UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA

FRANCISCO KNEBEL
LUCIANO ZANCAN

syncBox:
THREADS, SINCRONIZAÇÃO E COMUNICAÇÃO

Monografia apresentada como relatório de trabalho na disciplina de Sistemas Operacionais II N.

Orientador: Prof. Dr. Alberto Egon Schaeffer Filho

Porto Alegre
2017

Introdução

Este projeto consiste na implementação de um serviço de sincronização de arquivos semelhante à plataforma Dropbox, implementado utilizando a linguagem C. O serviço desenvolvido no projeto recebeu a nomenclatura de *syncBox*, e seu desenvolvimento foi efetuado utilizando a ferramenta *Git* para versionamento e armazenamento. Após a entrega final, o projeto estará integralmente e publicamente disponível na plataforma GitHub, em *https://github.com/FranciscoKnebel/syncBox*.

A tecnologia do projeto se utilizou de diversas ferramentas para o desenvolvimento, incluindo, mas não limitada, de Sockets TCP do Unix para comunicação e envio de mensagens, POSIX threads para implementação de concorrência e a API inotify para monitoramento do sistema de arquivos, afim de verificar modificações na pasta do usuário.

Figura 1. Log de inicialização do servidor, esperando por conexões.

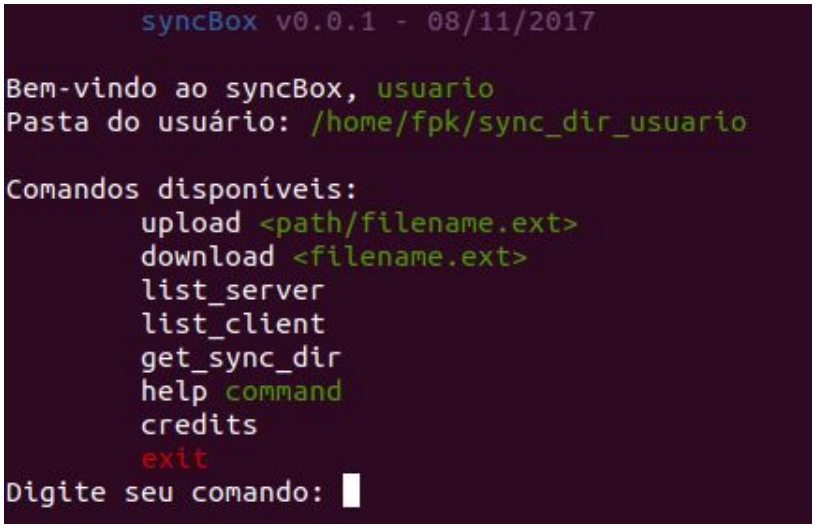

Figura 2. Interface de comando visível pelo cliente.

Descrição do Ambiente de Teste

A aplicação foi desenvolvida e testada em múltiplos ambientes separados, com as seguintes configurações, sendo que a maior parte foi utilizando os ambientes 1 e 2. O ambiente 3 é utilizado apenas para automação de testes, utilizando uma ferramenta de integração contínua. O **syncBox** foi testado em diversas situações: apenas um computador, em mais de um computador na mesma rede e em mais de um computador utilizando acesso remoto pela internet.

Ambiente 1

Ubuntu 17.10 Artful Aardvark, kernel 4.13.0
Processador Intel i5-7600K @ 3.80 GHz x4 DDR4 2x8 GB @ 3000 MHz RAM
Compilado com GCC v7.2.0

Ambiente 2 - Asus f55c-sx011h

Ubuntu 16.04.4 Xenial Xerus, kernel 4.4.0
Processador Intel i3-3110 @ 2.4 GHz, 2 nucleos, 4 threads DDR3 1600/1333 MHz RAM
Compilado com GCC v5.4.0

Ambiente 3 - Travis CI

Ubuntu 14.04 Trusty, Kernel 4.4.0
Sem informações de máquina, utilizado apenas para automação de testes.
Compilado com GCC v5.4.1

Execução

Compilação

O projeto foi desenvolvido utilizando GCC para processos de compilação. Todas as instruções utilizadas estão presentes no Makefile do projeto.

