Trabalho Final de Teleinformática e Redes 2
Implementação de um Inspetor HTTP baseado em um Servidor Proxy

Nome: Roberta Renally da Silva Melo Matricula: 14/0161317

1 - Resumo

Este trabalho tem como objetivo explicar a implementação de um servidor proxy, de um spider, que consiste em enumerar todas as URLs subjacentes a uma URL definida formando sua árvore hipertextual. E por fim, de um cliente recursivo, que nos moldes do aplicativo wget faz o dump de todo o conteúdo a partir de uma URL raiz.

2 – Apresentação Teórica

2.1 – Servidor Proxy Web

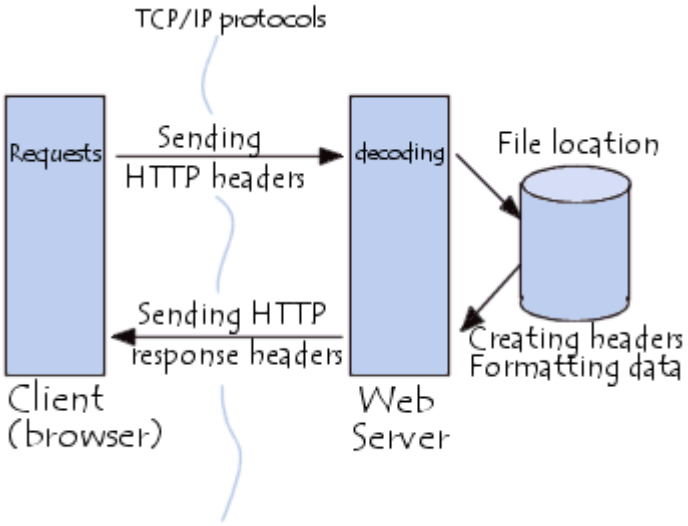
No contexto de redes de computadores, um servidor proxy é um servidor, que atua como intermediário entre pedidos de clientes que buscam recursos de outros servidores. Um cliente se conecta ao servidor proxy solicitando algum serviço, como por exemplo, um browser faz uma requisição de uma página web, um pedido é enviado ao proxy, que então envia esta solicitação ao servidor no qual o site é hospedado, devolvendo para o browser a resposta do servidor remoto.

Um servidor proxy pode ser um sistema de computador ou até mesmo uma aplicação e atualmente a maioria dos servidores proxy são web, que possuem como características o fácil acesso ao conteúdo da internet, proporcionar anonimato durante a navegação e a possibilidade de bloqueio de endereço IP. A utilização de servidores proxy surgiu como opção de ferramenta de segurança e controle, inclusive de acesso a páginas web indesejadas.

O servidor proxy permite controlar os serviços acessados na internet através do protocolo HTTP, sendo responsável pela gestão de acesso a sites e outras aplicações baseadas no protocolo.

2.2 – Protocolo HTTP

O protocolo HTTP (HyperText Transfer Protocol) é usado na camada de aplicação. O objetivo do protocolo HTTP é permitir uma transferência de arquivos localizados por uma URL entre um navegador (o cliente) e um servidor Web. O navegador efetua um pedido HTTP e o servidor trata o pedido e, em seguida, envia uma resposta HTTP.



Um pedido HTTP é um conjunto de linhas enviado ao servidor pelo navegador. Ele compreende uma linha de pedido, que é a linha que especifica o tipo de documento solicitado, o método a ser aplicado e a versão do protocolo utilizada. Também fazem parte do pedido HTTP os campos de cabeçalho do pedido, que é um conjunto de linhas facultativas que permitem dar informações adicionais sobre o pedido e/ou o cliente (navegador, sistema operacional, etc.).

Por fim, há o corpo do pedido, conjunto de linhas opcionais que devem ser separadas das linhas precedentes por uma linha vazia e que permite, por exemplo, um envio de dados com um comando POST durante a comunicação com o servidor, através de um formulário.

