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

INF01151 – SISTEMAS OPERACIONAIS II N
SEMESTRE 2018/1
TRABALHO PRÁTICO PARTE 1: THREADS, SINCRONIZAÇÃO E COMUNICAÇÃO

Professor Alberto Egon Schaeffer Filho

Daniel Machado Nidejelski, Iago Martins, Hugo Constantinopolos, Dimek F.R, Rodrigo Oliveira Batista

Introdução

Este projeto consiste na implementação de um serviço semelhante ao Dropbox, para permitir o compartilhamento e a sincronização automática de arquivos entre diferentes dispositivos de um mesmo usuário, utilizando protocolo UDP e executando a aplicação em ambientes Linux.

Descrição de ambientes de testes

Máquina 1 Processador: AMD FX(™) 8350 Eight-Core Processor 4.00GHz Memória: 16GB Sistema Operacional: Windows 10 64bits Versão do GCC: Mingw-64 v5.0.3	Máquina 2 Processador: Intel Core (™) i5-4440 Quad-Core Processor 3.10GHz Memória: 8GB Sistema Operacional: Windows 10 64bits Versão do GCC: Mingw-64 v4.6.1
Máquina Virtual VM1 Processador: AMD FX(™) 8350 Eight-Core Processor 4.00GHz Memória: 8GB Sistema Operacional: Ubuntu 64 bits 18.04 Versão do GCC: GCC 7.3.0	Máquina Virtual VM2 Processador: Intel Core (™) i5-4440 Quad-Core Processor 3.10GHz Memória: 2GB Sistema Operacional: Ubuntu 64 bits 17.10 Versão do GCC: GCC 7.2.0

*Apesar de saber que o ambiente do sistema operacional deveria ser Linux, também testamos a aplicação em ambientes Windows.

Concorrência

A implementação da aplicação foi desenvolvida de modo a permitir que threads concorrentes executem. Cada cliente que abre uma conexão com o servidor cria uma thread, sendo possível conectar até dez clientes simultâneos, ou seja, dez threads concorrentes. Para permitir essa

possibilidade, acrescentou-se primitivas do tipo *lock* e *unlock (mutex)*, visando garantir que somente uma thread tivesse acesso à região crítica.
Para permitir a conexão de até dez clientes e alocação de recursos, usou-se o mecanismo de *semáforos*, sendo utilizada como variável global iniciada com o máximo de clientes permitidos na thread main e manipulada conforme abertura ou fechamento de conexões do cliente, permitindo a utilização de novos devices enquanto o limite não fosse antigido (caso limite alcançado, thread era bloqueada).

Sincronização

Foi necessário garantir sincronização através das primitivas de *lock e unlock* antes da sincronização do cliente com o servidor. Quando a thread do cliente precisava fazer acesso a uma região crítica (momento de sincronização dos arquivos), era necessário bloquear a thread principal para garantir a exclusão mútua.
Em relação a sincronização de arquivos (e nao de acesso), foi desenvolvido um 'módulo' a parte da implementação, através do arquivo *watcher.c (.h)*, que é nada mais que um 'observador' (como o nome sugere) que utiliza a API *inotify (UNIX)* para verificar se um arquivo foi recentemente modificado ou não.

Estrutura adicionadas/modificadas

Interface	Descrição
struct client { int devices[2]; char userid[MAXNAME]; struct file_info[MAXFILES]; int logged_in; int n_files; }Client;	*n_files - Adicionado campo de número de arquivos dentro da pasta do cliente.*

typedef struct server_info {
char ip[sizeof(DEFAULT_ADDRESS) * 2];
char folder[MAXNAME * 2];
int port;
}ServerInfo;
	Estrutura com informações do servidor
	ip - ip do servidor
	folder - pasta do servidor
	port - porta do servidor

typedef struct connection_info{ int socket_id; char client_id[MAXNAME]; char* ip; char buffer[BUFFER_SIZE]; int port; }Connection;	*Estrutura com informações da conexão estabelecida entre cliente e servidor* *socket_id - id do socket do cliente no servidor* *client_id - id do cliente* *ip - ip do cliente* *buffer - buffer auxiliar* *port - porta usada na conexão*
typedef struct client_node{ Client* client; struct client_node* next;	*Estrutura referente a lista de clientes conectados ao servidor* *client - cliente*

