Dropbox - Parte II

Augusto Bennemann, Fabrício Martins Mazzola e Txai Wieser

1 Ambiente de desenvolvimento e teste

O projeto foi desenvolvido utilizando um ambiente com as seguintes caracteristicas:

- Configuração da máquina:
  - Processador(es): Intel Core i5-2520M
  - Memória: 2x Kingston 4GB 1333Mhz DDR3
- Sistema operacional:
  - Distribuição: Debian GNU/Linux
  - Versão: Stretch (testing) Kernel 4.9.0-3-amd64
- Software suporte:
  - Compiladores (versões): gcc 6.3.0 20170516

2 Descrições e justificativas

2.1 Sincronização de relógios

Para a sincronização de relógios, utilizamos um algoritmo de sincronização externa bastante simples, o algoritmo de Christian. Seu funcionamento é, basicamente, fazer uma requisição de horário para um servidor de tempo (T_s) e, após receber a resposta, somar ao horário recebido uma estimativa do tempo de transferência, obtida dividindo-se o RTT (diferença entre o tempo de envio (T_0) da requisição e a chegada da resposta (T_1)) pela metade. O tempo obtido no cliente pode ser mostrado pela fórmula $T_c = T_s + \frac{(T_1 - T_0)}{2}$.

Essa estimativa não é muito precisa pois não necessariamente os tempos de ida e de volta são exatamente o mesmo - eles podem variar de acordo com o congestionamento da rede e as rotas utilizadas. Por isso, o algoritmo de Christian é mais indicado para redes locais, nas quais os tempos de transmissão são menos variáveis e menores em comparação a redes de longa distância. Ademais, outra limitação desse algoritmo é o uso de um único servidor de tempo dentro de uma rede local, consistindo em um ponto único de falha.

Esse algoritmo foi utilizado no trabalho implementado para garantir a sincronização de relógios entre cliente e servidor, de modo a permitir que todos os arquivos que são enviados ao servidor possuam o horário correto e sincronizado do servidor, independente do horário da máquina local. A cada vez que um usuário faz o envio de arquivo pelo terminal ou arrasta um arquivo para a sua pasta de *sync_ dir*, a função *getLogicalTime()*, responsável pela sincronização, é executada e o timestamp do arquivo ajustado.

2.2 Segurança

A primeira versão do Dropbox implementada carecia de primitivas que garantissem segurança à aplicação. Isto é, não era possível ter certeza de que a comunicação estava ocorrendo com o servidor esperado. Além disso, as mensagens transmitidas não eram criptografadas de ponto a ponto, possibilitando assim que qualquer entidade má intencionada pudesse interceptar o tráfego e inclusive manipulá-lo.

Na segunda versão, adicionamos suporte a SSL para prover uma comunicação segura entre os clientes, frontends e servidores. Assim, é possível garantir autenticação e evitar ataques de man-in-the-middle. Utilizamos a versão TLS 1.2 pois o sistema operacional Debian não aceitava mais as versões antigas.

Para implementar esse recurso, foi necessário realizar algumas mudanças estruturais no código da primeira versão. A função *localserver*, responsável por ouvir as chamadas de *PUSH* e *DELETE* do servidor no cliente no cliente, comportava-se como um servidor, ao qual o servidor do Dropbox se conectava. Agora, o servidor principal é quem inicia esse segundo servidor, ao qual a *localserver* se conecta. Essa alteração foi necessária pois senão seria necessário que cada cliente tivesse um certificado, o que não faria sentido.

2.3 Replicação Passiva

A replicação passiva foi implementada para um servidor primário e até duas réplicas secundárias. O cliente, quando deseja se conectar a um servidor do Dropbox, precisa primeiramente se conectar a um frontend (FE). O FE consiste em um proxy, que executa em um processo independente, responsável por fazer a conexão entre o cliente e o servidor primário da aplicação. Ao se conectar ao FE, o cliente transmite seu nome de usuário e aguarda a mensagem de sucesso indicando que a conexão ao servidor primário foi estabelecida. Após isso, o FE é responsável por passar todo o tráfego do cliente para o servidor primário e vice-versa.

