Introdução às Redes de Comunicação
Relatório do Trabalho Prático
Sistema Cliente-Servidor para Transferência de Ficheiros

Pedro Rodrigues 2018283166
Miguel Rabuge 2018293728

1. Introdução

As Redes de comunicação dotam os seus utilizadores com a capacidade de comunicar, por todo o mundo, numa questão de segundos. Estas revolucionaram a forma como vivemos, quer pela facilidade de acesso à informação digital quer pela partilha da mesma.

Na implementação deste trabalho, um sistema cliente-servidor para transferência de ficheiros entre dois sistemas, tivemos a oportunidade de explorar dois protocolos de comunicação, TCP e UDP, assim como a encriptação da informação a ser transmitida, com recurso à biblioteca *Libsodium*.

Nos seguintes capítulos, iremos falar de como foi feita a implementação de cada um destes.

2. Cliente

O cliente está organizado em 2 ficheiros: *cliente.h* e *cliente.c*. No *header file* estão presentes as seguintes estruturas de dados**:**

client_settings_t contendo o **porto** e o **protocolo** que são utilizados quer na ligação do cliente ao proxy, quer na deste ao servidor, os **endereços IP** do proxy e do servidor e os **endereços de rede** do proxy e do servidor (obtidos com recurso à função inet_pton que converte o endereço IP passado por parâmetro (String) num valor para a estrutura ***in_addr*** que será posteriormente atribuída à variável **struct in_addr sin_addr** presente na **struct sockaddr_in** que mencionaremos abaixo).

stats_t contendo informações relativas a um download, tais como o **nome** e **número de bytes** do ficheiro transferido, bem como o **protocolo** utilizado e o **tempo** que demorou, quer em segundos, quer em microsegundos.

No ficheiro cliente.c, o programa adota o seguinte procedimento: Verifica se o número de argumentos passado está correto. Se sim, carrega essa informação para uma variável do tipo ***client_settings_t*** com recurso à função get_settings (passando por parâmetro todos os argumentos de entrada da função main), que irá dar parse da informação concedida. Feito isto, são criados dois sockets, um para TCP e outro para UDP seguidos do preenchimento de duas estruturas ***sockaddr_in***, tanto para o proxy como para o servidor, que acomodam os endereços destes com a finalidade do cliente, proxy e servidor se "conhecerem" e poderem comunicar. Estas duas estruturas são passadas por parâmetro à função "client" onde se sucede o seguinte: O cliente, através da função "connect", tenta ligar-se ao proxy pelo socket previamente criado e transmite-lhe, em caso de sucesso na ligação, o protocolo a ser utilizado unicamente para a ligação (TCP), ficando à espera da resposta do mesmo. O proxy comunicará com o servidor, recebendo uma resposta deste que irá retransmitir ao cliente. Se este lhe responder com "REJECTED" significa que o número máximo de clientes for excedido, e portanto o cliente interpretará isso como uma instrução para terminar a sua execução, ou, caso contrário, procede à sua normal execução, esperando por instruções (**LIST**, **DOWNLOAD** e **QUIT**) pela linha de comandos. A responsabilidade da obtenção da lista de ficheiros presentes no servidor, do ponto de vista do cliente, foi atribuída à função "get_list". Analogamente, a função "get_file" está responsável pelo download do ficheiro a ser transmitido, assim como por todas as estatísticas referentes ao seu encargo.

3. Proxy

Analogamente, para o proxy temos também 2 ficheiros: ircproxy.h e ircproxy.c. No *header file* estão presentes as estruturas:

proxy_settings_t contendo o **porto**, o mesmo que é utilizado na ligação do cliente ao proxy e deste ao servidor, o **endereço IP** previamente definido por um «*define PROXY_IP_ADDRESS "127.0.0.4"»*, e o **endereço de rede** (obtido com recurso à função inet_pton, como referido anteriormente).

pclient_thread_t contendo informações referentes ao index no array de threads "clients" ao encargo da variável **thread_index**, e os *file descriptors* do socket associado a essa thread (cliente) à responsabilidade da variável **client_fd.**

No ficheiro *ircproxy.c*, o programa tem o seguinte comportamento: É verificado se o número de argumentos passado está correto. Se sim, é feito o parsing do comando e e a alocação dinâmica para as threads cliente e *file descriptors* dos sockets. Seguidamente são criados dois sockets, um para TCP outro para UDP. O socket para o TCP é posteriormente associado a um endereço através da função "bind" e posto à espera de ligações nesse mesmo socket, através da função "listen", ligações estas que serão feitas por clientes com o intuito de se conectarem ao servidor. Para dar resposta à gestão da ligação dos clientes e à receção de comandos na linha de comandos do proxy, decidimos criar uma thread - proxy_thread - explicada mais abaixo, que satisfaz a primeira necessidade, enquanto a função main responde aos comandos enviados para o proxy - **LOSSES**, **SHOW** e **SAVE.**