struct client_node* prev; }ClientNode;	*next - ponteiro para o próximo cliente da lista* *prev - ponteiro para o cliente anterior da lista*
typedef struct d_file { char path[MAXNAME]; char name[MAXNAME]; } DFile;	*Estrutura auxiliar referente a arquivos* *path - caminho do arquivo* *name - nome do arquivo*
typedef struct dir_content { char* path; struct d_file* files; int* counter; } DirContent;	*Estrutura para auxiliar com dados de conteúdo de diretórios* *path - ponteiro para caminho do diretório* *files - ponteiro para arquivo do diretório* *counter - contador auxiliar de arquivos*
typedef struct datagram{ int message_id; bool ack; char buffer[BUFFER_SIZE]; char user[MAXNAME]; }datagram;	*Estrutura referente a pacotes de dados* *message_id - id da mensagem* *ack - confirmação da mensagem* *buffer - tamanho do buffer do pacote* *user - usuario que enviou pacote*

Primitivas de comunicação

Como a implementação do trabalho deveria ser em protocolo UDP, foi necessário o uso de troca de mensagens de confirmação explícita (ACKs) entre cliente e servidor para garantir a comunicação. Esse tipo de primitiva foi utilizado diversas vezes durante o desenvolvimento: estabelecer de uma conexão entre cliente-servidor, sincronizar arquivos entre os mesmos, comandos de upload e download, login do cliente no servidor, etc.

Exemplo:

1. Envio de arquivo

Cliente envia pedido de upload de arquivo para o servidor
Servidor envia confirmação de que pode executar comando de upload (se cliente não receber ACK, upload foi negado)
Cliente informa tamanho do arquivo a ser enviado
Servidor confirma que recebeu informação de tamanho do arquivo
Cliente envia arquivo em pacotes de tamanho fixo
*Análogo para os processos de download e delete

2. Sincronização do servidor no cliente

Servidor envia pedido de sincronização
Cliente confirma pedido
Servidor envia numero de arquivos do servidor para o cliente

Cliente confirma que recebeu informação
Servidor envia nome do primeiro arquivo (ou único) a ser sincronizado
Cliente confirma que recebeu nome do arquivo
Servidor envia informação de última modificação do arquivo
Se o servidor receber o ACK e a última modificação for diferente, então sincroniza (envia última modificação para o cliente)

Como utilizar o DropBox
Executar os comandos:
make clean
make

Servidor
./server <ip>
Caso o IP não seja fornecido, 127.0.0.1 será usado como padrão. A porta utilizada default é a 9999.

Cliente
./client <usuario> <ip> <porta>
A porta deve ser a mesma, bem como o IP do servidor. Caso o usuário não exista, será adicionada uma nova pasta para o mesmo.

Problemas encontrados
A implementação de uma ferramenta como o DropBox proporciona diversos desafios. Tivemos vários problemas quanto a sincronização em ambos os lados, tanto cliente como servidor. Garantir acesso a região crítica é simples quando se usa somente dois processos e se pode mapear os acessos, mas pode ser bastante trabalhoso quando de aplica à várias threads e quando se torna muito difícil 'debugar' as mesmas.
Outro problema encontrado foi a utilização do protocolo UDP, uma vez que o mesmo não fornece garantias de entrega das mensagens, o que dificultou bastante a implementação do projeto, visto que programar as primitivas a cada interação entre cliente e servidor é, além de muito trabalhoso e cansativo, complexo na medida em que as possibilidades e combinações de situações vão crescendo junto com as funcionalidades.
Um grande problema para o grupo foi trabalhar em conjunto e partilhar tarefas no desenvolvimento. O grupo teve grandes dificuldades em desenvolver como um time e em sintonia, pois não havia entendimento em relação à prazos e os horários de cada colega dificultou a implementação.
Como sugestão, sugere-se ao Professor modificar a dinâmica de desenvolvimento para que a entrega seja por etapas, conforme é feito em outras disciplinas. Ao invés de colocar um prazo muito grande e longe da data de especificação do trabalho, diminuir os prazos e colocar datas para entrega de etapas menores e incrementais, com auxílio do Professor no entendimento dos

problemas enfrentados e dicas para solucioná-los. Achamos que essa é uma melhor maneira de absorver os conhecimentos adquiridos em aula, uma vez que da forma atual o trabalho acaba sendo extremamente extenso e cansativo, tornando um pouco enfadonho desenvolver até o final e manter o mesmo nível de qualidade do começo. Além disso, trabalhar em grupos muito grandes de 4 ou 5 pessoas é muito mais complexo do que duplas ou trios, visto que alguns acabam por executar menos ou mais tarefas que outros.