INSTITUTO DE INFORMÁTICA
UFRGS

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

INF01151 – SISTEMAS OPERACIONAIS II N
SEMESTRE 2017/1
TRABALHO PRÁTICO PARTE 2: RELÓGIOS FÍSICOS, REPLICAÇÃO E SEGURANÇA

ESPECIFICAÇÃO DO TRABALHO

Este projeto consiste na implementação de um serviço semelhante ao Dropbox e está dividido em duas partes. Na primeira etapa, foi necessário implementar o funcionamento básico do serviço, com enfoque nos aspectos de programação com múltiplos processos/threads, comunicação e controle de concorrência. Nessa etapa, você deverá estender o serviço com algumas funcionalidades avançadas, onde destacam-se: sincronização de relógios físicos, replicação e segurança.

O programa deverá executar obrigatoriamente em ambientes Unix (Linux) mesmo que o trabalho tenha sido desenvolvido em outra plataforma. Quando apropriado, a especificação abaixo define funções e trechos de código que você deverá implementar. No entanto, é possível estender estas funções caso for julgado necessário, destacando as modificações no relatório final a ser entregue.

FUNCIONALIDADES AVANÇADAS

Na segunda parte deste projeto, a aplicação deverá ser estendida para atender os conceitos estudados na segunda metade da disciplina. Isto inclui: sincronização de relógios físicos, replicação e segurança.

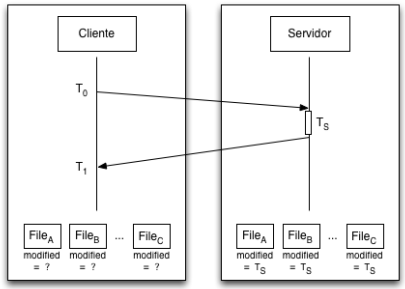
1. Sincronização de Relógios Físicos

Devido à falta de um relógio global absoluto, a noção temporal do servidor e do cliente pode ser diferente. Dessa forma, os timestamps associados à hora de última modificação dos arquivos no cliente e servidor (campo *last_modified [MAXNAME]* da estrutura *file_info*) poderão estar inconsistentes. Obs.: dependendo da forma como você implementou o serviço na Parte I, essa inconsistência pode ter um impacto maior ou menor -- i.e., impacto baixo se o campo *last_modified* for usado apenas na visualização de metadados do arquivo, ou impacto alto se *last_modified* for usado para determinar se uma atualização deve ser propagada ou não. Em qualquer um dos casos acima, iremos assumir que o relógio da aplicação cliente não é confiável, e para corrigir essa distorção você deverá implementar uma versão simplificada do Algoritmo de Cristian visto em aula. Este algoritmo adaptado será usado para atualizar a data de última modificação de um arquivo no cliente, tomando como referência a hora do servidor.

Quando necessário (antes de atualizar o campo *last_modified*), o cliente deverá enviar uma requisição, no tempo T_0, ao servidor perguntando qual é a sua hora. Ao processar a informação, o servidor deverá retornar ao cliente uma resposta, anexando sua hora T_s. Ao receber a resposta, no tempo T_1, o cliente deverá reajustar a hora de modificação do arquivo para que fique consistente com a referência temporal do servidor. Conforme o algoritmo, a hora no cliente é calculada da seguinte forma:

$$T_c = T_s + \frac{T_1 - T_0}{2}$$

Como simplificação, você não precisa necessariamente alterar as configurações de data/hora da máquina cliente; basta apenas alterar os campos de metadados mantidos pela aplicação cliente para que esses estejam consistentes com a visão do servidor. Para tal, é necessário que o cliente disponha de uma estrutura de dados similar à *struct file_info* (mantida no servidor). A interação esperada entre cliente e servidor é ilustrada abaixo:



Como referência, o cliente deverá solicitar a execução da função abaixo para obter uma referência temporal consistente com a visão do servidor:

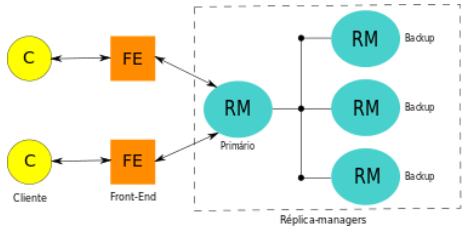
Interface	Descrição
`int getTimeServer(void);`	Requisita o horário do servidor na forma de um timestamp Unix. O resultado será usado pela aplicação cliente para calcular sua nova referência temporal (a ser utilizada nos registros de arquivos).

2. Replicação Passiva

O servidor implementado na Parte I, caso por ventura falhar, acarretará em indisponibilidade de serviço para seus clientes. Este é um estado que não é desejável no sistema. Para aumentar a disponibilidade do sistema implementado, você deverá aplicar o conceito de *replicação passiva, onde o servidor Dropbox será representado por uma instância de replica manager (RM) primária, e uma ou mais instâncias de replica managers secundárias (ou backup)*.

Podemos entender este modelo adicionando um *front-end* (FE) entre a comunicação do cliente (C) e servidor, agora representado por um conjunto de *replica managers* (RMs). Este *front-end* será responsável por realizar a comunicação entre estas entidades, tornando transparente para o cliente qual é a cópia primária do servidor. Você precisará garantir que:

(1) todos os clientes sempre utilizarão a mesma cópia primária;
(2) após cada operação, o RM primário irá propagar o estado dos arquivos aos RMs de backup;
(3) somente após os backups serem atualizados o primário confirmará a operação ao cliente.



Como referência, o RM primário deverá solicitar a execução da função abaixo para propagar as alterações aos backups:

Interface	Descrição
`int updateReplicas(void);`	Executado pela réplica primária para realizar o espelhamento da operação executada em todas as réplicas de backup. A operação só é confirmada caso todos os backups recebam a atualização.

3. Autenticação do Cliente/Servidor e Comunicação Segura

Autenticação permite que um participante (cliente) possa verificar e validar a identidade de outro participante (servidor) em um canal de comunicação. Nessa etapa do trabalho, será necessário estender as aplicações cliente e servidor para que suportem a funcionalidade de autenticação. Você deverá utilizar a biblioteca OpenSSL. Por padrão, um servidor SSL sempre envia seu certificado, se disponível, permitindo verificar a autenticidade apenas do servidor. É necessário ter o pacote *libssl-dev* instalado. Para tal, execute:

%> sudo apt-get install libssl-dev

A solução desenvolvida deverá utilizar criptografia assimétrica (criptografia de chaves pública/privada). Para isso, você deverá gerar a chave privada e o certificado digital utilizando OpenSSL. Utilize o seguinte comando:

%> openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout KeyFile.pem -out CertFile.pem

Nesta demonstração, *KeyFile* se refere à chave privada e *CertFile* corresponde ao certificado digital gerado. Este certificado será auto-assinado (obs.: a geração de certificados auto-assinados não é uma boa prática fora do contexto deste trabalho).

- **Adicionando conexão segura ao Cliente**

Estenda o cliente de forma a comportar SSL. Para tal, será necessário a adição de, pelo menos, as seguintes funções abaixo. Você também precisará dos includes *openssl/ssl.h* e *openssl/err.h*.

Passos:

1. **Inicialize a engine SSL:** No código do cliente, crie o contexto, a estrutura principal SSL (a qual mantém todas as outras estruturas necessárias), inicialize o contexto fazendo uso da função definida no resumo das funções e crie a conexão.

```
SSL_METHOD *method;
SSL_CTX *ctx;
```

SSL_METHOD *method - inicializa um ponteiro para armazenar a estrutura que descreve as funções internas da biblioteca SSL que implementa o protocolo SSLv2. É necessário para criar o SSL_CTX.

