UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA

FRANCISCO KNEBEL
LUCIANO ZANCAN

syncBox:
EXCLUSÃO MÚTUA, REPLICAÇÃO E SEGURANÇA

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

Orientador: Prof. Dr. Alberto Egon Schaeffer Filho

Porto Alegre
2017

Introdução

O *syncBox* consiste na implementação de um serviço de sincronização de arquivos semelhante à plataforma Dropbox, implementado utilizando a linguagem C. Esse é o relatório sobre a segunda entrega do projeto, onde foram implementadas técnicas para exclusão mútua distribuída, autenticação para segurança e replicação.

Descrição do Ambiente de Teste

O ambiente de teste utilizado é o mesmo da primeira etapa entregue do trabalho. 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 núcleos, 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.

Agora que a aplicação suporta replicação, é necessário, além de informar o endereço e a porta em que o servidor rodará, o endereço e porta do servidor primário atual. E.g: *192.168.25.3 3000 192.168.25.4 3006*.

É necessário um arquivo chamado "*connection.config*" na pasta root do usuário a rodar o servidor, com os servidores (primário e réplicas) em ordem, com a sequência (ip, porta) separados por linha no arquivo. E.g: *192.168.25.3\n3000\n192.168.25.4\n3006*.

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. É necessário um arquivo chamado "*connection.config*" na pasta root do usuário a rodar o cliente, com os servidores (main e replicas) em ordem, com a sequência (ip, porta) separados por linha no arquivo. E.g: *192.168.25.3\n3000\n192.168.25.4\n3006*.

Alterações sobre a aplicação da primeira etapa

Antes da implementação das modificações requisitadas pela especificação, foi efetuada a correção sobre problemas apresentados na aplicação em sua versão anterior. Primeiramente, foi corrigido um problema com o timestamp utilizado para o tempo de modificação de um arquivo. Inicialmente, era utilizado o timestamp local de cada máquina, o que causava um problema de sincronização, pois os relógios de cada máquina são diferentes, podendo causar problemas. Assim, as operações de escrita sobre um arquivo foram refatoradas para receber o timestamp do arquivo no cliente, assim mantendo a sincronização pois utilizamos o relógio da mesma máquina. Um problema encontrado com isso foi que a aplicação encontrava problemas devido ao horário de verão, que modifica o timestamp em uma hora toda vez que salvo. Devido à isso, verificamos na estrutura do arquivo se o timestamp está ou não modificado para o horário de verão, e ajustamos caso estiver.

Em seguida, o comando *get_sync_dir* foi depreciado, pois a sua utilização agora é automática, ou seja, quando um arquivo é modificado em um dispositivo, a alteração é enviada para o outro dispositivo. Também foi inserido um mutex sobre a fila de clientes, mantida no servidor, para evitar problemas sobre modificações nessa estrutura, já que múltiplos clientes podem estar modificando o mesmo simultaneamente. Foram também encontrados e corrigidos bugs sobre a sincronização de arquivos que já foram apagados.

Entretanto, a modificação mais relevante foi sobre as funções de envio de mensagens. As funções *read* e *write* enviam buffers de tamanho fixo, mas por estarmos utilizando uma comunicação TCP, que utiliza *streams* de tamanhos variáveis, não há garantia que o bloco será totalmente enviado. A versão anterior assumia que sim, e utilizando a plataforma localmente não foram encontrados problemas. Entretanto, as *streams* enviadas de forma distribuída não estava recebendo todos os bytes esperados à tempo. Como esperavamos um bloco de tamanho fixo e recebíamos um bloco variável, de tamanho menor, a aplicação estava utilizando "*lixo*" de memória nas mensagens. As funções de envio e recebimento de mensagens foram refatoradas para garantir o recebimento dos blocos corretamente, garantindo que as mensagens são recebidas integralmente.

Após essas modificações sobre a aplicação anterior, pudemos implementar as funcionalidades previstas na especificação da segunda etapa do trabalho e detalhadas nas seções seguintes.

Justificativas gerais de desenvolvimento

Explique o funcionamento do algoritmo de exclusão mútua distribuído e suas limitações

Para a exclusão mútua, foi criado um algoritmo com arquivos “lock”. Ao tentar enviar um arquivo para o servidor primário, este primeiro cria um arquivo com o nome do arquivo, acrescido de “.lock”. Este arquivo só é excluído após concluída a transação. O arquivo lock não é enviado às réplicas. O que acontece é que ao concluir o envio de um arquivo (ou deleção do mesmo), o servidor primário chama o updateReplicas(), e só após sincronizar os servidores réplicas ele exclui o arquivo lock.

Como o cliente pode estar em devices diferentes, é utilizado um mutex para o arquivo lock no servidor primário, para proteger a seção crítica de testar se existe o arquivo lock, e se não existir criá-lo.

Como a replicação passiva foi implementada na sua aplicação e quais foram os desafios encontrados

Para a replicação passiva, cada servidor é também um cliente. Assim, cada servidor réplica consegue se comunicar com o servidor primário. Porém, este executa funções específicas de acordo com o cliente conectado, ou seja, um cliente comum ou uma réplica.

Do lado do cliente, tanto do servidor réplica quanto do cliente em si, foi implementado uma função para detectar quando uma desconexão com o servidor for detectado. Quando isso acontecer, o cliente irá automaticamente efetuar uma reconexão com uma réplica, para posteriormente enviar o comando procurado para o servidor, funcionando de forma transparente.