Exemplo de pedido HTTP:

```
--------------------------------------------------
Resquisicao HTTP do browser:
GET http://www.unb.br/ HTTP/1.1
Host: www.unb.br
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:60.0) Gecko/20100101 Fire
fox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: _ga=GA1.2.1106592708.1530662505; _gid=GA1.2.1410715515.1530662505; 3ec98
6f4f850e1d0bdcd00fd24b0c473=8p1nujj76mvitsb9fufcgk01ur; _gat=1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Uma resposta HTTP é um conjunto de linhas enviadas ao navegador pelo servidor. Ela compreende uma linha de status, que especifica a versão do protocolo utilizado e o estado do processamento do pedido através de um código e de um texto explicativo. Além disso, há os campos do cabeçalho da resposta, conjunto de linhas facultativas que dão informações suplementares sobre a resposta e/ou o servidor.

Exemplo de resposta HTTP:

```
HTTP/1.1 200 OK
Date: Mon, 31 Mar 2014 16:55:30 GMT
Server: Apache
Accept-Ranges: bytes
X-Mod-Pagespeed: 1.6.29.7-3566
Cache-Control: max-age=0, no-cache, must-revalidate
Vary: Accept-Encoding,Cookie
X-Node: askapacherackweb0
X-UA-Compatible: IE=Edge,chrome=1
Content-Length: 55069
Keep-Alive: timeout=6, max=46
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Content-Language: en
```

O protocolo HTTP utiliza o protocolo TCP da camada de transporte. Além disso, não guarda o estado, ou seja, se o cliente fizer um pedido de uma página web e minutos depois repetir o mesmo pedido, o protocolo HTTP vai enviar tudo de novo.

Além do que, o HTTP pode fazer uso de conexões persistente (onde todas as requisições e respostas dever ser enviadas por uma mesma conexão TCP) ou de conexões não persistentes (onde cada par de requisição/resposta deve ser enviado por uma conexão TCP distinta).

2.3 – TCP

O TCP (Transmission Control Protocol - Protocolo de Controle de Transmissão) é um dos principais protocolos da camada de transporte do modelo TCP/IP. Ele permite gerenciar os dados vindo da camada de redes, que utiliza o protocolo IP.

O TCP é um protocolo orientado para a conexão, isto é, ele define que antes das duas máquinas (cliente e servidor) troquem dados, devem primeiramente se apresentar trocando informações de controle para estabelecer uma conexão. Além disso, o TCP fornece um transporte confiável, ou seja, garante que todos os dados são enviados sem erro e na ordem correta.

O TCP também fornece um serviço de controle de congestionamento porque limita a capacidade de transmissão do processo (cliente ou servidor) quando a rede está congestionada.

3 – Desenvolvimento

3.1 – Arquitetura do Sistema desenvolvido

O trabalho foi desenvolvido em C mas utiliza algumas bibliotecas de C++ e foram implementadas as seguintes funcionalidades:

1 - Foi implementado um servidor proxy padrão que consegue receber as requisições HTTP do browser e envia-las para o servidor web, assim como receber a resposta HTTP do servidor web e repassa-la para o browser.

A implementação da arquitetura do servidor proxy está contida no arquivo web_proxy_server.cpp.

A arquitetura do servidor proxy e organizada da seguinte forma:

- ➔ Na função *int main (int argc, char *argv[])* cria o socket, que utiliza o protocolo IPv4, seta o endereço IP para 127.0.0.1 e seta a porta, que vai ficar escutando por requisições do browser. Lembrando que o número da porta deve ser passado como parâmetro quando for rodar o executável: /.proxy 8228
- ➔ Em seguida anexa um endereço local ao socket para o servidor proxy usando a função *bind( )* e torna o socket apto para receber requisições do browser.
- ➔ Em seguida, torna o servidor proxy apto para receber conexões do cliente e aloca uma fila, de tamanho, para conexões pendentes do browser por meio da função *listen().*
- ➔ Depois disso, o programa fica preso em um loop infinito e dentro do while bloqueia a execução do programa, até que exista um pedido de execução por parte do cliente por meio da função *accept();*
- ➔ Depois que recebe uma conexão do cliente, o programa chama a função *getRequestHTTP(),* que recebe a mensagem do cliente (browser) e armazena em um buffer.
- ➔ Mostra esse buffer para o usuário no terminal e se o usuário digitar a opção "apenas responder o browser" no terminal, a função *getRequestHTTP()* aloca memória para a um ponteiro da estrutura *"PedidoAnalisado"* e faz o parser da requisição HTTP recebida do browser, por meio da função *"Analise_do_Pedido()".*
- ➔ Em seguida chama a função *createServerSocket(),*que cria um socket para o servidor web, faz a conexão e retorna o identificador desse socket. A função *createServerSocket()* recebe como parâmetros o host da requisição HTTP e a porta.
- ➔ Depois é chamada a função *"sendToServerSocket()"*que é passado como parâmetros o identificador do socket do servidor web, a requisição HTTP e o total de bytes recebidos do browser. Tem como objetivo enviar para o servidor web a requisição HTTP do browser.
- ➔ Depois é chamada a função *receiveFromServer()* que é passado como parâmetros o identificador do socket para servidor web e o identificador do socket para o browser. Tem como função receber a resposta HTTP do servidor web por meio da função *iRecv = recv(Serverfd, buffer, sizeBuffer, 0)* num *while(iRecv > 0).*
- ➔ E cada pedaço da resposta HTTP recebida do servidor é armazenado em um buffer e enviado para o browser, por meio da chamada da função *sendToClientSocket ();*

A implementação da arquitetura do parser HTTP está contida no arquivo analyzer_web_proxy.cpp

- ➔ Que consiste basicamente na implementação de funções para alocar memória para um ponteiro do tipo "PedidoAnalisado", para realizar o parser de uma requisição HTTP, recuperar o buffer da requisição HTTP depois de ter sido feita o parser, desalocar memoria para o ponteiro do tipo "PedidoAnalisado" e etc.
- ➔ Essas funções são utilizadas para a implementação do servidor proxy e por isso são usadas no web_proxy_server.cpp.

A definição do parser HTTP está contida no arquivo analyzer_web_proxy.hpp

- ➔ E contém a definição da estrutura "PedidoAnalisado", assim como o cabeçalho das funções implementadas no analyzer_web_proxy.cpp

2 – Foi implementado um spider, que consiste em enumerar todas as URLs subjacentes a uma URL, digitada no browser, formando sua árvore hipertextual.

A implementação da arquitetura do spider está contida no arquivo spider.cpp

- ➔ Funciona da seguinte forma, depois que o servidor proxy intercepta a requisição HTTP do browser e o cliente seleciona a opção spider. O proxy não responde o browser porque vai estar ocupado fazendo o spider da url contida na requisição HTTP;
- ➔ Depois que o proxy chama o spider:

