Universidade de Brasília
Teleinformática e Redes 2 - Turma A

Alunos:
Andre Garrido Damaceno - 15/0117531
Lucas Campos Jorge - 15/0154135

Professor: João Gondim

Inspetor HTTP baseado em Proxy Server

Objetivos:

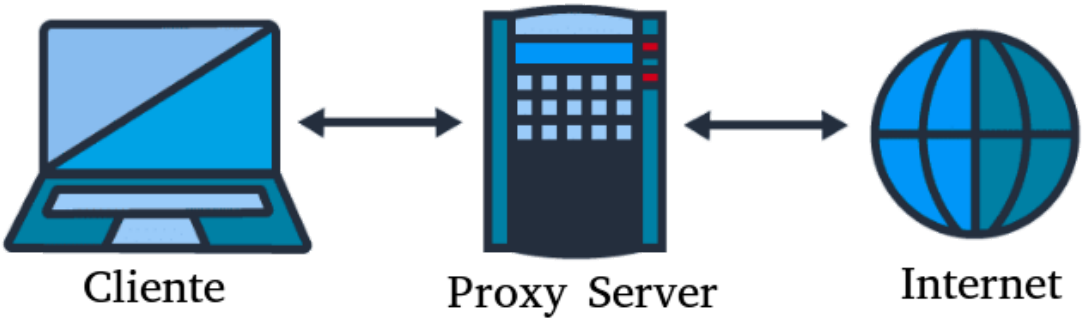
Aplicar os conhecimentos de camada de aplicação e transporte adquiridos em sala desenvolvendo um inspetor de tráfego HTTP baseado em um servidor proxy, com a opção de interceptação do tráfego no lado Cliente-Servidor ou Servidor-Cliente, opção de mostrar uma árvore hipertextual com as URLs obtidas do objeto (spider) e a opção de salvar todo o conteúdo de cada objeto da árvore hipertextual (cliente recursivo).

Introdução:

Proxy é um servidor que intermedia as requisições externas de um cliente, a fim de simplificar e / ou controlar sua complexidade, podendo também adicionar uma estrutura e encapsulamento aos sistemas distribuídos.

Neste trabalho em específico, foi feito o uso conjunto do protocolo HTTP (camada de aplicação) e do protocolo TCP (camada de transporte).

Uma comunicação feita entre um browser cliente e um servidor que redireciona ou armazena as páginas de internet usando estes protocolos, é uma aplicação exemplo de um servidor proxy.



O servidor proxy armazena em sua própria memória cópias dos objetos requisitados, com o intuito de reduzir a demanda de pedidos para a internet e aumentar a velocidade de resposta aos clientes, por esse motivo, também é chamado de cache web. O procedimento das requisições de um navegador no uso de um proxy-server funciona da seguinte maneira:

- O navegador abre uma conexão com o servidor proxy e envia uma requisição de um objeto utilizando o protocolo HTTP.
- O servidor proxy verifica se esse objeto está armazenado localmente em seu cache.
- Caso esteja armazenado, retorna o objeto ao navegador do cliente.
- Caso não esteja, o servidor proxy abre uma conexão com o servidor de origem e envia uma requisição HTTP do objeto. Após o servidor de origem enviar o objeto ao servidor proxy, o objeto é armazenado na cache e em seguida enviado em uma resposta HTTP ao navegador do cliente utilizando a mesma conexão inicial cliente-proxy.

O servidor proxy pode também ser usado para filtrar as requisições dos seus clientes, bloqueando requisições HTTP destinadas a hosts não desejados ou configurados a proibir acesso.

Protocolos

- **TCP (Camada de transporte):**

  A principal função da camada de transporte é a interligação lógica da comunicação entre os processos dos hospedeiros, sem a necessidade de intervenção ou conhecimento da aplicação. O protocolo TCP faz essa interligação garantindo a entrega confiável, ordenada e sem perda dos dados.

  TCP é um protocolo orientado a conexão, que é um processo prévio realizado para a comunicação (nesse caso chamado de handshake), que é um acordo inicial feito antes da troca de dados, para estabelecer a conexão.

  Uma das funções mais importantes do protocolo TCP é a multiplexação (organização do envio de dados de diferentes aplicações e sockets) e demultiplexação (organização do recebimento de dados de diferentes aplicações e sockets) das mensagens enviadas e recebidas por processos em um host. Além disso, há também o controle de congestionamento, que impede a quantidade excessiva de tráfego por uma conexão TCP em enlaces e roteadores, ao regular a taxa de envio de dados. Devido à suas características, é o protocolo da camada de transporte usado pelo protocolo HTTP para enviar e receber objetos.