Ao ser inicializado, um FE recebe por parâmetro os endereços IP e número de Porta TCP dos servidores disponíveis como replicas (RM - Replicas Managers). O primeiro servidor da lista representa o RM primário e os demais são os RM secundários (ou backups). Esse servidor primário, ao receber uma requisição de *upload* ou remoção de arquivo, primeiramente realiza essa operação na sua máquina e depois espelha as alterações para os outros servidores de forma sequencial. Após o RM primário receber a confirmação do sucesso da operação de todas as RM secundárias, ele envia uma confirmação para o FE, que a transmite ao cliente a conclusão da operação.

Quando um frontend detecta a falha do servidor primário, prontamente marca esse servidor como não disponível na lista de servidores, faz uma eleição do proximo servidor primário entre as replicas restantes e realiza a conexão com esse novo servidor primário. Quando o novo servidor torna-se primário recebe as informações atualizadas da lista de servidores e passa a informação da nova configuração adiante conectando com os demais RMs.

Outro caso possível de detecção de falha, é quando o servidor primário detecta uma falha em um servidores backups. Nesse caso o primário retira essa RM da configuração e atualiza as listas de servidores dos servidores backups e por final a dos frontends conectados.

2.4 Estruturas e funções adicionais

2.4.1 dropboxUtil.h

Para a segunda etapa do trabalho, fizemos algumas alterações explicadas a seguir:

```
1  #define MSGSIZE 512
2  #define MAXDEVICES 2
3  #define MAXSERVERS 3
4
5  // Both constants must have CONNECTION_MSG_SIZE characteres!
6  #define CONNECTION_FIRST "FC"
7  #define CONNECTION_NOT_FIRST "NF"
8  #define CONNECTION_MSG_SIZE 2
9
10 // Both constants must have CONNECTION_FIRST_MSG_SIZE characteres!
11 #define CONNECTION_FRONTEND "FE"
12 #define CONNECTION_SERVER "SE"
13 #define CONNECTION_FIRST_MSG_SIZE 2
```

constants_added.c

MSGSIZE - Tamanho fixo de todas as mensagens
MAXDEVICES - Número máximo de dispositivos conectados simultaneamente, por usuário

MAXSERVERS - Número máximo de servidores utilizados como Replica Managers (um primário e dois secundários)

CONNECTION_FIRST - Enviada pelo servidor para avisar ao FE que é a primeira vez que recebe alguma conexão. Nesse caso, o FE envia a lista de servidores para o servidor.

CONNECTION_NOT_FIRST - Enviada pelo servidor para avisar ao FE que já recebeu outras conexões

CONNECTION_MSG_SIZE - Tamanho fixo das duas mensagens anteriores

CONNECTION_FRONTEND - Enviada para o servidor caso a entidade que esteja se conectando a ele seja um FE, por exemplo, o FE se conectando ao servidor primário

CONNECTION_SERVER - Enviada para o servidor caso a entidade que esteja se conectando a ele seja outro servidor, por exemplo, o primário se conectando a um secundário

CONNECTION_FIRST_MSG_SIZE - Tamanho fixo das duas mensagens anteriores

```
1 typedef struct replication_server {
2   char ip[20];
3   int port;
4   int isAvailable;
5   SSL* socket;
6 } REPLICATION_SERVER_t;
```

replication_server.c

Foi criada a estrutura *replication_server* para armazenar as informações de cada um dos servidores. Essa estrutura contém IP e porta do servidor, a *flag isAvailable* que indica se o ele está disponível e o identificador do socket utilizado para se comunicar com ele.

2.4.2 dropboxFrontEnd.c

O FrontEnd foi totalmente criado nesta etapa do trabalho, baseado no cliente e no servidor. Nele adicionamos a lista *replication_servers* contendo os servidores disponíveis como RM. Essa lista foi implementada como um vetor de estruturas *REPLICATION_SERVER_t*.

