Relatório Parcial de Redes de Computadores

Grupo:
Francisco Machado
Jorge Batista
Sidnei Santiago
Tiago Gordiano
Vinícius Pinto

6 de Novembro, 2019

Universidade Federal da Bahia

Introdução

Em conformidade com as especificações do trabalho da disciplina de Redes de Computadores, ministrada pelo professor Gustavo B. Figueiredo, este documento visa relatar o desenvolvimento da camada de aplicação no que condiz à um programa de transferência de arquivos, correspondente aos requerimentos da segunda etapa do trabalho da disciplina.

Repositório: https://github.com/furran/mata59

Plataforma e bibliotecas

O código citado neste relatório foi escrito inteiramente na linguagem C, como exigido na especificação, e na plataforma *Windows*, utilizando o *GCC* como compilador padrão para o projeto. Vale notar que também utilizadas as bibliotecas *Winsock*[1] e *Windows* para a implementação do protocolo de transferência de arquivo e a interface gráfica do usuário, respectivamente.

O protocolo de transferência de arquivos

Iremos começar pela parte mais importante ou, devo talvez dizer, o foco desta etapa do trabalho. O nosso protocolo de transferência de arquivos, presente no arquivo nomeado criativamente de "ftp.c", foi construído sobre as pilhas de protocolo TCP/IP já implementadas na biblioteca *Winsock*. Não entraremos em detalhes quanto ao funcionamento de nenhum destes protocolos, uma vez que estes estão fora do escopo da implementação nesta etapa.

O nosso protocolo de transferência possui as seguintes funções:

```
int send_file(int sock, char* filename);
int recv_file(int sock);
int send_data(int sock, void* data, int length);
int recv_data(int sock, void* data, int maxbuflen);
int send_all(int sock, void* buffer, int length);
char* get_filename_extension(const char* filename);
```

Podemos ver que o protocolo é composto por duas funções principais — *send_file()* e *recv_file()* — que se utilizam de funções auxiliares (*send_data(), recv_data,* etc.) e estas, por sua vez, se utilizam das funções da camada de transporte. Vamos falar mais sobre elas a seguir.

[1] https://en.wikipedia.org/wiki/Winsock

As funções

send_file() e recv_file()

Começamos o envio do arquivo chamando a função *send_file()*, passando como parâmetro o descritor da *socket* de destino e o nome do arquivo local. Se o arquivo existe, então envia-se o nome do arquivo e começa-se uma rotina de leitura em binário e envio do arquivo em blocos de 512[2] bytes, até que a flag de fim de arquivo *feof()* seja acionada. Uma mensagem com um "header" de valor 0 é então transmitida, indicando o fim bem sucedido da transmissão.

Para que o envio do arquivo seja bem sucedido — ou mais ou menos sucedido — é preciso que a máquina alvo esteja esperando com *recv_file()*. Ao finalmente receber o nome do arquivo remoto, a extensão associada ao nome é então extraída (.txt, .zip, .png, ...) para gerar o arquivo local com o nome "download[.*extensão*]" ou "download(*i*)[.*extensão*]", dependendo se já existe ou não um arquivo com mesmo nome no local. Depois disso segue então as rotinas de recebimento do blocos de bytes e escrita no arquivo até que se receba uma mensagem com o "header" igual a zero.

Caso ocorra qualquer erro durante a transferência do arquivo, seja no envio ou recebimento dos dados, na leitura ou escrita do arquivo, a transferência do arquivo falha.

send_data(), recv_data() e miscelâneas

Agora você talvez esteja se perguntando: "*Mas como estes blocos de dados são enviados entre os hosts?*". E eu respondo: é aí que as funções *send_data()* e *recv_data()* entram.