O servidor réplica, fica constantemente ouvindo o servidor mestre. No cliente, sempre antes de executar qualquer função como upload, download, delete e list, ele envia uma mensagem e o servidor deve responder caso estiver online. Caso este caia, a função read retorna o valor ‘0’. Na função de read, existe uma variável chamada “tentativa”. Cada vez que um cliente tentar ler de uma conexão perdida, o SSL_read() retorna valor 0. Com 5 valores 0’s sequentes, ele tem aceita que o servidor caiu e tenta se reconectar com o próximo do arquivo “connection.config”. Só após reestabelecer conexão é que a função que estava sendo executada volta a ser executada.

No caso da réplica, ainda, antes de tentar reconectar ele testa se o ip e a porta do arquivo “connection.config” não são dele próprio. Caso afirmativo, chama a função pthread_exit() e encerra a thread cliente, passando a ser apenas servidor main (podendo receber outras réplicas também).

Explique os aspectos de segurança fornecidos pela nova versão da aplicação, em relação a desenvolvida na Parte I

Para essa nova versão da aplicação, foi introduzido um sistema para autenticação utilizando mecanismos de SSL. O protocolo SSL provê privacidade e integridade de dados entre duas aplicações que se comunicam pela internet. Com a autenticação de ambas as partes, utilizando a certificação, podemos efetuar a cifragem dos dados transmitidos entre as duas partes, evitando que intermediários interfiram na comunicação, obtendo acesso indevido ou modificando os dados que estão sendo transmitidos.

Para a sua implementação, foi apenas necessário a inicialização dos mecanismos do protocolo, como definido na especificação, e a refatoração sobre as funções de recebimento e envio de mensagens, para utilizar o socket SSL criado. Foi utilizado um certificado autoassinado para testes.

Estruturas e funções adicionais que você implementou

Estruturas adicionais

- Não foram necessárias estruturas extras sobre a implementação anterior. Entretanto, foi necessário a modificação de alguns tipos de atributos para modificar os sockets comuns anteriores para SSL. Também foi inserido alguns campos nas estruturas existentes, como char* host e int port para o cliente, para guardar as informações de conexão referentes aos clientes e servidores. Estas modificações não foram utilizadas, mas poderiam ser utilizadas caso fosse implementado um mecanismo de re-conexão automático, sem o arquivo “connection.config”

Funções adicionais do Cliente

- void reconnect_server() - Irá buscar no arquivo “connection.config” por um endereço para efetuar uma nova conexão, em um dos servidores réplica. Quando encontra, efetua a conexão com o servidor primário novo, possibilitando a continuidade do serviço.

Funções adicionais do Servidor

- void synchronize_replica_send(SSL *ssl_sync, ClientList* ClientList, char* serverFolder) - Chamada na primeira conexão de um servidor réplica. Envia as pastas e arquivos do servidor main para a réplica;
- void synchronize_replica_receive(SSL *ssl_sync, char* serverFolder) - Chamada na primeira conexão pelo servidor réplica. Recebe as pastas e arquivos do servidor main para si;
- int updateReplicas(char* file_path, char* command) - envia comandos de criação de pasta, deleção ou atualização de arquivos para todos os servidores réplicas;

- void* connect_server_replica (void* connection_struct) - conecta uma estrutura de conexão de um servidor réplica em um servidor principal, e fica ouvindo este servidor. A cada comando recebido, executa.
- int reconnect_server_replica() - reconecta o servidor replica no novo servidor principal, buscando o endereço dele no arquivo “connection.config”. Caso o novo servidor seja ele próprio, chama a função de encerramento de thread (pthread_exit).

Funções adicionais Gerais

- int read_to_file(FILE* pFile, int file_size, SSL* ssl) - lê um tamanho file_size de um ssl para um arquivo pFile;
- int write_to_socket(SSL *ssl, char* buffer) - escreve em um socket ssl um buffer;
- int read_from_socket(SSL *ssl, char* buffer) - le um buffer de um socket ssl. Caso não consiga ler, indicando que o socket está fechado, chama a função de reconexão.

Desafios de desenvolvimento

Para implementação da exclusão mútua e utilização de SSL, não houve problemas reportados. SSL foi necessário apenas refatorações simples e seguir as informações documentadas na especificação. Sobre a exclusão mútua, é uma implementação simples para garantir um suporte mínimo sobre a funcionalidade esperada.

O grande problema sobre essa parte do projeto foi a implementação da replicação passiva. Foi gasto uma quantidade considerável de tempo para a discussão de como poderíamos implementar, sendo que foi considerado trocar a forma de replicação para ativa, pois pensávamos dessa forma para resolver o problema de disponibilidade. Entretanto, decidimos por seguir com a forma passiva, implementando o mecanismo já descrito acima, em que cada servidor réplica é também cliente.

Conclusão

Encerramos o desenvolvimento da segunda etapa do **syncBox** com muita satisfação. Encontramos muitos problemas gerados pela mudança do escopo de uma aplicação simples para uma que execute de forma distribuída e paralela. Foi um projeto que demandou muito planejamento, e devido à isso, a implementação da segunda etapa requereu menos tempo do que a primeira parte, já que se tratou de refatorações para criar funcionalidades extras sobre a base criada anteriormente.

Ficamos satisfeitos também com a correção de todos os bugs encontrados, que foram reportados no relatório anterior, sobre a primeira versão da aplicação, criando uma nova versão mais estável e confiável.

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 do trabalho. Foi criado

uma aplicação exemplo funcional, que demonstra várias soluções para problemas de computação paralela e distribuída, e agradecemos pela oportunidade e instrução oferecida por parte do professor orientador.

Como informado na primeira etapa, esse projeto será distribuído de forma *open-source* e livre, no repositório *https://github.com/FranciscoKnebel/syncBox/*, que foi utilizado para desenvolvimento da aplicação, para divulgação pública do que foi desenvolvido e para que outros possam partilhar de nossas soluções.