UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
Sistemas Operacionais 2 - INF01151
Carolina Lange Mello - 229354
Jady Souza Feijo - 230210
Lucas Marques Silva - 275618
Vinicius Meirelles Pereira - 285637

TRABALHO PRÁTICO PARTE 1

Descrição do ambiente de testes

O projeto foi implementado em computadores com Ubuntu 14.04 LTS, os quais possuem processadores Intel Core i5 com 8GB de RAM. O compilador utilizado foi o GCC 4.8.5.

Como utilizar

- Para compilar o cliente, utilize o comando "make dropboxclient"
- Para compilar o servidor, utilize o comando "make dropboxserver"
- Para executar o servidor, utilize o comando "./dropboxserver"
- Para executar o cliente, utilize o comando "./dropboxclient"

Threads

- **Servidor**

Para que o servidor seja capaz de atender múltiplos clientes simultaneamente, ele gera uma nova thread para cada cliente que se conecta. Assim, enquanto o servidor executa existem *n+1* threads rodando simultaneamente, sendo *n* o número de clientes conectados com sucesso. A thread principal é responsável por aceitar novas conexões e criar as threads para os novos clientes. Ela também verifica se algum cliente desconectou utilizando a função *Synchronization_server::check_finished_threads()*. Para que seja possível fazer ambas as coisas, utilizamos a função *accept* de forma não-bloqueante.

Uma terceira funcionalidade da thread principal é verificar se o server deve ser fechado. Assumir que ele executa para sempre estaria errado. Por qualquer motivo que seja, pode ser necessário fechá-lo. Porém só fechar o servidor poderia interromper envios/recebimentos e causar problemas nos clientes. Portanto capturamos o sinal *SIGINT* (enviado ao processo quando se utiliza *ctrl+c* no terminal) e então sinalizamos a todas as outras threads que o servidor deve ser fechado. As threads então terminam as

operações que estão em execução e sinalizam a todos os clientes que o servidor irá fechar. Após todas as threads terminarem, o servidor fecha os sockets de todos os clientes (e o utilizado pela thread principal), realiza *join* de todas as threads e termina.

- **Cliente**

Por parte do cliente existem duas threads: a que cuida da sincronização do cliente com o servidor e a que cuida da interface com o usuário durante a execução. Isso foi feito para que o usuário possa utilizar a interface e enviar os comandos para o servidor, ao mesmo tempo em que acontece a verificação e sincronização das modificações na pasta *sync_dir* (tanto do server quanto do cliente) a cada 10 segundos.

As threads são finalizadas quando o usuário digita o comando "exit" na interface, o cliente fecha seu socket e realiza *join* das threads e termina.

Sincronização de threads

Uma das exigências do trabalho era que todas as mudanças feitas em um cliente conectado fossem refletidas em outros clientes de mesmo usuário. Conseguimos realizar este requisito. O servidor mantém um vetor com informações de todos os arquivos de cada usuário. Quando um cliente conecta com um usuário que já está conectado, ele recebe uma referência a este vetor. Logo, ambos possuirão os mesmos arquivos.

Periodicamente com um intervalo predeterminado de tempo, o cliente irá executar o comando *get_sync_dir*. Ao receber este comando, o servidor manda informações sobre todos os arquivos daquele usuário. Cabe então ao cliente identificar que arquivos são mais novos no servidor ou localmente e enviar comandos de *download* e *upload* a fim de se manter no mesmo estado que o servidor.

Sincronização de threads

- **Servidor**

No caso do servidor, a garantia de sincronização dos threads é necessária principalmente nos usuários conectados e na lista de arquivos desses clientes, e portanto são necessários dois mutex: um para cada usuário conectado e um para cada arquivo.

O mutex para cada usuário é necessário porque dois clientes (em dispositivos diferentes) podem ser do mesmo usuário e manipulam esse mesmo vetor de arquivos, porém em threads diferentes. Dessa forma, quando um cliente estiver mexendo no

vetor, outro cliente que tiver o mesmo usuário não poderá fazer isso. Por sua vez o mutex para cada arquivo não permite que conflitos ocorram quando uma thread quer baixar ou deletar um arquivo.

- **Cliente**

A garantia da sincronização de threads por parte do cliente é necessária porque um mesmo cliente pode acessar sua conta em dois dispositivos diferentes ao mesmo tempo. Então o cliente deve se comunicar com o servidor constantemente para atualizá-lo dos arquivos baixados, deletados e enviados, e também para se manter atualizado com as modificações que possam ser feitas em outro dispositivo.

A manipulação desses dados do cliente deve ser protegida por um mutex, evitando que duas threads do arquivo alterem, por exemplo, a lista de arquivos do cliente e causem problemas de persistência de arquivos no servidor.