Para sua utilização, é apenas necessário abrir um terminal no diretório do projeto e executar *make*. Esse comando irá efetuar a compilação dos objetos utilitários, seguindo pela compilação de cada módulo da aplicação *Client*, encerrando com a compilação do programa que será utilizado pelos usuários. Após o encerramento da parte do cliente, o equivalente acontece com os arquivos respectivos do servidor, gerando o executável do servidor. É possível executar cada uma dessas tarefas individualmente, caso necessário.

Make test inclui alguns testes de módulos, comando utilizado automaticamente, a cada novo commit no repositório remoto, pelo Travis CI, ferramenta utilizada para garantir integração contínua do projeto. Além de executar os testes, ele também irá indicar caso houver algum erro de compilação no projeto. *Make clean* irá remover todos os arquivos gerados pelo processo de compilação.

Servidor

A aplicação *Servidor* não necessita interação por usuários, apenas sendo necessário iniciar o seu executável. Ele recebe dois parâmetros para sua execução, em ordem: o IP onde a aplicação está executando e a porta onde ela será acessível. Caso um desses valores não seja informado, os valores padrão ‘127.0.0.1’ e ‘3000’ serão utilizados. Após o início da execução, o servidor estará pronto para receber conexões com até *MAX_CLIENTS* clientes, garantidos pelo uso de um semáforo.

Cliente

Cliente requer três parâmetros para execução, em ordem: nome do usuário, IP e porta do servidor. Após o início da execução, a aplicação efetuará comunicação com o servidor, e após essa conexão estiver estabelecida, a interface de comandos estará disponível para o usuário. Caso o usuário tenha dúvida sobre o funcionamento de algum comando, uma das opções disponíveis é help, que aceita um comando como atributo. Todas as informações sobre o comando, de acordo com especificação, estão presentes neste guia.

Justificativas gerais de desenvolvimento

Como foi implementada a concorrência no servidor

Para a implementação de concorrência, foi decidido utilizar a biblioteca *pthreads* devido à facilidade de manuseio e conhecimento dos desenvolvedores. Para auxílio no gerenciamento de clientes, o armazenamento dessas informações foi implementada utilizando uma lista simples encadeada. Para limitar o número de usuários conectados ao mesmo tempo, foi utilizado um semáforo no servidor, que é inicializado com a constante MAX_CLIENTS, e é decrementado a cada nova conexão e decrementado em cada desconexão. Caso atinja a saturação, qualquer cliente que tente se conectar é bloqueado, assim o servidor espera alguém desconectar para incrementar o semáforo e conectar o cliente que solicitou e foi bloqueado.

O servidor principal está em loop pela busca de conexões de novos clientes. Assim que um novo cliente efetua uma nova tentativa de conexão através da primitiva *accept*, o servidor decrementa o semáforo e cria uma nova thread, utilizando a função denominada **clientThread**, que se encarrega de testar a conexão, checando se o cliente já está conectado ou não, além de verificar o número de dispositivos conectados para o mesmo cliente, pois há um limite sobre o mesmo. Caso ultrapasse os parâmetros, dados segundo a especificação (dois dispositivos simultaneamente), esta thread encerra a conexão com esse dispositivo, juntamente com a thread criada. Se, entretanto, a conexão for bem sucedida, a thread continua sua execução normal até que o cliente desejar efetuar a desconexão com o servidor, quando então o semáforo é liberado.

Sincronização no acesso a dados

Existem funções de upload, download e delete, sobre suas respectivas funcionalidades (envio de arquivos ao servidor, recebimento de arquivos do servidor e exclusão de arquivos no servidor). Quando um cliente modifica, cria ou deleta um arquivo em seu diretório local, este mesmo é sincronizado com o servidor a partir destas 3 funções.