SSL_CTX *ctx - ponteiro que armazenará o contexto.

```
OpenSSL_add_all_algoritms();
SSL_load_error_strings();
method = SSLv2_client_method();
ctx = SSL_CTX_new(method);
if (ctx == NULL){
   ERR_print_errors_fp(stderr);
   abort();
}
```

OpenSSL_add_all_algoritms(); - Adiciona todos os algoritmos (digests e ciphers).

2. **Anexe SSL ao socket:** Crie um novo estado de conexão SSL, anexe este ao descritor de socket e realize a conexão.

```
ssl = SSL_new(ctx);
SSL_set_fd(ssl, sockfd);

if (SSL_connect(ssl) == -1)
      ERR_print_errors_fp(stderr);
   else{
      */ conexão aceita */
   }
}
```

Anexando SSL ao descritor de socket já criado na Parte I.

3. **Leia e escreva de forma segura:** Ao ter a conexão aceita, é possível fazer uso das funções para envio e recebimento, SSL_write e SSL_read, respectivamente.

`int SSL_write(SSL *ssl, const void *buf, int num)`	Escreve *num* bytes do buffer *buf* na conexão *ssl*.
`int SSL_read(SSL *ssl, void *buf, int num)`	Lê *num* bytes do buffer *buf* na conexão *ssl*.

4. **Mostre o certificado**: você poderá mostrar no terminal os dados do certificado digital.

```
X509 *cert;
char *line;

cert = SSL_get_peer_certificate(ssl);
if (cert != NULL){
   line = X509_NAME_oneline(
      X509_get_subject_name(cert),0,0);
   printf("Subject: %s\n", line);
   free(line);
   line = X509_NAME_oneline(
      X509_get_issuer_name(cert),0,0);
   printf("Issuer: %s\n", line);
}
```

Mostra o certificado do peer com o seu nome e quem assinou seu certificado.

*X509 *cert; - cria um ponteiro para receber um certificado do padrão X.509;*

cert = SSL_get_peer_certificate(ssl); - Obtém o certificado do servidor.

- **Adicionado conexão segura ao Servidor**

Estenda o servidor de modo a comportar SSL. Para tal, será necessário a adição de, ao menos, as seguintes funções abaixo. Você precisará dos includes *openssl/ssl.h* e *openssl/err.h*.

Passos:

1. **Inicialize a engine SSL:** No código do servidor, crie o contexto, a estrutura principal SSL (a qual mantém todas as outras estruturas necessárias), inicialize o contexto fazendo uso da função definida no resumo das funções e crie a conexão.

```
SSL_METHOD *method;
SSL_CTX *ctx;
```

SSL_METHOD *method - inicializa um ponteiro para armazenar a estrutura que descreve as funções internas da biblioteca ssl que implementa o protocolo SSLv2. É

`OpenSSL_add_all_algoritms();` `SSL_load_error_strings();` `method = SSLv2_server_method();` `ctx = SSL_CTX_new(method);` `if (ctx == NULL){` `  ERR_print_errors_fp(stderr);` `  abort();` `}`	necessário para criar o SSL_CTX. SSL_CTX *ctx - ponteiro que armazenará o contexto. OpenSSL_add_all_algoritms(); - Adiciona todos os algoritmos (digests e ciphers).

2. **Carregue o certificado:** Crie o contexto, inicialize o contexto fazendo uso da função definida no resumo das funções, carregue o certificado e abra o listener.

`int SSL_CTX_use_certificate_file(SSL_CTX *ctx, const char *file, int type);`	Carrega o certificado armazenado em *file* para *ctx*.
`int SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx, const char *file, int type);`	Adiciona a chave privada armazenada em *file* para *ctx*.

3. **Anexe SSL ao socket:** No laço, aceite a conexão do cliente, crie um novo contexto para a conexão SSL e anexe-a ao descritor de socket.

