UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
INF01151 - Sistemas Operacionais II

Arthur Lucena Fuchs 00261577
Isadora Oliveira 00264109
Gabriel Tadielo 00277942

I. Ambientes de testes

S.O.: Ubuntu 18.04
Configuração: Intel(R) Core(TM) i3-7100 CPU @ 3.90GHz - 8GB RAM
Compilador: gcc version 7.3.0 (Ubuntu 7.3.0-27ubuntu1~18.04)

S.O.: Ubuntu 16.04
Configuração: Intel(R) Core(TM) i3-370M CPU @ 2.40GHz - 4GB RAM
Compilador: gcc version 5.4.0 (Ubuntu 5.4.0-6ubuntu1~16.04.10)

II. Implementação

A. Concorrência no servidor

Para que fosse possível que múltiplos usuários fossem atendidos ao mesmo tempo pelo servidor, foi criada uma thread exclusivamente responsável por receber mensagens. Quando chega ao servidor uma mensagem de um novo cliente, é criada uma instância da classe ServerSession (para o cliente, há uma classe similar: ClientSession). Para determinar qual ServerSession deve receber uma nova mensagem, assumindo que é de uma sessão já existente, foi criado um std::map que mapeia a concatenação do IP do cliente com sua porta para sua respectiva instância de ServerSession [Imagem 1]. Assim, quando uma nova mensagem é recebida pelo servidor, ele consegue gerenciar para qual instância de ServerSession encaminhá-la de acordo com o cliente que a mandou e sua respectiva porta usada na comunicação.

```
26      std::map<std::string, std::shared_ptr<ServerSession>> _serverSessions;
27      // ServerSessions organized by ServerSession
28      // Only one ServerSession is stored here
```

[Imagem 1] Mapeamento de um cliente para sua respectiva ServerSession

B. Sincronização no acesso aos dados

Para evitar que, ao receber dados ou sinal de remoção de um arquivo, o cliente note uma mudança na sync_dir e tente fazer upload deste arquivo para o servidor, o que poderia causar a entrada em um laço infinito, foi necessário sincronizar o acesso aos dados entre a thread que monitora a pasta sync_dir e a thread que recebe dados do servidor, na função de download. Para isso, foi usado um mutex na ClientSession, nos métodos *downloadFile, deleteFile* e no laço principal da função da thread *_monitoringThread*.

C. Principais estruturas implementadas

UDPSocket

Classe essencial para aplicação. Abstrai as chamadas de um Socket com protocolo UDP, facilitando a o uso das primitivas de comunicação para o servidor e cliente.

Server

Classe que trata da inicialização e gerenciamento de ServerSessions.

Essa classe aguarda por mensagens e encaminha a mensagem para a ServerSession de destino ou cria uma nova ServerSession caso esta ainda não tenha sido inicializada.

ServerSession

Classe que trata da comunicação do servidor com um cliente.

Seu método *onSessionReadMessage* faz o tratamento do packet recebido, chamando os métodos necessários para o tipo de packet recebido.

SessionSupervisor

Classe que faz a conexão entre duas ou mais ServerSessions, para fazer sincronização entre sessões de um mesmo cliente.

Client

Classe que recebe os packets vindos do servidor e os encaminha para ClientSession. Também oferece uma interface para o usuário com métodos para manipular os arquivos no servidor e no cliente.

ClientSession

Classe que trata da comunicação do cliente com o servidor.

Assim como na ServerSession, seu método *onSessionReadMessage* faz o tratamento do packet recebido, chamando os métodos necessários para o tipo de packet recebido.

Por exemplo, um packet do tipo DATA vai ser tratado pelo método *downloadFile*, um packet do tipo DELETE vai ser tratado pelo método *deleteFile.*

FileManager

Classe que oferece métodos para manipular arquivos em um diretório específico. Usado para manipular arquivos dentro da serverSession, uma vez que o servidor guarda e manipula os dados de cada usuário em uma pasta específica.