- **HTTP (Camada de aplicação):**

A principal função da camada de aplicação é fornecer serviços para as aplicações de modo a separar a existência de comunicação em rede entre processos de diferentes computadores. O protocolo HTTP, chamado de Hypertext Transfer Protocol, realiza uma comunicação entre servidor-cliente na WEB através de trocas de mensagens padronizadas que definem a forma de envio e respostas, utilizando o protocolo da camada de transporte TCP. Com essa arquitetura baseada em camadas, a comunicação da aplicação fica mais simples, por não precisar se preocupar com os detalhes de transmissão da camada de transporte.

Uma requisição HTTP se inicia com o estabelecimento de uma conexão TCP com o servidor, que é feito pelas aplicações cliente-servidor por meio de suas interfaces sockets. O socket é a porta do processo do cliente-TCP para o cliente, e servidor-TCP para o servidor.

Após o estabelecimento da conexão TCP, o cliente envia mensagens de requisição (requests) HTTP para a interface socket, que entrega ao servidor suas requisições. Analogamente, o servidor analisa as requisições recebidas do cliente, e envia mensagens de respostas (responses) HTTP para a interface socket, que entrega ao cliente as respostas das requisições.

O protocolo HTTP não possui estado, visto que não mantém nenhuma informação a respeito dos clientes, e portanto todas as requisições de objetos feitas são requisitadas ao servidor.

