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 2

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.
Também utilizamos os computadores do laboratório de informática para os testes.

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 backup, utilize o comando:
  “/dropboxServer <active_server_ip_address> <active_server_port>”
- Para executar o cliente, utilize o comando:
  "./dropboxClient <username> <server_ip_address> <port>"

Correções da Parte 1

Quando apresentamos a parte 1 do trabalho, tivemos dificuldades com o envio de arquivos. Quanto maior o arquivo, menor chance de o envio ocorrer corretamente. O que acontecia era que ao receber um header, não se avançava a posição do buffer. Era assumido que o header era recebido inteiro de uma vez, o que não necessariamente ocorre.

Implementamos outra parte do programa que beneficiaria a primeira parte do trabalho. Antes, se um cliente conectasse, enviasse arquivos ao servidor e desconectasse, o servidor ia perder o registro dos arquivos (porque o client desconectou). Agora, um arquivo .txt é utilizado para armazenar informações dos arquivos. O server pode ser desligado e voltar e os clientes podem desconectar e conectar. Tudo isso sem que o server perca registro dos arquivos.

Uma reclamação do professor foi que o retorno de list_server e list_client no terminal não estavam legíveis. Sendo assim, trocamos a listagem dos arquivos de lado a lado para um abaixo do outro.

Na parte 1, o programa detectava caso fosse inserido ctrl+c no terminal para fechar o servidor de forma correta. Como a parte 2 depende de utilizar ctrl+c para parar o servidor abruptamente, esta função foi desativada.

Replicação Passiva

A ideia da replicação passiva é que, ao haver falha no servidor principal, servidores backups possam assumir seu papel de forma transparente para o usuário e manter o gerenciamento de arquivos funcionando. Para que isso ocorra, os backups devem estar sempre sincronizados com o servidor, já que são uma “réplica” do servidor, e verificam de tempo em tempo se o servidor permanece vivo. Se o servidor morreu, ele não responde a verificação dos backups (*timeout*) e os backups iniciam uma eleição para saber quem vai assumir o papel de servidor principal.

Uma modificação que precisou ser feita para que o *timeout* no recebimento de mensagens funcionasse corretamente foi deixar o **read** como não-bloqueante, pois sendo bloqueante ficava esperando uma resposta para sempre.

A seguir serão citadas novas estruturas e métodos criados para explicar como a replicação passiva foi implementada.

- **Servidor**

  Algumas modificações foram necessárias no servidor para a replicação passiva. Agora além de esperar pela conexão de clientes, o servidor fica à espera de backups para se conectar com eles. Ele espera esta conexão em dois sockets, não apenas um. Existem duas conexões entre backup e servidor. Uma delas é responsável pelo envio de arquivos e comandos do servidor ao backup. A outra é utilizada pelo servidor para indicar ao backup que ainda está vivo. Quando este envio dá timeout no backup ou quando o socket é fechado pelo servidor, o backup assume que o servidor morreu.

  Quando um cliente conecta ao servidor, o servidor manda todos os arquivos deste usuário já existentes no servidor aos backup com a função **send_all_files**.

  O servidor possui um vetor com os IPs de todos os backups. Ela será enviada a qualquer usuário ou backup que se conecte a ele.

  O envio de comandos e arquivos do servidor aos backups é feito pela thread de cada client no backup, protegido por exclusão mútua (somente uma thread pode se comunicar com o backup de cada vez). Sempre que o servidor recebe um arquivo, manda aos backups. Sempre que o servidor deleta um arquivo, envia o comando de deleção aos backups.

- **Backup**

  Para a implementação da funcionalidade dos backups, criamos uma nova classe chamada **backup**. Ela é inicializada quando, ao chamar o comando *dropboxserver* no terminal, são passados os argumentos IP e porta. O IP passado é

referente ao IP do servidor com o qual o backup irá se conectar, assim como a porta é a porta em que acontecerá a conexão do servidor com o backup.

Essa classe gerencia a comunicação do backup com o servidor, a sincronização de arquivos dos backups com os arquivos do servidor e a comunicação dos backups entre si para quando ocorre a eleição (e para que, após a eleição, os backups possam se reconectar).

Cada backup possui uma lista de *backup_info,* que é uma estrutura onde são armazenados os IPs, sockets e ids dos backups conectados a esse backup, e ela é usada tanto para reconectar os backups entre si quanto para enviar as mensagens no momento da eleição.

