CHUNK INFO (3))
- *peer* -> cliente (RESPONSE (5))

Tratamento de Erros

Indisponibilidade de *chunks*. Um cliente pode requisitar *chunks* que não estão disponíveis dentro de 3 hops na rede P2P (lembre-se que 3 é o TTL inicial colocado nas mensagens de QUERY). Neste caso, o cliente não receberá mensagens CHUNK INFO para alguns *chunks*, mesmo se estiverem disponíveis na rede. Cada *chunk* indisponível deve ser impresso no arquivo de log com uma linha “0.0.0.0:0 - chunkID”, onde chunkID é o ID do *chunk* indisponível. A detecção de *chunks* indisponíveis deve ser feita por meio de um temporizador (p.ex., 5 segundos).

Retransmissão em caso de erros. Neste trabalho não iremos tratar o caso de erros de transmissão que levam a perda de pacotes. Desta forma, podemos assumir que todos os pacotes serão entregues até o destino. Valendo 3 pontos extras, os alunos podem implementar retransmissões de mensagens para tratar perda de pacotes. Em particular, o não recebimento de mensagens CHUNKS INFO deve levar à retransmissão da mensagem de HELLO e execução de nova busca pela rede P2P. O não recebimento de uma mensagem RESPONSE deve levar à retransmissão da mensagem GET. As retransmissões devem ser disparadas por meio de temporizador. Mensagens HELLO devem ser retransmitidas um número finito de vezes (p.ex., 5 vezes), para evitar travamento do cliente em caso de *chunks* indisponíveis (ver acima). Mensagens GET também devem ser retransmitidas um número finito de vezes (p.ex., 5 vezes), para evitar travamento do cliente em caso de *peers* que saírem da rede P2P.

Ambiente experimental

Mininet

O Mininet[2] é uma ferramenta para emular uma rede completa de *hosts*, *links* e *switches* em uma única máquina. O Mininet cria redes virtuais usando virtualização baseada em processos e *namespaces* de rede (a partir do Kernel do Linux). O objetivo de utilizarmos essa ferramenta é para facilitar a visualização da rede sobreposta criada pela rede *peer-to-peer*. Com o Mininet é possível criar diversas topologias de redes, desde as mais simples (uma rede interna da sua empresa), até redes complexas como em *datacenter* (com milhares de *host* e *switches*). Neste trabalho, utilizaremos uma topologia de rede pré definida e simples. O principal objetivo é criarmos os *peers* e os *clientes.* Sugerimos esse tutorial para instalar e configurar o Mininet. **Atente-se ao processo de instalação, utilização do Mininet e criação de topologias.** Além disso, você pode optar por seguir a instalação oficial neste link. A instalação é realizada no seu sistema operacional local. No entanto, caso não queira realizar a instalação localmente, você pode utilizar uma máquina virtual[3] disponibilizada pelos próprios desenvolvedores do Mininet.

Disponibilizamos uma máquina virtual com interface gráfica já com o Mininet funcional. Baixe aqui.

Topologia e Exemplo de Execução

A topologia que iremos utilizar é a *single*. Essa topologia é formada por N hosts e um *switch*. Para o caso base deste trabalho, utilizaremos 7 *hosts*, sendo 5 que irão compor a rede *peer-to-peer* e 2 que serão os *clientes.* Fica a critério de vocês experimentarem outras topologias. Para criar a rede base, execute o seguinte comando:

```
sudo mn --xterms --topo single,7
```

A partir disso, 9 terminais irão se abrir. Os terminais "*controller:c0" e "switch:s1"* podem ser fechados. Os testes serão feitos nos outros terminais (h1-h7). Os *hosts* h6 e h7 serão os *clientes* e os demais serão os *peers*. Um aspecto importante do mininet é sempre rodar o comando --clean após o uso do mesmo. Para fazer isso, basta utilizar o seguinte comando:

```
sudo mn --clean
```

A Figura 2 ilustra um exemplo de execução do mininet, bem como um exemplo de topologia da rede *peer-to-peer* e os clientes. Neste exemplo, temos o *peer* 1 conectado ao *peer* 2 e 3, o *peer* 2 conectado ao *peer* 1 e 5 e assim sucessivamente. O cliente 1 tem contato direto com o *peer* 1 e o cliente 2 com o *peer* 4. Neste cenário, a execução se dá como se segue:

No peer 1: `./peer 10.0.0.a:5001 key-values-files_peer1 ip_peer3:5003 ip_peer2:5002`
No peer 2: `./peer 10.0.0.b:5002 key-values-files_peer2 ip_peer1:5001 ip_peer5:5005`
No peer 3: `./peer 10.0.0.c:5003 key-values-files_peer3 ip_peer1:5001 ip_peer5:5005`
No peer 4: `./peer 10.0.0.e:5004 key-values-files_peer4 ip_peer5:5005`

[2] http://mininet.org/
[3] http://mininet.org/download/

No peer 5: `./peer 10.0.0.f:5004 key-values-files_peer5 ip_peer2:5002 ip_peer3:5003 ip_peer4:5004`

No cliente 1 : `./cliente ip_peer1:5001 5,6,7`
No cliente 2 : `./cliente ip_peer4:5004 1,3,4,5,9`

Além disso, cada *peer* deve conter seu próprio arquivo `key-values-files_peer[id]`, que indica qual *chunk* o *peer* detêm. Aqui vocês podem encontrar exemplos (e o padrão a ser seguido) dos arquivos. Fiquem à vontade para experimentarem outras distribuições dos *chunks* nos *peers*.

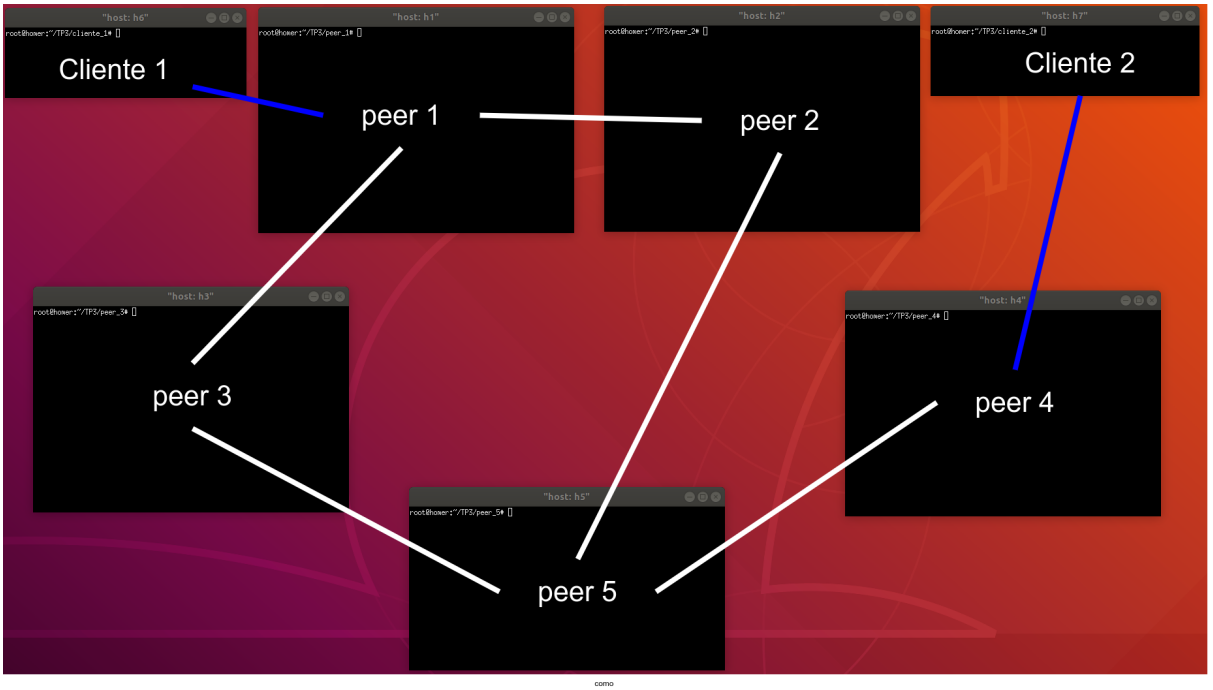


Figura 2 - Exemplo de topologia da rede *peer-to-peer*

Avaliação

Este trabalho deve ser realizado individualmente e deve ser implementado em uma das seguintes linguagens: C, C++, Java, Python, Rust e Go, utilizando apenas as bibliotecas *sockets* das respectivas linguagens. Seu programa deve rodar no sistema operacional Linux e, em particular, não deve utilizar bibliotecas do Windows, como o winsock. Procure escrever seu código de maneira clara, com comentários pontuais e bem indicados; isto facilita a correção dos monitores e tem impacto positivo na avaliação.