```
pedido = PedidoAnalisado_create();
Analise_do_pedido(pedido, mensagem, strlen(mensagem));
memset(what,'\0',5000);
strcpy(what,"./spider ");
strcat(what,pedido->host);
system(what);
```

- ➔ Na função *int main (int argc, char *argv[])* do código do spider.cpp, chama a função *get_mensagem(char *host)* que cria um socket para o servidor web que o host é hospedado, faz a conexão, monta a requisição HTTP, com método GET para aquele host e envia essa requisição para o servidor web.
- ➔ Em seguida, na mesma função, recebe a resposta HTTP do servidor web, armazena em um buffer e escreve em um arquivo index.html.
- ➔ Depois disso, dentro da função main, chama a função *spider(char *host)* que lê o arquivo index.html, extrai os links da página,e armazena em uma lista parametrizada do tipo: *queue <string> gquiz* os links que pertencem ao domínio da url raiz, além de verificar se o mesmo link já foi inserido na lista.
- ➔ Depois dentro da função *int main (int argc, char *argv[])* chama a função *recursivo(queue <string> gquiz)*, que para cada link inserido na lista *queue <string> gquiz*, repete os passo citados anteriormente.

3 – Foi implementado um cliente recursivo, que nos moldes do aplicativo wget faz o dump de todo o conteúdo a partir de uma URL digitada no browser.

A implementação da arquitetura do cliente recursivo está contida no arquivo cliente_recursivo.cpp

- ➔ Funciona da seguinte forma, depois que o servidor proxy intercepta a requisição HTTP do browser e o cliente seleciona a opção cliente recursivo. O proxy não responde o browser porque vai estar ocupado fazendo o dump recursivo do host contido na requisição HTTP;
- ➔ Depois que o proxy chama o cliente recursivo:

```
pedido = PedidoAnalisado_create();
Analise_do_pedido(pedido, mensagem, strlen(mensagem));
memset(what,'\0',5000);
strcpy(what,"./my_wget ");
strcat(what,pedido->host);
system(what);
```

- ➔ Na função *int main (int argc, char *argv[])* do código do cliente_recursivo.cpp, chama a função *directory(char *host)* que cria e navega pelos diretórios do host para armazenar os arquivos .html.
- ➔ Na função *int main (int argc, char *argv[])* chama a função *get_page(char *host)*, que cria um socket para o servidor web que o host é hospedado, faz a conexão, monta a requisição HTTP, com método GET para aquele host e envia essa requisição para o servidor web.
- ➔ Em seguida, na mesma função, recebe a resposta HTTP do servidor web, armazena em um buffer e escreve em um arquivo index.html.
- ➔ Depois disso, dentro da função main, chama a função *spider(char *host)* que lê o arquivo index.html, extrai os links html da página e armazena em uma lista parametrizada do tipo: *queue <string> gquiz;*
- ➔ Depois dentro da função *int main (int argc, char *argv[])* chama a função *recursivo(queue <string> gquiz)*, que para cada link inserido na lista *queue <string> gquiz*, repete os passo citados anteriormente.

Os arquivos que contem a implementação do trabalho são:

- web_proxy.cpp
- analyzer_web_proxy_cpp //implementa as funções de parser HTTP
- analyzer_web_proxy_hpp //biblioteca do parser http
- spider.cpp
- cliente_recursivo.cpp
- Makefile

3.2 – Como compilar

Depois de descompactar a pasta zipada, pelo terminal dever chegar ao diretório que contém os arquivos listados acima e digitar:

$ make

Vai gerar todos os executáveis, para rodar basta digitar no terminal:

$ ./proxy <port>

Exemplo:

$ ./proxy 8228

Verifique também, que o browser esteja configurado para utilizar o seguinte endereço IP: 127.0.0.1 na porta 8228.

3.3 – Problemas de Implementação

No Spider:

- Como o spider lista as urls subjacentes a uma url raiz, e repete isso de forma recursiva para cada url. O spider acaba inserindo urls repetidas, além do que também insere urls foram do domínio da url raiz.
- Nos testes que fiz, o spider não segue essas urls.

No Cliente Reecursivo:

- Uma limitação do meu cliente recursivo, é que em alguns links listados pelo spider, ele não consegue obter a página web, pois esses links não são de páginas estáticas. Então é uma limitação esperada porque o cliente recursivo só consegue obter páginas estáticas, ou seja, .html.