`int client = accept (server, &addr, &len);` `ssl = SSL_new(ctx);` `SSL_set_fd(ssl, client);`	Anexando SSL ao descritor de socket já criado na Parte I.

4. **Aceite a conexão, leia e escreva de forma segura:** Ao ter a conexão aceita, é possível fazer uso das funções para envio e recebimento, SSL_write e SSL_read, respectivamente.

`int SSL_write(SSL *ssl, const void *buf, int num)`	Escreve *num* bytes do buffer *buf* na conexão *ssl*.
`int SSL_read(SSL *ssl, void *buf, int num)`	Lê *num* bytes do buffer *buf* na conexão *ssl*.

5. **Mostre o certificado**: você poderá mostrar no terminal os dados.

`X509 *cert;` `char *line;` `cert = SSL_get_peer_certificate(ssl);` `if (cert != NULL){` `  line = X509_NAME_oneline(` `  X509_get_subject_name(cert),0,0);` `  printf("Subject: %s\n", line);` `  free(line);` `  line = X509_NAME_oneline(` `  X509_get_issuer_name(cert),0,0);` `  printf("Issuer: %s\n", line);` `}`	Mostra o certificado do peer com o seu nome e quem assinou seu certificado. *X509 *cert; - cria um ponteiro para receber um certificado do padrão X.509;* *cert = SSL_get_peer_certificate(ssl); - Obtém o certificado do servidor.*

- **Compilacão**

Para compilar tanto o cliente quanto o servidor, utilize os seguintes parâmetros via linha de comando:

%> gcc -o server server.c -lssl -lcrypto
%> gcc -o client client.c -lssl -lcrypto

DESCRIÇÃO DO RELATÓRIO A SER ENTREGUE

Deverá ser produzido um relatório fornecendo os seguintes dados:

- Descrição do ambiente de teste: versão do sistema operacional e distribuição, configuração da máquina (processador(es) e memória) e compiladores utilizados (versões).
- Apresente claramente no relatório uma descrição dos pontos abaixo:
  - (A) Explique o problema de sincronização de relógios endereçado pelo algoritmo de Cristian e suas limitações;
  - (B) Como a replicação passiva foi implementada na sua aplicação e quais foram os desafios encontrados;
  - (C) Explique os aspectos de segurança fornecidos pela nova versão da aplicação, em relação àquela desenvolvida na Parte I;
  - (D) Estruturas e funções adicionais que você implementou;
- Também inclua no relatório problemas que você encontrou durante a implementação e como estes foram resolvidos (ou não).

A **nota será atribuída baseando-se nos seguintes critérios**: (1) qualidade do relatório produzido conforme os itens acima, (2) correta implementação das funcionalidades requisitadas e (3) qualidade do programa em si (incluindo uma interface limpa e amigável, documentação do código, funcionalidades adicionais implementadas, etc).

DATAS E MÉTODO DE AVALIAÇÃO

O trabalho deve ser feito em grupos de **3 OU 4 INTEGRANTES**, conforme a configuração de grupos da Parte I. Não esquecer de identificar claramente os componentes do grupo no relatório.

Faz parte do pacote de entrega, os arquivos fonte e o relatório em um arquivo ZIP. O trabalho deverá ser entregue de acordo com o seguinte cronograma:

- Turma A: até 08:30 horas do dia 18 de julho.
- Turma B: até 08:30 horas do dia 17 de julho.

A entrega deverá ser via moodle. As demonstrações ocorrerão no mesmo dia, no horário da aula.

Após a data de entrega, o trabalho deverá ser entregue via e-mail para alberto@inf.ufrgs.br (subject do e-mail deve ser "INF01151: Trabalho Parte 2"). Neste caso, será descontado 02 (dois) pontos. O atraso máximo permitido é de uma semana após a data prevista para entrega, isto é, nenhum trabalho será aceito após o dia 25 de julho para a Turma A ou 24 de julho para a Turma B.