O exemplo de mensagem HTTP a seguir exemplifica uma requisição GET, que é usada para requisitar o conteúdo de uma página WEB ou um objeto.

```
GET http://flaviomoura.mat.br/paa.html HTTP/1.1
Host: flaviomoura.mat.br
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:63.0) Gecko/20100101 Firefox/63.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

A primeira linha da requisição está pedindo ao servidor HTTP o envio do objeto /paa.html pela requisição GET, afirmando na mesma linha a versão do HTTP que está sendo utilizada na comunicação (HTTP/1.1).

A segunda linha informa o nome do servidor que hospeda o objeto requisitado, flaviomoura.mat.br.

A terceira linha informa a respeito do ambiente do cliente (navegador, sistema operacional, versão usada etc).

A quarta linha informa a respeito do tipo de retorno esperado pelo cliente (nesse caso um objeto do tipo texto/html).

A quinta linha informa a língua preferencial da requisição.

A sexta linha o tipo de codificação aceita (objetos comprimidos por exemplo).

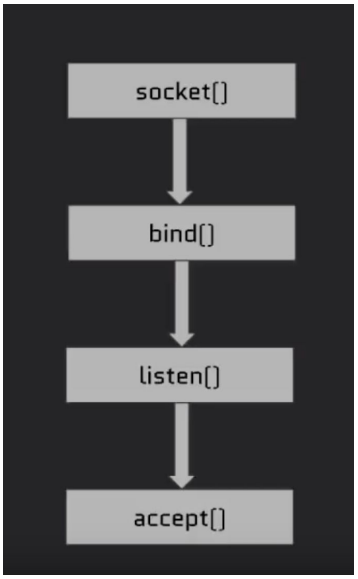
A sétima linha informa que o tipo de conexão é persistente.

Exemplo de mensagem de resposta HTTP:

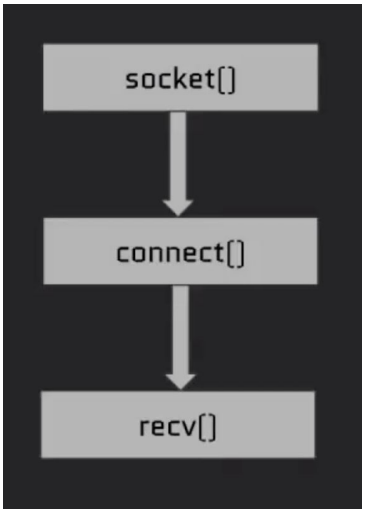
```
HTTP/1.1 200 OK
Server: GitHub.com
Date: Tue, 04 Dec 2018 03:43:36 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 32685
Vary: Accept-Encoding
Last-Modified: Mon, 03 Dec 2018 19:37:18 GMT
Vary: Accept-Encoding
ETag: "5c0585ee-7fad"
Access-Control-Allow-Origin: *
Expires: Tue, 04 Dec 2018 03:53:36 GMT
Cache-Control: max-age=600
Accept-Ranges: bytes
X-GitHub-Request-Id: CD80:485B:CD19FA:118495D:5C05F7E8
```

Similarmente à requisição, a resposta contém a versão do protocolo utilizada, status da mensagem (200 OK), diversas outras informações a respeito da resposta do servidor, e por fim (não mostrado na imagem), o objeto requisitado após o header.

Segue abaixo a lógica por trás da implementação de conexões servidor-cliente utilizadas no programa:



(servidor)



(Cliente)

Implementação do trabalho

Foi escolhida a linguagem C para a implementação do trabalho. Os sistemas implementados foram: Servidor Proxy, Spider, Cliente recursivo (dump), cache e um parser (analisador de strings).

- **Servidor Proxy**

  O servidor proxy feito age sobre a arquitetura cliente-servidor, podendo receber diversas requisições HTTP de diversos clientes que forem configurados para utilizá-lo. Todas as requisições são interceptadas pelo proxy, podendo ser analisadas e editadas pelo usuário, e redirecionadas pelo proxy para seus devidos servidores HTTP. Para o cliente, o proxy age como um servidor (pois é quem devolve para ele os requests), e para os servidores finais que o cliente fez um request, o proxy é um cliente (pois pede e recebe os itens requisitados).

  O proxy foi implementado pelo módulo ‘proxy.c’ em conjunto com o ‘parser.c’.

- **proxy.c**

O módulo proxy possui funções de conexão e requisições TCP/HTTP e possui as seguintes funções:

int create_socket() - Retorna um novo socket que pode ser usado pelo proxy tanto para a abertura de comunicação com o cliente, quanto para o servidor.

void config_address(int port, struct sockaddr_in *) - Configura o endereço de uma conexão.

int get_ip(char hostname[], char *) - Retorna um IP por referência a partir de um hostname fornecido.

int proxy_connect(int socket,struct sockaddr_in *) - Realiza a conexão com o system call Connect() (que conecta um socket até um endereço).

int proxy_receive(int socket, char *) - Recebe por referência um enviado pelo cliente ou servidor.

int proxy_bind(int socket, struct sockaddr_in *) - Realiza o binding entre um socket e um endereço.

int proxy_accept(int socket) - Chama system call de Accept(), aceitando uma requisição de um socket.

int proxy_send(int, char *) - Envia um pacote por referência em um socket.

void error(char *msg) - Imprime a mensagem de erro e finaliza o programa.

void get_server_response(char *, char *) - A partir de um request HTTP busca a resposta no servidor, e guarda a resposta em ‘./tmp/server_response.txt’. A função recebe um hostname e uma url.