FileMonitor

Extensão do FileManager que oferece o método *diff_dir*, que detecta diferenças ocorridas no diretório. Utilizada na *_monitoringThread* da ClientSession.

Packet

Packet é uma estrutura essencial para o funcionamento da aplicação, com o uso dela que são feitas todas as trocas de mensagens. Essa estrutura foi definida como abaixo [Imagem 2].

```
50  struct Packet{
51      // Defines what the Packet contains
52      PacketType::E type;
53      // Incremented after each message; the side that receives a packet with a given
54      // packetNum p must send an Packet ACK whose packetNum is p
55      uint32_t packetNum;
56      // The first fragment is always zero; independent of packetNum
57      uint32_t fragmentNum;
58      // Number of fragments the file was divided in
59      uint32_t totalFragments;
60      // Size of this particular Packet's data
61      uint16_t bufferLen;
62      // Size of this particular Packet's filename
63      uint16_t pathLen;
64
65      // Data itself
66      char buffer[BUFFER_MAX_SIZE];
67      char filename[FILENAME_MAX_SIZE];
68  };
69
```

[Imagem 2] Estrutura Packet

Os tipos (PacketType) possíveis para um packet são:
ACK,
DATA,
LOGIN,
EXIT,
DOWNLOAD,
DELETE.

D. Primitivas de comunicação

Utilizamos a classe UDPSocket para abstrair as primitivas de comunicação, com métodos de leitura e escrita específicos para o cliente e para o servidor.

***Packet UDPSocket::read(void)*:** função utilizada pelo servidor; como é possível receber mensagens de qualquer endereço, não é necessário passar um endereço.

Packet UDPSocket::read(sockaddr_in*): função utilizada pelo cliente; como se deve receber mensagens apenas do servidor, é necessário especificar seu endereço.
***int UDPSocket::send(Packet*, sockaddr_int*)*:** função utilizada pelo servidor; como é possível enviar mensagens para múltiplos clientes, é necessário especificar o endereço para o qual será enviada a mensagem.
***int UDPSocket::send(Packet*)*:** função utilizada pelo cliente; como o cliente deve enviar mensagens para um único endereço, ele não deve especificar o endereço para o qual enviar.

III. Dificuldades e problemas encontrados

Em questão da concorrência no servidor: inicialmente, a mesma thread responsável por receber as mensagens era responsável por tratar estas mensagens. Isso acabou sendo um problema, pois quando era necessário enviar um arquivo, esperava-se por um ACK após o envio de cada fragmento. Entretanto, como a thread que deveria receber este sinal estava bloqueada esperando pelo ACK (ao invés de estar recebendo mensagens), o servidor nunca recebia este sinal. Por causa disso foi criada uma estrutura ServerJob [Imagem 3], e o servidor passou a ter um std::vector dessa classe e um std::vector de std::thread. O primeiro std::vector representa as mensagens que devem ser tratadas, contendo também a respectiva sessão associada a esta mensagem. Além disso, as threads contidas dentro do outro std::vector ficam constantemente tentando retirar um ServerJob do primeiro std::vector para poderem tratar esta mensagem. Por existirem múltiplas threads tentando acessar um mesmo recurso (o std::vector de ServerJob), foi necessário garantir a sincronização no acesso a esse recurso.

```
12  typedef std::pair<std::shared_ptr<ServerSession>, std::shared_ptr<Packet>> ServerJob;
13
```

[Imagem 3] Estrutura ServerJob

Em questão da transferência de dados: quando a aplicação foi testada em computadores diferentes, inicialmente um arquivo podia ser corrompido por causa da rede. No caso de um ACK demorar para chegar no servidor quando este estivesse mandando o arquivo para o cliente (e vice-versa), ele manda o packet novamente, e o cliente escrevia o mesmo packet no arquivo mais de uma vez. Isso foi corrigido colocando uma condição no método *downloadFile* da classe ClientSession (e no *receiveFile* do ServerSession) verificando se o fragmento de arquivo atual já foi recebido.