UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
DEPARTAMENTO DE INFORMÁTICA APLICADA

SISTEMAS OPERACIONAIS II N - INF01151

Prof. Alberto Egon Schaeffer Filho
Daniel Maia - 243672
Denyson Grellert - 243676
Felipe D'Amico - 243677
Rodrigo Dal Ri - 244936

Relatório Trabalho Prático 1

1. **Descrição do Ambiente de Teste:**
   Versão sistema operacional e distribuição: Ubuntu 16.04 LTS
   Configurações da Máquina: 4gb RAM, Core i5 3,3ghz.
   Compilador: Gcc

2. **Justificativas:**

a) Como foi implementada a concorrência no servidor:

Para cada cliente é criado dois sockets e duas threads. Uma thread e um socket são usados para escutar requisições do cliente e fazer as transferências requisitadas por este, a segunda thread e o segundo socket são usados exclusivamente para sincronização do cliente com o servidor, utilizando-os para refletir as alterações feitas pelo cliente no diretório "Sync_dir<nome de usuário>" no diretório do usuário localizado no servidor.

b) Em quais áreas do código foi necessário garantir sincronização no acesso a dados:

Nas funções do server que enviam informações do arquivo e dão update no arquivo foram usados dois semáforos. Eles devem garantir que nenhuma informação seja escrita enquanto estiver sendo feita uma leitura. Um sem_read e outro sem_write. A ideia é ter a mesma funcionalidade do problema dos escritores e leitores com prioridade aos leitores, onde somente leitores podem operar concorrentemente.

c) Estruturas e funções adicionais:

Foram adicionadas duas novas estruturas no dropboxServer.h, *client_list* e *client_request. client_list* foi a estrutura utilizada para criar a lista de usuários no servidor e seus respectivos arquivos. Enquanto a estrutura *client_request* é usada para fazer requisições do cliente para o servidor, nela existe o campo com o tipo da requisição e o campo com o nome do arquivo que somente é passado quando necessário.

No lado do cliente foram adicionadas as seguintes funções:

void client_interface() - Apresenta o menu de opções para o usuário, envia e processa requisição feita pelo usuário.

void show_files() - Envia uma requisição para o servidor para serem listados todos os arquivos contidos no diretório do cliente.

void sync_client_first() - Cria diretório de sincronização na home do cliente, inicializa monitoramento de ações nesse diretório e cria thread de sincronização no cliente.

*void *sync_thread()* - Thread de sincronização do cliente, nela é monitorado o diretório "sync_dir_<nome_de_usuário>", as alterações feitas nesse diretório são repassadas para o servidor a partir dessa função.

void initializeNotifyDescription() - Inicializa descritor que monitora o diretório "sync_dir_<nome_de_usuário>".

void delete_file_request(char* file, int socket) - Envia requisição para deletar um arquivo, essa chamada é feita quando no diretório "sync_dir_<nome_de_usuário>" é deletado um determinado arquivo.

No lado do Servidor foram adicionadas as seguintes funções:

*int initializeClient(int client_socket, char *userid, struct client *client)* - Procura se cliente está na lista de clientes do servidor, se ele estiver atualiza a estrutura da lista com o novo dispositivo conectado, se não estiver na lista, cria uma entrada para ele.

*void *client_thread (void *socket)* - Thread que roda no servidor para cada cliente, faz a chamada da função listen_client.

*void listen_client(int client_socket, char *userid)* - Executa dentro de um loop onde escuta as requisições feitas pelo cliente e as processa.

void initializeClientList() - Função chamada no início do programa, nela inicializa-se a lista de clientes a partir dos diretório dos clientes que estão no diretório do servidor.

*void send_file_info(int socket, char *userid)* - Envia para o cliente as informações de todos arquivos presentes no diretório do cliente.

*void updateFileInfo(char *userid, struct file_info file_info)* - Atualiza as informações do arquivo de um determinado cliente.

No Util foram adicionadas as seguintes funções e defines:

*void newList(struct client_list *client_list)* - Cria uma nova lista de clientes.

*void insertList(struct client_list **client_list, struct client client)* - Insere cliente na lista.

*int isEmpty(struct client_list *client_list)* - Verifica se a lista de clientes está vazia.

*int findNode(char *userid, struct client_list *client_list, struct client_list **client)* - Procura nodo do cliente na lista.

int getFileSize(FILE ptrfile)* - Calcula tamanho de um determinado arquivo.