int send_request() - Pega o request que se encontra em ‘./files/request.txt’, busca a resposta no servidor e armazena a resposta em ‘./files/reply.txt’.

- **parser.c:**

  O módulo de parser faz todas as manipulações de strings e análises de textos, para buscar os parâmetros necessários para a análise dos requests e responses HTTP. Foram implementadas nele as seguintes funções:

  parseData parseHtml (char *, int) - Função que dada uma resposta do servidor, separa os dados em uma struct ‘parseData’ em ‘header’ e ‘data’.

  FILE * CreateDataFile(char *) - Função de criação de um arquivo de acordo com o nome passado por referência.

  FILE * OpenDataFile(char *) - Função que abre um arquivo de acordo com o nome passado por referência.

  void GetFromText(char *, int, char, char *, int, char *, int) - Função que analisa um array passado por referência, em conjunto com seu tamanho, para a requisição de uma string a partir de um identificador de string inicial ‘parameter’, sendo indicado a partir daquela string onde deve ser começado a pegar o dado, e onde esse dado termina (identificado por um caracter indicador de fim também passado). A função retorna por referência a string passada, ou string vazia (todos os itens com ‘\0’), caso não ache o item.

void SaveToFile(char *,int, char *) - Salva um array de strings em um arquivo. Ambos o array de string e o arquivo são passados por referência.

void GetLinkFromHeader(char *, int, char *, int) - A partir de uma requisição HTTP, retorna por referência a url do site requisitado.

void GetHttpFileName(char *, char *, int) - A partir de uma url, retorna por referência o nome do arquivo (objeto de request) que será salvo pelo cache.

void GetHostFromHeader(char *, int, char *, int) - A partir de um request HTTP, retorna por referência o endereço do host requisitado. (utiliza a função '**GetFromText**' como auxílio)

void GetHttpMainFather(char *, char *, int) - A partir de uma url, retorna por referência o nome do domínio do site.

void GetHttpFolderPath(char *, char *, int) - A partir de uma url, retorna por referência o nome dos diretórios em que deve ser salvo o arquivo (objeto de request) do cache.

void DumpFile(char *) - A partir de uma url (usada para saber o diretório e o nome do arquivo), salva os dados encontrados em './tmp/server_request.txt' no local adequado da cache, removendo antes de salvar os dados o header da resposta HTTP do servidor.

void RemoveChar(char, char *, int, int) - A partir de uma string e um carácter de parâmetro, remove esse caracter da string no início ou / e no final (se requisitado no final pelo último argumento).

void DumpTemp(char *) - A partir do nome de um diretório passado por referência, remove o header da resposta HTTP do servidor, armazenada em './tmp/server_request.txt', e salva os dados em cache no diretório passado.

void RemoveTmp() - Remove a pasta e todos os itens do './tmp'

void CreateTmp() - Cria a pasta './temp'

void ClearString(char *, int) - Limpa todos os itens de uma string, colocando '\0' nos seus elementos.

void RemoveTmpHeader() - Remove o header da resposta HTTP salva em ‘./tmp/server_request.txt’, salvando em um novo arquivo ‘./tmp/new_server_request.txt’ com a resposta sem o header.

int StringContains(char *, char, int) - Verifica se uma string contém um caracter passado para a função.

int StringContainsAtEnd(char *, char, int) - Verifica se uma string contém um caracter no fim, passado para a função.

int StringLenth(char *) - Conta a quantidade de caracteres de uma string até o ‘\0’.

int LinkHasHttpOrHttps(char *) - Analisa a url passada por referência e verifica se ela inicia com ‘http’, ‘https’ ou nenhum dos dois, retornando um inteiro para cada resposta para a diferenciação entre os 3 tipos de resultados.

int LinkHasMailTo(char *) - Verifica se a url contém ‘mailto:’ ou ‘/mailto:’ no inicio da url.

void GetLinkWithoutHttp(char *, char *, int) - A partir de uma url passada, retorna por parâmetro a url sem ‘http’ ou ‘https’ no inicio.

FILE * GetHttpFromCache(char *) - A partir de uma url passada, analisa o diretório adequado e abre o arquivo da cache. Retorna o arquivo aberto. Caso o arquivo não exista na cache, o retorno é NULL.

- **Spider e cliente recursivo (dump)**