A *struct client* foi modificada a partir da especificação, sendo adicionada a esta um array de mutexes de tamanho MAXFILES. Ao criar um novo cliente para a lista encadeada, a função de criação já inicializa todos os mutexes. Então, a cada requisição de upload, download ou delete, o servidor chama uma função que procura o índice do arquivo no array de *structs file_info* que o cliente possui. Este índice retornado é utilizado para bloquear o mutex específico para aquele índice do arquivo relacionado ao usuário. Assim, garante-se que um mesmo cliente conectado em dois dispositivos não consegue realizar upload, download ou delete no mesmo arquivo ao mesmo tempo. Isso garante integridade, pois evita acesso incorreto a um arquivo que já estava sendo acessado. Para automatizar o envio, ao iniciar a execução do programa cliente, é criado uma thread que efetua o monitoramento da pasta do usuário. Quando um arquivo é criado/modificado/removido, o programa irá automaticamente efetuar as operações necessárias para manter a sincronização desses arquivos com o servidor.

Estruturas e funções adicionais que você implementou

Partindo da estrutura mínima sugerida na especificação, desenvolvemos a plataforma conforme a estrutura seguinte:

- syncBox
    - bin - Arquivos .o gerados durante a compilação;
    - dst - Arquivos executáveis da aplicação Cliente e Servidor;
    - headers - Headers com estruturas de dados, constantes e outras definições;
    - src - Código-fonte do projeto (arquivos .c);
    - Makefile - Makefile com Make e Clean;
    - test - Código-fonte de testes sobre módulos do projeto.

O diretório *bin* contém os arquivos objeto gerados pelos módulos compilados, que são unidos para serem utilizados nos executáveis principais, que são gerados em *dst*. Em *headers*, estão contidos todos arquivos de cabeçalho utilizados, com as definições das estruturas e funções exportadas entre os módulos.

A pasta *src* que contém a lógica em si, o código da aplicação. Ela contém os arquivos *dropboxClient.c*, *dropboxServer.c* e *dropboxUtil.c*, esperados pela definição, além dos demais arquivos que regem a aplicação. Dentro foram criados diretórios para separar a lógica da aplicação do cliente e do servidor. A pasta *test* incluí código de teste para módulos criados na aplicação. Na pasta raíz, também se encontram outros arquivos de configuração. *Makefile* contém as instruções de compilação, dependência entre módulos e limpeza do projeto. *README.md* é um sumário do repositório, criado por razões informativas.

Estruturas adicionais

- *struct d_file (dropboxUtil.h)*, que contém nomes e diretórios dos arquivos. Esta estrutura é intermediária à file_info da especificação.
- *struct dir_content(dropboxUtil.h)*, que engloba a d_file, tendo um ponteiro para ela e o número de arquivos no diretório.
- *struct client_list (dropboxServer.h)*, que contém a lista encadeada de clientes. Possui um ponteiro para o próximo cliente e informações do cliente atual.
- *struct server_info (dropboxServer.h)*, que contém informações do servidor como IP, diretório do servidor e porta.
- *struct connection_info (dropboxServer.h)*, que contém informações de ip e id do socket. É utilizada para passar informações sobre o ip e socket do cliente para a nova thread de concorrência, quando criada.
- struct client (dropboxServer.h) modificada para conter um array de mutex de tamanho MAXFILES, que é utilizado para garantir que um mesmo cliente em dois dispositivos não acesse o mesmo arquivo no mesmo tempo.
- *struct user_info (dropboxClient.h)*, que contém informação de nome e pasta do usuário.

Funções adicionais do Cliente

- Cabeçalho de *send_file* modificado para incluir uma flag, com função de indica se deve imprimir ou não a resposta do envio do arquivo.
- Cabeçalho de *get_file* modificado para receber um caminho alternativo, para indicar um diretório diferente da pasta definida para o cliente dentro do dispositivo. Esse argumento é utilizado no comando *download*.
- Função adicionada: *list_server*. Utilizada pelo comando de mesmo nome.