Principais funções implementadas no **backup**:

- void check_server(int* main_check_sockfd, int *server_died);
- void connect_backup(int* main_check_sockfd, int *server_died);
- int connect_backup_to_main();
- int connect_backup_to_backup(string ip);
- int connect_chk_server();
- void close_backup(int main_check_sockfd);
- string election(struct backup_info this_backup);
- void receive_commands(int sockfd, int *server_died);
- string create_user_folder(string username);
- void mtime_to_file(string path, time_t mtime, string username);
- void delete_mtime_from_file(string path, string username);

A função **connect_backup_to_main** inicializa a conexão do servidor principal com uma das réplicas quando ela é inicializada. Assim que essa conexão é estabelecida, o servidor principal envia para o backup o número de backups que já foram conectados ao servidor, e se existir algum envia as informações de cada um deles para o backup. O backup então chama a função **connect_backup_to_backup** que conecta esse backup em questão com todos os backups que já estão conectados no servidor, e guarda suas informações (ip, socket e id) na lista de *backup_info*.

Depois disso o backup guarda seu próprio id em uma variável. O id do backup é decidido pela sua ordem de conexão com o servidor, ou seja, o primeiro backup que se conectar com o servidor vai ter id igual a 0, o segundo vai ter id igual a 1. A prioridade para virar o servidor principal é sempre do backup de maior id, então o último que se conectar vai ter prioridade.

Duas threads são abertas em seguida: a que fica esperando por conexões de próximos backups que podem ser inicializados (que chama a função **connect_backups**) e a thread que verifica se o servidor ainda está vivo (que chama o método **check_server**).

Em seguida é chamada a função **receive_commands**, que é responsável por ficar recebendo os arquivos (e comandos de deleção) do server, e dessa forma manter os backups atualizados.

No momento que o servidor morre a função **election** é chamada, a qual retorna o IP do próximo líder. Se o backup em questão for eleito o próximo líder principal, ele assume as características de um servidor inicializando a classe **Syncronization_server** e os backups e os usuários se reconectam a ele.

- **Cliente**

  Assim como no servidor, modificações foram feitas no cliente para que, no caso do servidor parar de funcionar, o cliente possa se conectar com as réplicas sem que o usuário perceba.

  Ao se conectar com o servidor principal, o cliente recebe os IPs de todos os backups conectados com servidor, então quando o servidor principal cai, o cliente passa a tentar a se conectar com todos os backups até que consiga se conectar com um deles (o que ganhou a eleição).

  Da forma que foi feito, para que o cliente saiba da existência de um backup e possa se conectar com ele quando o servidor cair, esse backup deve ter sido conectado ao servidor antes do usuário se conectar.

  Quando o servidor cai, o cliente tenta se conectar com um dos backups em seu vetor de backups. Se não tem sucesso, tenta com o próximo. Se não consegue conectar com algum, espera 3 segundos (para não sobrecarregar a rede com mensagens) e tenta novamente desde o início. O cliente não sabe quem ganhou a eleição, mas como tenta se conectar com todos, sempre consegue conectar.

  No cliente, existe uma thread que recebe input do usuário através do comando getline. Como esta função é bloqueante, após o cliente detectar que o servidor morreu, qualquer string precisa ser inserida no terminal. Após receber a string, continua normalmente o funcionamento. Isto poderia ser mudado, mas consideramos que o investimento de tempo era muito grande para as vantagens que a implementação trazia. Ainda mais considerando que funcionalidades importantes ainda não haviam sido implementadas.

Eleição de Líder

O algoritmo de Eleição de Líder foi feito baseado no Algoritmo Valentão (Bully), e foi implementado no método **“election”**, na classe **backup**. Os backups possuem uma thread separada para se comunicarem entre si, e dessa forma podem se enviar mensagens enquanto acontece a eleição, a qual é iniciada quando os backups percebem que o servidor morreu (por *timeout*). No caso do Algoritmo Valentão, a eleição pode ser iniciada quando o backup percebe que o servidor morreu ou quando recebe uma mensagem de outro backup, mas no caso do nosso algoritmo todos os backups sempre percebem que o servidor morre e então chamam o método para iniciar a eleição.

Quando a eleição é iniciada, o backup recebe uma mensagem de eleição dos backups com id menor que o seu, e responde a esses backups (se ainda estiver ativo), após isso, o backup envia uma mensagem de eleição para os backups com id maior que o seu e espera por uma resposta. Por fim, cada um dos backups verifica se recebeu mensagem de algum com id maior que o seu: se recebeu, o maior dos ids maiores é o líder. Caso contrário se torna o líder e os demais se conectam a ele como o novo servidor principal.

Problemas enfrentados durante a implementação

- Um dos maiores problemas enfrentados foi a necessidade de utilizar mais máquinas para testar, o que só conseguimos fazer efetivamente na UFRGS. Além disso,

somos quatro componentes no grupo e possuímos cada um uma conta no INF. Sendo assim, raramente conseguimos utilizar as 5 máquinas necessárias para a apresentação.

- Durante grande parte da implementação da parte 2, os backups não conseguiam receber arquivos. Isto ocorreu porque o socket no backup para a comunicação com o servidor estava sendo iniciado como não-bloqueante sem necessidade. Somente o socket que recebe as mensagens “alive” do servidor precisa ser não-bloqueante.
- O getline bloqueante foi um problema que não conseguimos resolver.