Não consegui implementar uma interface gráfica.

- A interface feita, foi em linha de comando.

Não implementei uma interface para que o usuário conseguisse editar as requisições HTTP recebidas do browser.

3.4 – Documentação do código

A documentação das funções está nos próprios arquivos .cpp, com uma descrição breve do funcionamento de cada função.

3.5 – Link do repositório do github

https://github.com/robertarenally/Trab-TR2.git

3.6 – Imagens da compilação/execução do programa

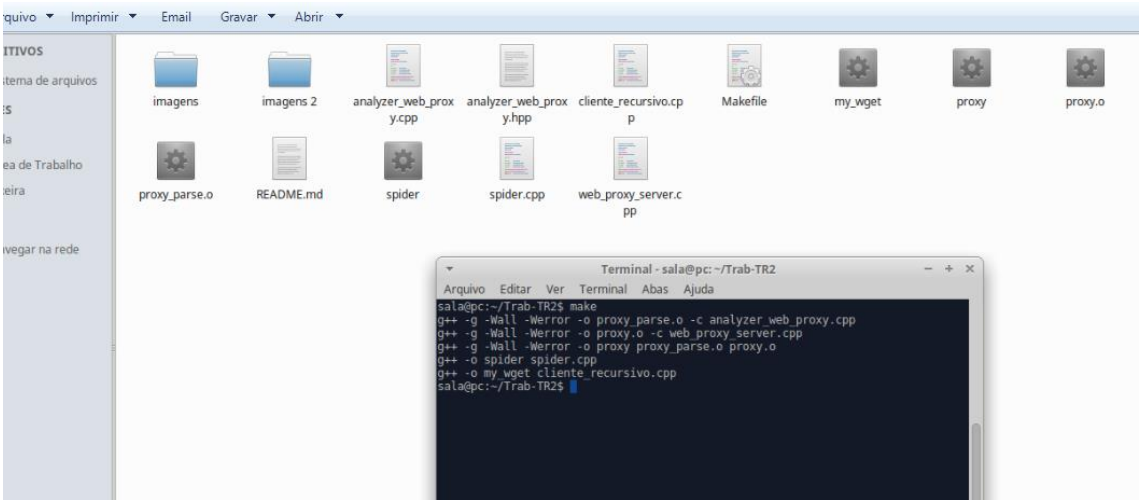


Figura 1- Compilando o código

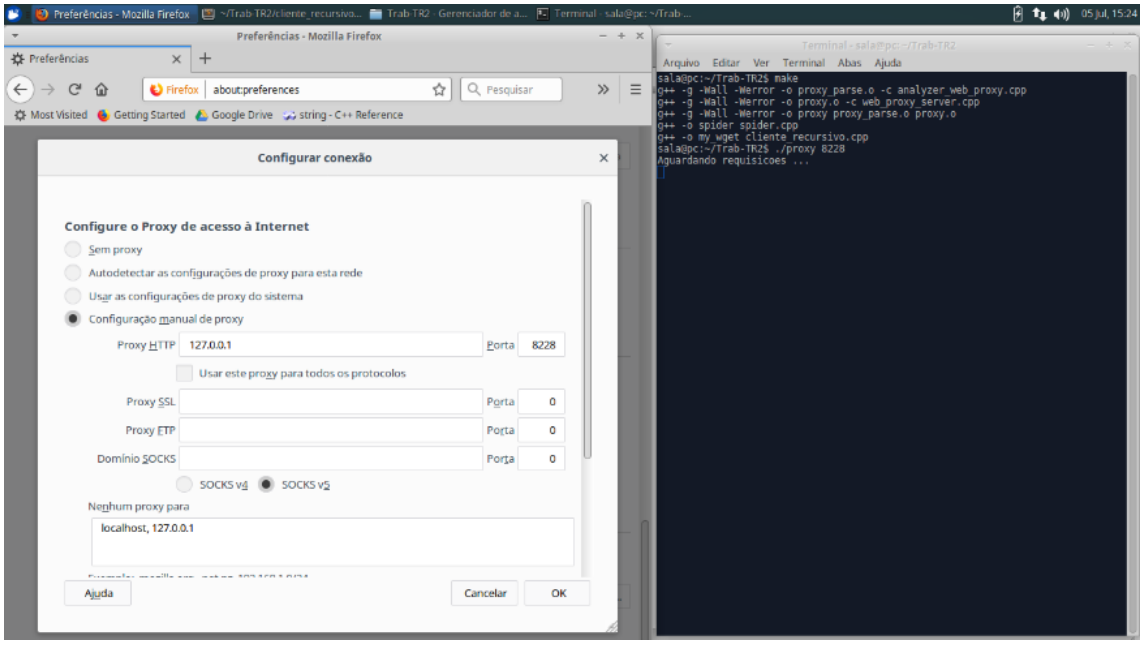


Figura 2- Configurando o browser para enviar requisições HTTP para o proxy

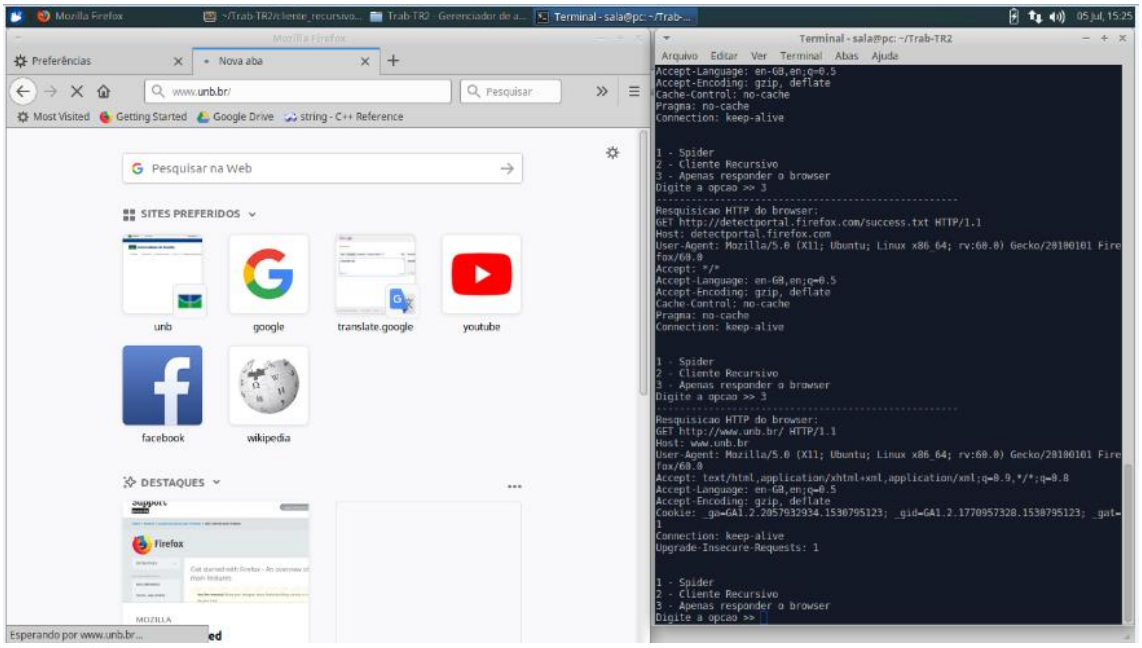


Figura 3 - Servidor Proxy intercepta a requisição HTTP do browser

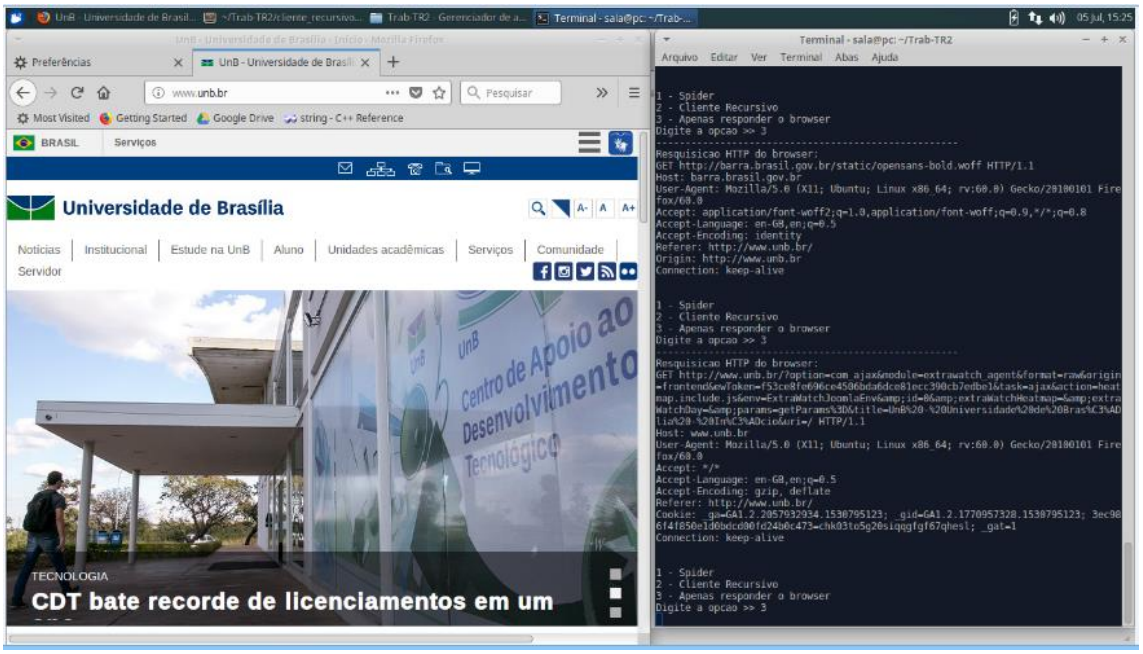


Figura 4 - Servidor proxy repassando a resposta HTTP do servidor web para o browser

Lembrando que para cada requisição HTTP interceptada pelo servidor proxy, o usuário deve selecionar a opção do menu:

```
------------------------------------------------------
Resquisicao HTTP do browser:
GET http://www.unb.br/ HTTP/1.1
Host: www.unb.br
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:60.0) Gecko/20100101 Fire
fox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: _ga=GA1.2.2057932934.1530795123; _gid=GA1.2.1770957328.1530795123; _gat=
1
Connection: keep-alive
Upgrade-Insecure-Requests: 1


1 - Spider
2 - Cliente Recursivo
3 - Apenas responder o browser
Digite a opcao >>
```

Figura 5 - Interface na linha de comando

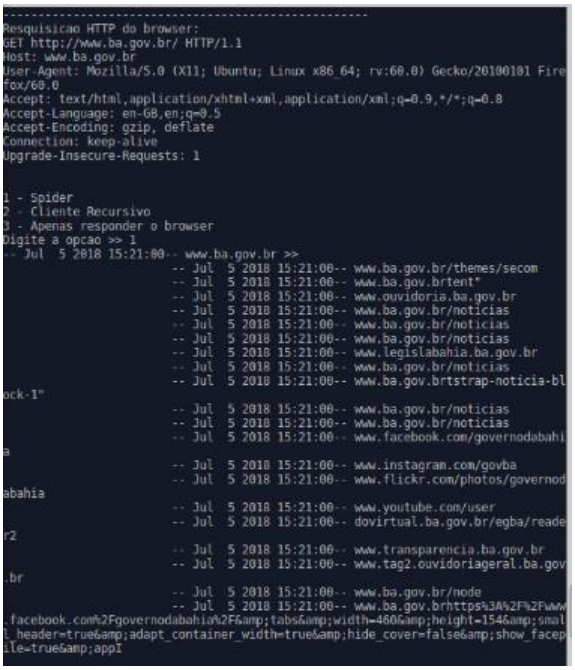


Figura 6 - Execução do Spider

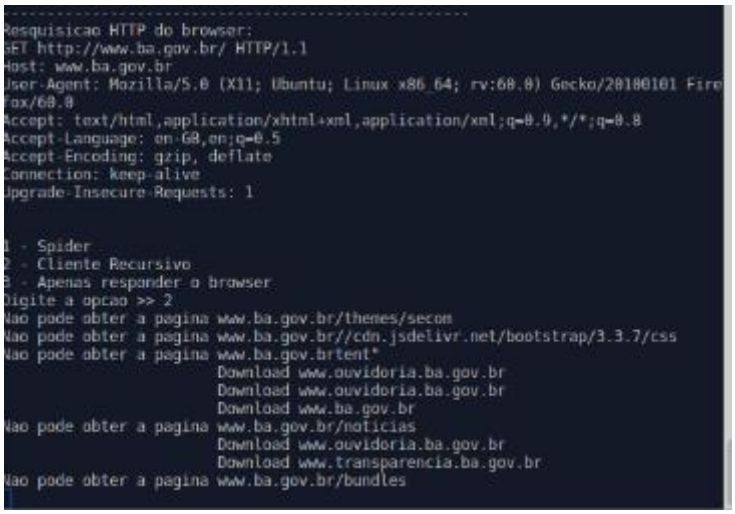


Figura 7 - Execução do Cliente Recursivo

4 - Conclusão

Esse trabalho nos permitiu aprofundar conhecimento sobre aplicações do que era visto apenas em sala de aula, de forma teórica. Foi possível ver o funcionamento dos protocolos e suas funções, como é o comportamento e importância de cada camada organizada. As maiores dificuldades enfrentadas estavam associadas à manipulação de algumas bibliotecas.

Por meio desse trabalho, foi possível colocar um servidor proxy para funcionar e entender melhor o funcionamento da Web, e protocolos HTTP e TCP. Além disso, houve a implementação de um spider e a tentativa de fazer o dump de todo o conteúdo do site, nos moldes do wget.