Funções de interface

- *void print_commands()* - Imprime os comandos disponíveis na tela;
- *void show_intro_message()* - Imprime informações de boas vindas na tela;
- *int is_not_exit_command(char* command)* - compara command com o comando exit;
- *int is_valid_command(char* command)* - Testa se o comando recebido pelo usuário é válido;
- *void callCommand(char* command, char* attribute, int check)* - Função que recebe o comando enviado pelo usuário e chama a função desejada;
- *int parseCommand(char* command, char* commandName, char* commandAttrib)* - efetua o parsing do comando informado pelo usuário. Retorna em *commandName* e *commandAttrib* o nome do comando e o argumento passado juntamente. O retorno da função informa se o comando do usuário inclui ou não um argumento, indicando erro caso um argumento seja esperado para esse comando.
- *void show_client_interface()* - Após a inicialização da aplicação e a conexão com o servidor, essa função será chamada. Ela é a responsável por mostrar a interface para o usuário, além da leitura do comando desejado. Essa função encerra apenas no encerramento do programa.

Funções de comando

Após o usuário informar que comando ele quer executar, uma dessas funções será chamada. Apenas os comandos *upload* e *download* recebem argumentos, como exposto na definição.

- *void command_upload(char* path)* - chama a função send_file;
- *void command_download(char* path)* - chama a função get_file, utilizando o diretório local como segundo argumento, definindo assim onde o download de *path* será efetuado;
- *void command_listserver()* - chama a função list_server, mostrando os arquivos sincronizados com o servidor;
- *void command_listclient()* - chama a função print_dir_content, mostrando o conteúdo da pasta local do cliente;
- *void command_getsyncdir()* - cria a pasta do usuário e efetua um pedido de sincronização dos arquivos locais;
- *void command_credits()* - imprime na tela as informações dos créditos;

- *void command_clear()* - limpa a tela dos usuários;
- *void command_help()* - chama as funções de ajuda.

Funções de ajuda

- *void help_*()* - imprime informações de funcionamento de um comando qualquer, disponível para o cliente. A lista de todas as opções disponíveis são:
    - upload
    - download
    - list_server
    - list_client
    - get_sync_dir
    - credits
    - clear
    - exit
    - help
- *void help_undef(char* command)* - utilizada quando o usuário informa um comando inválido para a função de ajuda.

Funções de sincronização

- *void synchronize_local(int sockid)* - sincroniza arquivos locais do cliente com o servidor;
- *void synchronize_server(int sockid)* - sincroniza arquivos do servidor com o diretório local no dispositivo do cliente.

Funções de monitoramento

- *void *watcher_thread(void* ptr_path)* - cria thread de monitoramento para o diretório do cliente. Responsável pelo trabalho de sincronização durante a execução da aplicação.

Funções adicionais do Servidor

- Cabeçalho de *sync_server* modificado para receber o id do socket e um ponteiro para a estrutura do cliente, pois de acordo com a especificação o sync_server não receberia nenhum parâmetro, mas como o servidor é multi thread, é complicado deixar alguma variável global para ser utilizada por este fim, pois necessitaria o uso de mutexes, semáforo ou monitores, o que dificultaria a concorrência do servidor;
- Cabeçalho de *receive_file* modificado para receber o id do socket, pois da mesma forma que o sync_server, necessita de um ID de socket, o que necessitaria de algum tratamento e prejudicaria a concorrência do servidor;
- Cabeçalho de *send_file* modificado para receber o id do socket, pois há nesta função o mesmo problema que as duas anteriores (receive_file e sync_server), em que a falta de um parâmetro ID do socket poderia impedir ou prejudicar a concorrência do servidor;
- Função *void* clientThread(void* connection_struct)* adicionada, engloba a execução de um novo processo a cada novo cliente conectado, é chamada pela pthread_create;
- Função *void parseArguments(int argc, char *argv[], char* address, int* port)*, que valida os argumentos passados na inicialização do servidor. Assim, o servidor pode ser inicializado somente executando-o, ou pode ser especificado endereço e porta na inicialização.