- A instrução **LOSSES** controla a percentagem de informação que não é transmitida do servidor ao cliente pelo protocolo UDP. Este controlo é implementado através de uma variável inteira "losses" aplicada a uma estrutura condicional tal que se ((rand() % 100) > losses), então envia o pacote, caso contrário, este fica perdido. Só se aplica, obviamente, ao protocolo UDP.
- A instrução **SHOW** mostra na consola todos os clientes ativos através de meros printf()'s.
- A instrução **SAVE** permite gravar todos os ficheiros transferidos do servidor para os clientes, e esta funcionalidade é implementada através de uma variável global inteira "save" interpretada como um booleano que fica a 1 quando o utilizador introduz o comando "SAVE ON" e a 0 com "SAVE OFF". Esta variável entra em estruturas condicionais na função "transmit_file" que caso esteja "ON", o ficheiro é guardado na pasta "Saved_files" contida na diretoria do Proxy, onde estão também o proxy.h e proxy.c.

A thread proxy_thread, associada à função proxy, tem a função de aceitar os clientes que se querem conectar, atendendo ao número máximo permitido, guardar os seus *file descriptors* e indexes no array de threads na estrutura client_thread_t, que será passada por parâmetro às novas threads, criadas também pela proxy_thread, associadas à função "new_client", que permitem representar individualmente cada cliente e suportar multiplos pedidos ao servidor criando um novo socket por cada um, tentando estabelecer uma ligação ao servidor, através da função "connect" pelo endereço que estes lhe indicam. É também responsabilidade da função "new_client" o parsing dos pedidos dos clientes já referidos como **LIST**, **DOWNLOAD** e **QUIT**, que são intermediados pelo proxy. Responsáveis por este processamento, estão as funções "transmit_dir" e "transmit_file", respetivamente para o **LIST** e para o **DOWNLOAD**.

4. Servidor

Também para o servidor temos dois ficheiros: server.h e server.c. No *header file* está presente a estrutura ***server_settings_t*** que contém o **porto**, já anteriormente referido, **o número máximo de**

clientes, o **endereço IP**, previamente definido por *«#define SERVER_IP_ADDRESS "127.0.0.2"»* e o **endereço de rede** (obtido com recurso à função inet_pton, já referido).

De igual modo, no servidor é verificado se o número de argumentos passados por parâmetro está correto. Se sim, é feito, analogamente, o parsing do comando,através da função "get_settings" e a alocação dinâmica para as threads cliente e para os *file descriptors* correspondentes aos sockets de cada cliente. Posteriormente são criados dois sockets, um para TCP e o outro para UDP, que são associados a um endereço agora definido através da função "bind" e o socket do TCP fica à espera de novas ligações provenientes do proxy, através da função "listen". A partir deste momento, qualquer ligação solicitada ao servidor é aceite, tendo em conta o número máximo de clientes, guardado o seu index do array de threads e o seu *file descriptor* na estrutura ***client_thread_t*** que será passada por parâmetro aquando da criação de numa nova thread cliente associada à função "new_client". Caso contrário o servidor escreve para o socket do cliente em questão "REFUSED", que é intermediado pelo proxy e posteriormente recebido pelo cliente, que desiste, controladamente, do requisito, como referido no ponto 2.

Cada thread destas fica ativa até receber a instrução **QUIT**, oriunda do cliente que elimina as suas threads e fecha os sockets que estavam associados a este, quer no servidor quer no proxy, assim como termina a sua execução no processo cliente. Para além desta instrução, o servidor sabe também processar mais duas, nomeadamente o **LIST** e o **DOWNLOAD** cujas funções responsáveis estas são "list_dir" e "isInDirectory" seguido da "send_file", respetivamente

5. Implementação das instruções LIST e DOWNLOAD

- **LIST** - A instrução LIST, desencadeada num processo cliente é interpretada e enviada ao proxy a string "LIST" através da função "write". O proxy repete este mesmo processo mas para o servidor. O servidor, por sua vez, lê o que lhe foi enviado através do socket pelo proxy e, sendo "LIST", chama a função "list_dir". Esta função, abre o diretório "Server_files" e percorre cada um dos ficheiros presentes. Caso esse ficheiro não seja "." ou "..", ficheiro usados na identificação da pasta em que nos encontramos e a pasta na qual esta está contida, irá escrever o nome deste pelo socket para o proxy, um a um. Importa salientar, que antes de começar e depois de terminar o envio destes nomes, é enviado o caráter especial «#define CHAR "$"» para controlar tanto o inicio como o fim da listagem. De igual modo, o proxy recebe o caráter especial, tanto no início, como no fim, e pelo meio os nomes dos ficheiros disponíveis. Este procedimento, ao encargo da função "transmit_dir" apenas tem utilidade de ler o que recebe do servidor e retransmitir para o cliente, desde que recebe o primeiro "$" até ao segundo, e último, "$". Por fim, o cliente, tendo um comportamento idêntico ao proxy, lê o que lhe é transmitido por este, porém, em vez de retransmitir, mostra no ecrã a informação que lhe chegou, desde o primeiro "$" até ao último "$", através da função printf(). Do ponto de vista do cliente, a função responsável por este procedimento é a "get_list".