Principais estruturas e funções implementadas

- **Servidor**

O servidor foi dividido nas seguintes classes:

- **communication_server**: cuida da parte de comunicação entre o servidor e o client
- **file_server**: representa um arquivo no servidor
- **synchronization_server**: responsável pela parte de sincronização do server com o cliente
- **connected_client**: classe que guarda os dados do cliente que já está conectado no servidor

Na **communication_server** criamos uma estrutura de mensagem. Seguimos a sugestão definida no relatório para essa estrutura, com o tipo do pacote, número de sequência, número total de fragmentos, comprimento do payload e dados do pacote. O tamanho máximo do nosso *payload* é 502, e como ele foi definido como *uint16_t* o máximo de pacotes enviados é o equivalente a 32mb (502 * 2^16).

Ainda nessa classe temos uma estrutura de argumentos para serem passados ao criar uma thread, a qual contém o objeto daquela classe, um novo socket (cada cliente conectado tem a sua conexão com o servidor), o nome de usuário, se a thread foi terminada ou não, o vetor de arquivos de usuário e o mutex desse vetor.

Na **synchronization_server** tem uma estrutura para um arquivo, que guarda quando o arquivo foi modificado pela última vez e também o nome desse arquivo.

Principais funções implementadas na **communication_server**:

- long int receive_payload(int sockfd, struct packet *pkt, int type);
- void receive_header(int sockfd, struct packet *header);
- void send_file(int sockfd, string path);
- void send_string(int sockfd, string str);
- void send_int(int sockfd, int number);
- void receive_file(int sockfd, string path);
- int create_folder(string path);
- void *receive_commands(int sockfd, string username, int *thread_finished);
- void update_user_file(string path, time_t mtime);

As funções **receive_payload**, **receive_header**, **receive_file** e **receive_commands** são responsáveis por receber os dados, o header, o arquivo e os comandos (upload, download, delete, etc) respectivamente. Enquanto isso, as funções **send_file**, **send_string**, **send_int** são responsáveis por enviar um arquivo, uma string e uma resposta do usuário (se ele já está fechado ou ainda aceita que o usuário envia comandos para o servidor).

Se uma pasta para o server daquele usuário ainda não foi criada, a **create_folder** é chamada e então é criada uma pasta com o nome de *server_sync_dir_<user>*. Colocamos o “server” na frente porque, ao rodar os testes localmente, a pasta do server e do client ficavam iguais e se sobrescreviam.

Principais funções implementadas na **file_server**:

- void start_reading();
- void done_reading();
- void start_writing();
- void done_writing();

Funções de leitura e escrita dos arquivos controladas por mutex para que, quando o usuário conectado em um dispositivo for escrever num arquivo, esse mesmo usuário em um dispositivo diferente não pode editar o arquivo. A prioridade é do escritor. Leitores podem ler o mesmo arquivo simultaneamente.

Principais funções implementadas na **synchronization_server**:

- void *accept_connections();
- void close_server();
- void check_finished_threads();

A função **accept_connections** verifica se é possível o servidor estabelecer uma conexão com o cliente, e se for possível ela valida se o número de clientes com o mesmo usuário é menor ou igual a 2 (número máximo de clientes com o mesmo usuário que podem logar ao mesmo tempo).

Para fechar o servidor basta fazer ctrl + c no terminal, após isso o servidor chama a função **close_server** que espera que os clientes mandem algum comando

(que sempre vai acontecer em no máximo 10 segundos, já que o client fica enviando o comando *get_sync_dir* para o servidor) para que ele possa avisar que está fechando, e então finaliza.

A **check_finished_threads** verifica se alguma thread dos clientes conectados foi finalizada, em caso positivo retira aquele cliente da lista de clientes conectados.

Principais funções implementadas na **connected_client**:

- void init(string username, int sockfd, int num_connections, int port, int header_size, int max_payload);
- void init(string username, int sockfd, int num_connections, int port, int header_size, int max_payload, vector<File_server> *user_files_pointer, pthread_mutex_t *user_files_mutex_pointer);
- int new_connection();
- void remove_connection();

Funções de manipulação de um cliente conectado. Quando um cliente novo se conecta a **new_connection** é chamada, já a **remove_connection** é chamada quando um cliente é desconectado. As funções **init** inicializam um cliente conectado.

- **Cliente**

O cliente foi dividido em duas classes:

- **client**: cuida da parte de interação com o usuário, criação de threads, loop de sincronização e checagem de arquivos
- **communication_client**: cuida da parte de comunicação do cliente com o servidor e envio de dados.

Com relação às estruturas, assim como na server resolvemos utilizar a sugestão definida no relatório para a estrutura de mensagem, com o tipo do pacote, número de sequência, número total de fragmentos, comprimento do payload e dados do pacote. O tamanho máximo do nosso *payload* é 502, e como ele foi definido como *uint16_t* o máximo de pacotes enviados é o equivalente a 502 * 2^16. Essa estrutura se encontra na **communication_client**.

Já na **client**, foi criada uma estrutura para um arquivo, a qual guarda quando foi feita a última modificação do deste arquivo e seu nome.