Funções da lista encadeada de clientes

- *ClientList* newClient(char* userid, int socket, ClientList* client_list)* - adiciona um novo cliente à lista encadeada de clientes, inicializando o cliente também com seus diretórios e arquivos no servidor. Retorna a nova lista encadeada (client_list recebida, com novo cliente adicionado);
- *Client* searchClient(char* userId, ClientList* client_list)* - procura por um cliente na lista encadeada de clientes. Retorna um ponteiro para o cliente caso encontre, ou NULL caso não encontre;
- *ClientList* removeClient(Client* client, ClientList* client_list)* - remove um cliente da lista encadeada de clientes client_list recebida. Caso a lista possua mais de um cliente, remove e retorna a nova lista, ou remove e retorna NULL caso possua apenas um cliente;
- *int addDevice(Client* client, int socket)* - adiciona um dispositivo a um cliente;
- *int removeDevice(Client* client, int device, ClientList* client_list)* - remove um dispositivo de um cliente, e retorna o índice do dispositivo removido, ou -1 caso o client não exista;
- *ClientList* check_login_status(Client* client, ClientList* client_list)* - testa se um cliente da lista client_list está conectado em algum dispositivo. Caso não esteja conectado em nenhum dispositivo, chama removeClient e seta logged_in para 0. Caso esteja conectado em algum dispositivo retorna a mesma client_list recebida.

Funções de comandos

- *void select_commands(int socket, char buffer[], Client* client)* - quando recebe um comando de um cliente, esta função é chamada para direcionar qual funcionalidade será executada, de acordo com o comando passado pelo usuário;
- *void upload(int socket, Client* client)* - upload de arquivos pelo usuário, ou seja, recebe um arquivo do cliente;
- *void download(int socket, Client* client)* - download de arquivos pelo usuário, ou seja, envia um arquivo ao cliente;
- *void list_server(int socket, Client* client)* - envia informações sobre o diretório do usuário no servidor, para o cliente;
- *void delete(int socket, Client* client)* - remoção de um arquivo pelo usuário;
- *void sync_dir(int socket, Client* client)* - Sincroniza a pasta local do usuário de acordo com o servidor, ou seja, sincroniza o cliente com os arquivos remotos.

Funções de sincronização

- *void synchronize_client(int sockid_sync, Client* client_sync)* - sincroniza os arquivos do usuário com os do servidor.
- *void synchronize_server(int sockid_sync, Client* client_sync)* - sincroniza os arquivos do servidor com os arquivos do usuário.

Utilitários (cliente e servidor)

Diretórios

- *int get_dir_content(char * path, struct d_file files[], int* counter)* - retorna em *files* o conteúdo do diretório *path*. *Counter* retorna a quantidade de arquivos encontrados;
- *int get_all_entries(char * path, struct d_file files[])* - Wrapper de *get_dir_content*, útil no caso em que a quantidade de arquivos é irrelevante;
- int print_dir_content(char * path) - Outro wrapper de *get_dir_content*, onde o conteúdo do diretório não precisa ser armazenado. Função serve para imprimir o conteúdo do diretório *path*;
- *int get_dir_file_info(char * path, FileInfo files[])* - Mais um wrapper de *get_dir_content*, mas que utiliza *getFileModifiedTime* e *getFileExtension* para estender as informações retiradas dos arquivos encontrados no diretório *path*.

Arquivos

- *void getFileModifiedTime(char *path, char* last_modified)* - retorna em *last_modified* uma string contendo informações sobre o tempo da última modificação sobre o arquivo *path;*
- *void getFileExtension(const char *filename, char* extension)* - retorna em *extension* a extensão do arquivo *filename*;
- *int getFileSize(char *path)* - retorna o tamanho do arquivo *path*;
- *void getLastStringElement(char filename[], char* string, const char *separator)* - retorna em string a última substring de *filename*, de acordo com seu separador *separator* (ou seja, com filename sendo *path/arquivos/texto.txt*, retornará *texto.txt* caso *separator* for igual a “/”);
- *time_t getTime(char* last_modified)* - parsing da string recebida na função *getFileModifiedTime* para uma estrutura padrão *time_t*;
- *int older_file(char* last_modified_file_1, char* last_modified_file_2)* - retorna um inteiro para indicar qual dos dois arquivos é mais antigo que outro. O retorno 1 indica que o segundo arquivo é mais velho que o primeiro, e por consequente, 0 indica o oposto;
- int fileExists(char* pathname) - retorna um inteiro indicando se o arquivo em *pathname* existe ou não;
- int fileExists_stat(char* pathname, struct stat* st) - retorna a struct stat do arquivo em *pathname*, mantendo a mesma funcionalidade de *fileExists*;
- int getFileIndex(char* filename, FileInfo file_info[]) - Percorre a estrutura *file_info* procurando por um arquivo *filename*, retornando o índice encontrado.