- **DOWNLOAD** - A instrução DOWNLOAD, originada, interpretada e enviada ao proxy, também por um processo cliente, é apreciada por este é retransmitida para o servidor que procura no diretório "Server_files" pelo nome do ficheiro que é pedido pelo cliente, através da função "isInDirectory", e caso o encontre, envia a string "FOUND" para o proxy, que a retransmite. Após o envio desta mesma string, o servidor verifica qual o protocolo a utilizar e, recorrendo à função "send_file" abre o ficheiro em questão com a instrução fopen(), arranja o *file descriptor* deste ficheiro com fileno() e obtem os atributos deste ficheiro, tendo especial atenção o tamanho do mesmo. Tendo o tamanho, envia-o ao proxy, que por sua vez o envia ao cliente. Seguidamente, lê do ficheiro e escreve no socket, através da função fread() e write(), respetivamente, a informação deste até ter enviado toda a informação. Esta informação, dividida em chunks, é recebida pelo proxy, até que a quantidade de informação dos chunks que vão chegando, seja igual à ao tamanho do ficheiro previamente recebido. É neste

momento que, caso save = 1, os chunks são guardados, através da função fwrite() para um determinado *file descriptor* e caso haja losses definidas, estas são condicionadas por rand()%100 >losses, no caso do protocolo ser UDP. Este procedimento foi deixado ao encargo da função "transmit_file" do ponto de vista do proxy. Analogamente ao **LIST**, a função vai escrever no socket do cliente, que pediu o download do ficheiro, os chunks à medida que os recebe. De igual forma, o cliente vai ler os chunks, enquanto o tamanho destes seja inferior ao tamanho que recebeu preliminarmente, e escrevê-los para um *file descriptor* através da função fwrite(). O tempo de transmição do ficheiro é obtido através da função "gettimeofday" que é registada numa **struct timeval** inicio e fim, pouco depois de ler o "FOUND" e terminada assim que todos os chunks forem lidos, respetivamente. Tendo o inicio e o fim, trata-se apenas de fazer uma subtração para obter o tempo, quer em segundos, quer em microsegundos.

6. Requisitos de Segurança - Confidencialidade e Autenticação

Para garantir a confidencialidade implementamos um sistema de encriptação baseado em chaves simétricas que são atribuídas ao cliente e ao servidor e que permite a estes intervenientes encriptar e desencriptar uma mensagem

Com a mesma chave que deve ser mantida em segredo e que só estes devem ter acesso tal como de uma senha se tratasse.Utilizando as funções disponibilizadas na livraria de encriptação fornecida (sodium library) conseguimos através da sua API utilizar uma função de encriptação que recebe uma senha (conjunto de bytes) e encriptar uma mensagem segundo este padrão recorrendo ao algoritmo de encriptação "XSalsa20 stream cipher".

A autenticação é obtida através do uso de uma nonce (number once in a lifetime) que funciona como uma espécie de assinatura digital da mensagem que pretendemos enviar. Cada mensagem nova possui um identificador único representado por este nonce. E que é disponibilizado ao destinatário aquando o envio dos dados. Sabendo que o destinatário deve possuir a chave para conseguir desencriptar a nonce é um segundo requerimento para que se consiga obter a informação desencriptada garantindo que após o uso dela a mensagem foi assinada como recebida. A API disponibilizado serve-se do algoritmo de encriptação Poly1305 MAC.

O processo de encriptação no nosso programa serve-se de duas funções básicas uma que encripta a ficheiro com uma certa chave e que produz um ficheiro temporário com um identificador único que irá ser enviado ao proxy e posteriormente apagado após o envio. Este ficheiro contém uma cópia do ficheiro que foi requisitado pelo cliente mas encriptado.O proxy por sua vez vai transmitir o ficheiro encriptado ao cliente que através da função de desencriptação consegue recuperar os dados enviados lendo o ficheiro e fazendo o processo inverso. Precedido do envio do ficheiro encriptado é enviada a nonce para que através das funções disponibilizadas na API libsodium o cliente e o servidor consigam trocar entre si mensagens conseguindo ambos encriptar e desencriptar.