*int commandRequest(char *request, char*file)* - Processa requisição do comando dado pelo usuário.

*void getFilename(char *pathname, char *filename)* - Extrai o nome do arquivo a partir de um pathname.

*time_t getFileModifiedTime(char *path)* - Retorna a data da última modificação de um determinado arquivo a partir do seu path.

d) Explicar o uso de diferentes primitivas de comunicação:

As primitivas de comunicação utilizadas na implementação, read e write são ambas bloqueantes, pois o padrão TCP as define assim. Como as duas são bloqueantes elas nos garantem uma comunicação síncrona o que acarreta em uma programação mais simples. Em **comunicações síncronas**, o emissor e o receptor devem estar num estado de sincronia antes da comunicação iniciar e permanecer em sincronia durante a transmissão o tempo todo. Quando dois dispositivos estão trocando dados entre si, existe um fluxo de dados entre os dois. Em qualquer transmissão de dados, o emissor e o receptor têm que possuir uma forma de extrair dados isolados ou blocos de informação.

Numa comunicação assíncrona, cada bloco de dados inclui um bloco de informação de controle, para que se conheça exatamente onde começa e termina o bloco de dados e qual a sua posição na sequência de informação transmitida. Já em uma comunicação síncrona, cada bloco de informação é transmitido e recebido num instante de tempo bem definido e conhecido pelo transmissor e receptor, ou seja, os mesmos têm que estar sincronizados. Para se manter esta sincronia, é transmitido periodicamente um bloco de informação que ajuda a manter o emissor e receptor sincronizados.

Uma comunicação assíncrona deve ser usada quando a resposta não tenha que ser entregue urgentemente, esse tipo de comunicação é adequado para situações em que não seja necessário a entrega urgente dos dados, por outro lado, não seria apropriado usar esse tipo de comunicação se você precisa de interação imediata, que é o caso do trabalho proposto. A comunicação síncrona, por via dos sends/receives bloqueantes, é ideal para quando se precisa de espontaneidade, como no caso descrito.

3. Problemas Durante a Implementação:

O programa exigiu muito conhecimento e empenho de todos participantes. Por ser um trabalho longo e que requer uma noção forte dos conceitos ensinados em aula, foi necessário o suporte de tutoriais, bibliotecas de ensino e exemplos da Internet para melhor

compreensão de como executar diversas das funcionalidades exigidas. Dentre as principais, todo o processo de implementar sockets com protocolo TCP teve que ser estudado, com uma considerável ajuda proveniente dos laboratórios realizados em aula sobre o tema. Ainda, o uso de *inotify* para o *NotifyDescription* necessitou de variadas tentativas para entender seu funcionamento e conseguir aplicá-lo para o escopo do programa.

Notavelmente importante, o uso de mutexes e semáforos são essenciais para a confiabilidade do programa, visto que ambos têm a responsabilidade de proteger as áreas consideradas críticas. Entretanto, colocar o conceito em prática foi muito mais difícil do que esperado. É uma tarefa árdua reconhecer os trechos de código que necessitam de proteção a tal ponto que pareça ser atômica e, não suficiente, deve-se estar bem atento para que não se gere deadlocks ou comportamentos desviadores de programa graças a uso indevido de *wait`s e signal`s,* por exemplo.

Por fim, o principal problema da implementação se resumiu a tarefa de garantia de sincronização. A mesma envolve uma série de funções que estão atreladas ao *sync_dir*, ou seja, a essência do trabalho Dropbox. Alterações feitas no cliente deveriam se manifestar automaticamente para o servidor, seja para envio total de arquivos, renomeações, arquivos deletados. Toda essa sincronização exige diversos testes de garantia de funcionamento, inclusive aqueles nos quais não se deve permitir escritas de mesmos clientes com mesmos usuários ao mesmo tempo, exigência descrita na especificação. Esta, sem dúvida, foi a maior dificuldade do grupo, a qual, infelizmente, não foi 100% resolvida a ponto de providenciar todas as certezas de sincronização requeridas.

Entretanto, depois de uma jornada de trabalho longo para implementar o trabalho, conclui-se que o conhecimento, tanto teórico quanto prático, adquirido no caminho foi muito gratificante, pois, por mais complexo que pareça de início o projeto, é de absoluta importância para o cenário atual de tecnologias que se relacionam a Sistemas Operacionais.