Principais funções implementadas da **client**:

- string createSyncDir();
- void userInterface();

- void check_files();

Ao ser inicializada, a client tenta se comunicar com o client (usando a **communication client**) e em seguida chama a função **createSyncDir**, verificando se já existe um diretório para aquele cliente ou não. Se não existe, cria o diretório com o nome *sync_dir_*<*username*>.

Na **userInterface** implementamos uma interface com o cliente, que é chamada por uma das threads do cliente, na qual o usuário pode realizar operações básicas do sistema via linha de comando. Mantivemos a chamada de cada comando conforme especificado na definição do trabalho, porém o funcionamento da chamada dos comandos que enviam e deletam arquivos acabou ficando um pouco diferente. No caso do *upload*, o arquivo não é enviado diretamente para o servidor, ele é copiado de seu path original para a pasta *sync_dir* (a qual é criada assim que o cliente estabelece uma conexão com o servidor) e outro método que cuida desse envio. A mesma coisa acontece com relação ao *delete*, o arquivo é apenas deletado da pasta *sync_dir*.

A **check_files** é uma função que, em uma outra thread, fica constantemente checando se houve alguma modificação na pasta *sync_dir* (arquivos deletados, enviados, atualizados) e atualiza uma lista de arquivos “vigiados”, sempre atualizando o tempo de modificação.

Principais funções implementadas da **communication_client**:

- bool connect_client_server(Client client);
- void send_command(int command);
- void send_filename(string filename);
- void send_file(string filename, string path);
- void send_mtime(time_t mtime);receive_header
- void receive_header(struct packet *_header);
- long int receive_payload(struct packet *pkt, int type);
- void receive_file(string path);
- int delete_file(string path);
- void upload_command(int command, string filename, string path, time_t mtime);
- void download_command(int command, string filename, string path, file *download_file);
- void delete_command(int command, string filename, string path);
- void list_server_command(int command);
- void get_sync_dir(int command, vector<file> *watched_files, string path);

A função **connect_client_server** inicializa devidamente a conexão com o servidor. As funções **send_command**, **send_filename**, **send_file** e **send_mtime** são responsáveis por enviar o comando, o nome do arquivo, o arquivo e o tempo de modificação do arquivo respectivamente. Já as funções **receive_header**, **receive_payload** são responsáveis por receber o header e os dados do pacote, sendo então usadas em funções como, por exemplo, **receive_file**, que recebe o arquivo. A

delete_file deleta o arquivo da pasta *sync_dir*. Essas funções utilizam o socket (*sockfd*) guardado pela **communication_client** para a comunicação com o servidor.

A função que é responsável por enviar, baixar e deletar arquivos do servidor é a **get_sync_dir**, e é ela que chama as funções que efetivamente enviam o comando de upload e delete para o servidor (**upload_command** e **delete_command**). A **get_sync_dir** fica em uma thread junto da **check_files** sendo chamada de 10 em 10 segundos para verificar se ocorreu alguma mudança no servidor que deve ser baixada para o client, ou se alguma mudança na pasta *sync_dir* (verificada pela **check_files**) deve ser enviada ao servidor. Esses comandos só são chamados nesse momento porque, como essa função é chamada constantemente, se o usuário enviasse o comando *upload* ou *delete* pela interface era possível que ocorresse algum conflito, por isso resolvemos seguir por essa abordagem.

As funções que enviam um comando para o servidor estão protegidas por mutex (*socket_mtx*), evitando que os arquivos sejam modificados por outro usuário enquanto isso.

Uso de primitivas de comunicação

As primitivas utilizadas para a comunicação entre o servidor e o client foram read e write. **Read** é uma primitiva bloqueante e recebe o *socket* usado para iniciar a comunicação, um buffer que irá receber a mensagem recebida e o tamanho desse buffer. **Write** também é bloqueante e, assim como o read, recebe o socket, um buffer com a mensagem que será enviada e o tamanho desse buffer. Como read e write não dão garantia de ler todos os bytes desejados (só garantem que pelo menos um byte será lido), foi necessário utilizar estas primitivas dentro de um laço. Elas são executadas tantas vezes quanto necessário até que leiam ou enviem todos os dados.

Exemplos de funções que utilizam essas primitivas seriam **receive_payload**, que recebe a mensagem byte a byte até receber o número de fragmentos esperado, e **send_payload**, que envia a mensagem também byte a byte.

Problemas enfrentados durante a implementação

- Modularização do código do cliente, já que não conseguimos separar adequadamente algumas funcionalidades. A ideia inicial era seguir a sugestão da definição do trabalho, em que temos um cliente com um módulo de comunicação e um módulo de sincronização. O módulo de comunicação foi implementado, porém não conseguimos separar bem a sincronização em um módulo. Apesar disso, tentamos modularizar e organizar da melhor forma possível.

- Tivemos bastante dificuldade com ponteiros.

- Enfrentamos dificuldades para lidar com pthreads em classes.
- Um dos maiores problemas enfrentados foi a necessidade de utilizar mais máquinas para testar, o que só conseguimos fazer efetivamente na UFRGS.