Para que um bloco de dados seja enviado, utilizamos *send_data()*, adicionando 3 parâmetros respectivamente: o descritor da *socket*, os dados à serem enviados e o "*tamanho*" do que está sendo enviado. Eu digo "*tamanho*" pois não é uma descrição justa de como este parâmetro está sendo utilizado na função. Na verdade o que acontece é que *send_data()* envia dois blocos de dados consecutivos. Primeiro, ele envia o *tamanho* na forma de uma c-string de 10 bytes, que funciona como um header para o bloco que vem logo em seguida — o bloco de dados. Poderemos alterar o papel deste "header" no futuro e reapropriá-lo para troca de sinais propriamente entre o remetente e o destinatário. No momento ele é especialmente útil para sinalizar o fim da transmissão.

A função *recv_data()* recebe as duas mensagens, extraindo o header e passando os dados para o buffer especificado nos parâmetros. Ambas as funções retornam apenas o número de bytes de **dados** enviados/recebidos, sem consideração pelo header.

A função *send_all()* insiste em enviar o conteúdo do buffer, até que tudo seja enviado, ou até que função falhe. Ela é usada internamente por *send_data()*. Já *get_filename_extension()* serve para pegarmos a extensão do arquivo a partir do nome. Esta é utilizada no recebimento do arquivo.

[2] Não... exatamente. Está sujeito a mudanças.

Execução e a interface gráfica do usuário

Entre os nossos códigos-fonte, além de “ftp.c”, encontraremos “cliente_gui.c”, “cliente.c” e “servidor.c”. Não entraremos em detalhes nesse códigos, mas posso dizer que neles se encontram a *utilização* do protocolo descrito em “ftp.c”. Eles são responsáveis pela abertura das *sockets*, o estabelecimento da conexão via IPv4 e a chamada da função de transferência de arquivo.

O programa “servidor.c”, após inicializar e estabelecer conexão com o outro hospedeiro, espera pacientemente pelo recebimento de um único arquivo. Após receber o arquivo, ou falhar em receber o arquivo, o programa para. Qualquer parte já recebida do arquivo permanece no disco.

Os programas “cliente.c” e “cliente_gui.c” são, respectivamente, a implementação em terminal e a implementação gráfica do que seria o agente responsável por enviar o arquivo ao “servidor.c”. A maior diferença entre os dois na prática é que enquanto “cliente.c” finaliza após enviar um arquivo, “cliente_gui.c” pode enviar quantas vezes quiser antes de ser finalizado. Isto é, supondo que “servidor.c” esteja executando para recebê-lo.

Ao compilar qualquer um dos códigos, adicionar “-lws2_32”, “-lwsock32”, ou o quer que seja para linkar a biblioteca do *Winsock*. No caso da interface gráfica, é preciso também adicionar “-lcomdlg”.

Erros, problemas e dificuldades

Quanto ao design do protocolo, acredito que a forma como incorporamos, por exemplo, o header poderia ser otimizada, ou ao menos feita de maneira mais limpa. O fato de que temos de enviar um bloco de header separadamente para cada bloco de dados provavelmente pesa no desempenho e resulta em um tempo maior de transferência do arquivo, ao contrário de se utilizássemos um único bloco para mandar os dados e o header de uma única vez.

Entre outros problemas encontrados, erros conhecidos e falhas de projeto temos:

- A Escolha de sistema operacional (*Windows*) — *possivelmente*[3];
- A interface gráfica para de responder durante o envio do arquivo — "*Quem dera tivéssemos fork()*[4]*"*;
- Organização;
- Procrastinação[5];
- *DevOps* em geral;

Levando tudo em consideração, acho que podemos considerar essa uma oportunidade para aprender com nossos erros e seguirmos em frente nesta jornada pelo conhecimento.

Nas palavras de Bob Ross[6]: *Nós não cometemos erros, apenas acidentes felizes*.

[3] A maioria referências e artigos de programação de sockets parecem ser orientados à sistemas Unix, Linux, BSD, etc....
[4] *CreateProcess()* literalmente leva 1 trilhão de parâmetros - https://docs.microsoft.com/en-us/windows/win32/procthread/creating-processes
[5] Ver data do relatório
[6] https://en.wikipedia.org/wiki/Bob_Ross