Distribuição dos pontos

Documentação - 25%
Código e testes - 75%

Correção Semi-automática

A avaliação será feita através das saídas dos clientes. Realizaremos uma bateria de testes onde os *chunks* estarão dispostos de diferentes maneiras nos *peers* e iremos comparar a saída (**output-IP.log**). Além disso, analisaremos a rede *peer-to-peer* em diferentes topologias no Mininet, bem como a implementação do protocolo (você pode testar com as implementações dos seus colegas). Os testes avaliam a aderência do seu *peer* ao protocolo de comunicação inteiramente através dos dados trocados através da rede (a saída do seu servidor na tela, e.g., para depuração, não impacta os resultados dos testes).

Entrega

Cada aluno deve entregar junto com o código um relatório em formato PDF que deve conter uma descrição da arquitetura adotada para o servidor, os refinamentos das operações identificadas e implementadas pelo código, as estruturas de dados utilizadas e decisões de projeto de protocolo não documentadas nesta especificação. Como sugestão, considere incluir as seguintes seções no relatório:

1. Introdução
2. Arquitetura
3. *Peer*
4. Cliente
5. Discussão

Cada aluno deverá entregar o código fonte. A implementação deve utilizar apenas as bibliotecas padrão de cada linguagem (i.e., não devem requerer a instalação de bibliotecas adicionais) e devem compilar no Linux (i.e., não devem utilizar bibliotecas específicas de outros sistemas operacionais). Entregas em C e C++ devem incluir um Makefile para compilação; entregas em Java devem incluir arquivos JAR; entregas em Rust devem utilizar cargo para compilação. Em qualquer um dos casos, os programas devem ser chamados “cliente” e “peer” (com extensões das respectivas linguagens, quando adequado).

Restrições e Dicas

Podem ser utilizadas apenas as bibliotecas padrões de cada linguagem de programação. Em particular, a comunicação em rede deve ser realizada com funções compatíveis com o padrão POSIX.

Lembramos que soquetes UDP não estabelecem conexão (não é necessário chamar a função `connect`). Recomendamos o uso das funções `sendto` e `recvfrom` para facilitar captura das informações do endereço IP e porto do cliente e *servent* que enviaram a mensagem. Além disso, como o UDP é orientado a pacotes (datagramas), todas as mensagens chegam necessariamente da forma como foram enviadas e não podem se juntar a outras mensagens, então não é preciso ter os mesmos cuidados que com o TCP.

Outras dicas:

- Poste suas dúvidas como comentários no Moodle publicamente acessíveis para os outros alunos
- Procure escrever seu código de maneira clara, com comentários pontuais e bem identado.
- Consulte o monitor antes de usar qualquer módulo ou estrutura diferente dos indicados.
- Não se esqueça de enviar o código junto com a documentação.
- Implemente o trabalho por partes. Por exemplo, crie o formato da mensagem e tenha certeza que o envio e recebimento da mesma está correto antes de se envolver com a lógica da entrega confiável e sistema de mensagens.

Sugestão de Abordagem

1. Implementar comunicação entre cliente e peer “ponto de contato”, com as mensagens HELLO, CHUNKS INFO, GET, RESPONSE. Com estas mensagens, o cliente consegue baixar todos os chunks armazenados no peer “ponto de contato”. Com esta funcionalidade, temos um sistema similar a um sistema cliente-servidor, por que o cliente interage com um único peer.
    a. Implementar HELLO no cliente e CHUNKS INFO no peer. Testar.
    b. Implementar GET no cliente e RESPONSE no peer. Testar.
2. Implementar a busca (mensagens QUERY) e ajustar a implementação do cliente para baixar os chunks de vários peers.