Além disso, foram criadas diversas funções responsáveis por lidar com as conexões entre o cliente e o servidor primário. Isto é, cada uma delas executa em uma thread exclusiva e são responsáveis por ouvir as requisições normais do cliente e do servidor e passar de um para o outro.

2.4.3 dropboxClient.c

Para realizar a sincronização de relógios, foi adicionada a função *getLogicalTime()*, seguindo o comportamento indicado na especificação do trabalho. Ademais, foi criada a lista de arquivos *client_files*, que armazena as informações de todos os arquivos que o cliente possui, assim como já era feito no servidor.

3 Problemas encontrados

3.1 Criação do proxy para servir como Front End

Implementar o frontend como um proxy inicialmente pareceu uma boa ideia por executar em um processo independente. Para manter o FE simples, optamos por definir um tamanho fixo às mensagens, assim não seria preciso copiar grande parte dos códigos do cliente e do servidor para ele. No entanto, isso exigiu, por exemplo, reescrever a maioria das instruções de leitura e escrita no socket, e demandou um pouco mais de trabalho que o esperado. Além disso, tivemos dificuldades em fazer o FE realizar com sucesso a conexão entre o cliente e o servidor e repassar as mensagens entre eles de maneira que não acontecesse um deadlock.

3.2 Deadlocks na sincronização de relógios

Para implementarmos a sincronização de relógio da maneira comentada acima, seria necessário chamar a função *getLogicalTime()* dentro da função *send_file()* do cliente. Ao ser invocada, a *getLogicalTime()* realizava uma requisição de tempo para o servidor. Entretanto, o servidor estava

dentro da função *receive_file()* e, consequentemente, não estava preparado para receber uma requisição *TIME*. Desse jeito, ambos ficavam eternamente em espera: o servidor pelo timestamp de modificação do arquivo e o cliente pelo tempo do servidor.

A solução adotada para esse problema foi incluir parte do fluxo de sincronização de relógio dentro dessas funções. O servidor passou a enviar seu tempo dentro da *receive_file()*, e o cliente a recebê-lo na *send_file()*, sem enviar uma requisição *TIME* ao servidor.

3.3 Testes

Uma das dificuldades encontradas pelo grupo foi a realização de testes no sistema desenvolvido. Por termos implementado o frontend como um proxy, em um processo independente do cliente, a necessidade de criar três processos (para o caso mais simples de teste) a cada vez que queríamos realizar algum teste acabou se tornando uma grande sobrecarga. Para cenários mais complexos, por exemplo, envolvendo dois usuários diferentes e o acesso da conta de um usuário em dois dispositivos diferentes, trazia a necessidade de criarmos nove processos diferentes. Caso um erro fosse encontrado nesse cenário, a necessidade de realizar uma limpeza dos diretórios *sync_dir* do servidor e dos usuários e a criação de todos os processos de novo gerava uma sobrecarga maior ainda.

4 Conclusões

Ao terminar a implementação do trabalho, percebemos o quanto aprendemos com a construção do mesmo. Ao analisar a especificação vimos que teríamos que aprender diversas técnicas para a evolução da aplicação, como sincronização de relógios, um pouco de segurança (SSL e certificados) e replicação em geral, ainda evoluindo uma já complexa base de código da etapa 1 do trabalho. Isso fez com que atentássemos para o quão importante é construir um sistema modular e bem estruturado, tendo uma base estável que possa ser evoluída com mais facilidade e robustez no futuro.

A oportunidade de poder colocar em prática os conhecimentos adquiridos na disciplina (e no restante do curso) em um trabalho com conceitos e funções bem atuais foi fantástica. Trabalhar em uma mesma base de código por mais de um trabalho nos fez aprender bastante sobre o quão importante é gerar um software bem escrito e bem documentado. Implementar em um trabalho funções similares a serviços/produtos populares (como o Dropbox) nos faz relacionar e visualizar os conceitos aprendidos com o que realmente utilizamos no nosso dia-a-dia.