Usuário

- *char* getUserName()* - retorna o nome do usuário logado na máquina (ex: *user*);
- *char* getUserHome()* - retorna a pasta do usuário logado na máquina (ex: */home/user/*).

Uso das diferentes primitivas de comunicação

Para realizar a comunicação entre os processos clientes e o processo servidor foi utilizado a API de Sockets do UNIX, com nomeação indireta através do IP da rede, e para garantir confiabilidade na entrega e recepção foi utilizado o tipo de comunicação TCP (Transmission Control Protocol), o qual provê uma comunicação do tipo *at-least-once* e um processamento do tipo *exactly-one*.

Utilizar sockets TCP permite a comunicação através de primitivas send e receive, onde é utilizado o ID do socket a ser recebido ou enviado as informações. Como o servidor é multi thread, cada thread possui seu id, que se comunica diretamente com o cliente conectado. O cliente por sua vez é um processo separado, e contém seu próprio id também, que se comunica diretamente com o id da thread servidor específica para ele.

Toda a comunicação do servidor baseou-se nessas duas primitivas (send e receive). Como são bloqueantes, é necessário sincronizar cada write no servidor com read no cliente e vice-versa para não haver problemas.

Desafios de desenvolvimento

Houve problemas de obtenção do id do socket pelo servidor, pois suas funções não recebiam informações sobre o socket utilizado pelo usuário. Para resolver foi necessário modificar os cabeçalhos para receber tanto o id do socket nas 3 funções mínimas do servidor, quanto um ponteiro para o cliente no caso da função *sync_server*.

Para garantir que um cliente em mais de um dispositivo não prejudique arquivos que ele pode estar modificando de outro dispositivo, foi necessário arranjar um meio de não permitir mais de um cliente modificando um mesmo arquivo de uma só vez. A solução encontrada foi criar um array de mutexes na struct do cliente, que garante a exclusão mútua sobre cada um dos arquivos de cada usuário.

Para impedir que mais de N clientes conectem-se ao servidor, foi necessário arranjar um meio de limitar o acesso. A solução encontrada foi a utilização de um semáforo inicializado me N, que decrementa após o *accept* de cada nova conexão no servidor, o que bloquea qualquer nova tentativa de conexão após o limite imposto, e só libera após algum cliente desconectar.

Conclusão

O desenvolvimento do **syncBox** requereu muito esforço e tempo, mas a sua primeira etapa foi concluída com sucesso. Após o desenvolvimento seguindo a especificação, houve um processo extenso de debugging para resolver os diversos problemas que surgiram com uma aplicação nessa escala, que logicamente requereu que muitas implementações fossem repensadas e refatoradas.

Foram encontradas alguns bugs bizantinos na aplicação, relacionados às funcionalidades de sincronia. Em certas ocasiões, ocorrem alguns defeitos na ordem de leitura sobre o socket, que acaba gerando arquivos inválidos. Como o aparecimento é instável e a origem não foi descoberta, a solução ainda não foi implementada. Esperamos conseguir encontrar e resolver esse problema em específico para a segunda etapa do trabalho.

O **syncBox** foi testado em diversas situações: apenas um computador, em mais de um computador na mesma rede e em mais de um computador utilizando acesso remoto pela internet, e em todos os casos os objetivos seguiam a definição imposta, tendo resultado satisfatório, mas queremos expandir cada vez mais a aplicação.