Ambos o spider e cliente recursivo foram implementadas pela função ‘**void Spider(char *, char *, int, spiderList **, int)**’, que dependendo dos parâmetros faz o dump ou apenas o spider. Essa função cria em memória duas listas, sendo uma para guardar as url’s já visitadas e a outra para guardar os links e a referência para os ‘pais’ desses links (para saber qual a árvore de referências das url’s no final). É possível limitar a profundidade do spider pelo último argumento (passando o número de níveis desejado), ou fazer a busca completa (passando -1).

As funções auxiliares para o uso do spider / dump foram implementadas no módulo ‘spider.c’.

void AddSpiderList(spiderList **, spiderList *, char *)- Cria a lista final do spider e também adiciona os links da lista, com a referência para as url's 'pais'.

void PrintSpider(spiderList *, spiderList *, int) - Imprime a árvore de referências das url's, do spider feito.

void AddVisitedList(visitedList **, char *) - Cria a lista de url's visitadas e também adiciona os links da lista, para que o spider saiba quais links já foram visitados e analisados o conteúdo.

void Spider(char *, char *, int, spiderList **, int) - Explicado acima. Realiza o spider / dump dependendo dos argumentos passados.

void DeleteSpiderList(spiderList **) - Apaga por completo a lista com a árvore do spider da memória.

int VisitedListContains(visitedList *, char *) - Verifica se a lista dos links visitados contém o link. Se tiver retorna 1, caso contrário 0.

void DeleteVisitedList(visitedList **) - Deleta toda a lista de links visitados da memória.

void SaveToFileSpider(FILE *, spiderList *, spiderList *, int) - Salva a árvore do spider em um arquivo de texto no computador, com o nome do arquivo de acordo com a url do domínio da url em que se foi feito o spider.

int downloadAndAnaliseLink(char *, int) - Função que analisa se o próximo link a ser analisado possui formato HTML, pois caso seja um arquivo diferente, ele apenas é adicionado na árvore, e não é analisado seu conteúdo (pois não haverão referências 'href' ou 'src' dentro de um arquivo não html).

void PrintVisited(visitedList *) - Imprime a lista dos links visitados.

- **Spider**

  É chamado pelo módulo 'aracne.c', que chama a função Spider descrita acima, com o parâmetro 'isDump' = 0, em seguida chamando também a função SaveToFileSpider, salvando a árvore dos links em um arquivo texto localizado em './files/spider/', sendo possível também

imprimir na tela a árvore recursiva das url's (recurso opcional). O spider é feito exatamente a partir da url visitada no navegador.

- **Cliente recursivo (dump)**

  É chamado pelo módulo 'aracne.c', que chama a função Spider descrita acima, com o parâmetro 'isDump' = 1, que salva todas as url's encontradas a partir do nó 'pai' na pasta '../cache', com o nome da pasta sendo o domínio do site.

- **Cache**

  O cache foi implementado pelo módulo 'parser.c' e chamado pelo 'aracne.c'. Sempre que o navegador faz uma requisição ao proxy, é passado a url requisitada para a função '**GetHttpFromCache**' e verificado no arquivo retornado se a url está ou não salva no cache. Caso esse arquivo exista, é perguntado ao usuário se ele deseja dar load na url usando o cache ou se deseja requisitar o objeto novamente para o servidor na internet.

  Para o armazenamento de objetos no cache, é necessário selecionar a opção de 'dump' no menu, que irá salvar em cache todos os objetos da árvore de referência criada a partir da url requisitada (usando a função '**Dump()**').

- **Funcionamento do programa**

O código fonte do projeto se encontra no seguinte link para download ou clonagem: https://github.com/lucascamposj/inspetor-http

Para compilar o código basta entrar na pasta /src do diretório principal baixado (http-inspector) e executar na linha de comando:

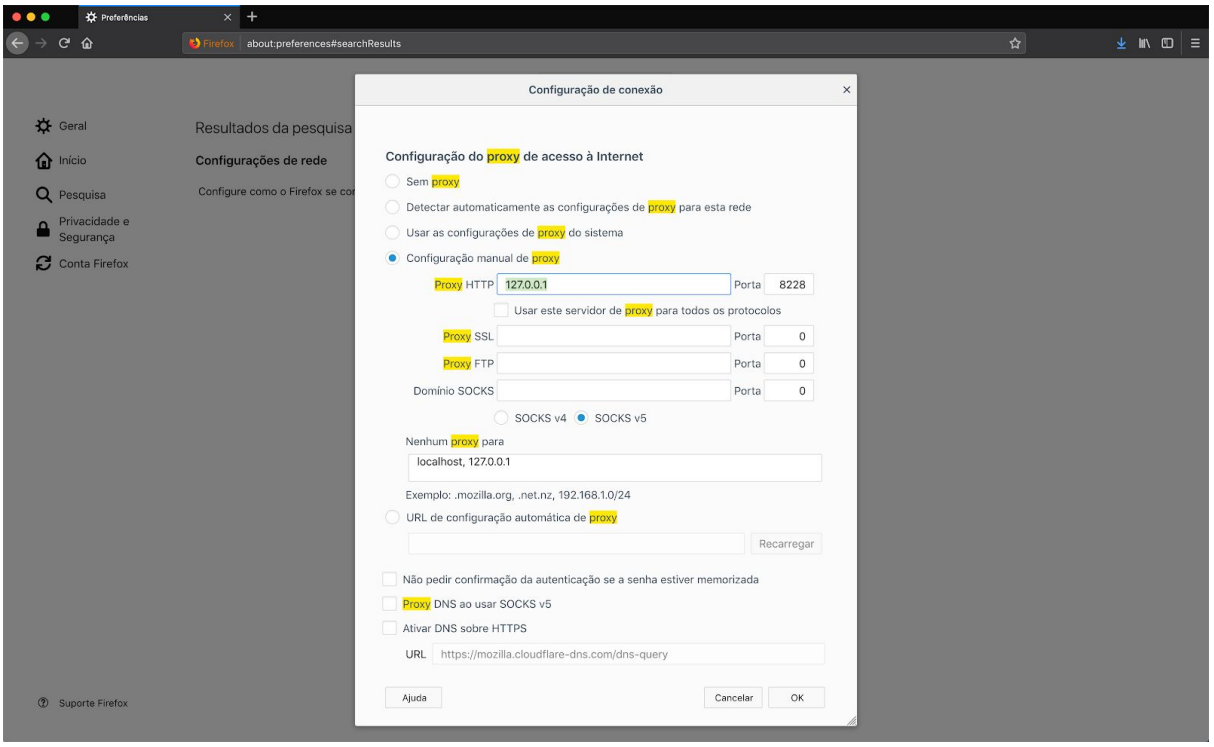
$ make

No fim da execução haverá um executável nesta pasta, /src.

Para executar o programa há a opção de selecionar a porta do proxy a ser utilizada, neste caso basta executar a seguinte na linha em seu terminal, sendo port a porta desejada:

./aracane -p port

Ao executar o programa, se iniciará a fase de espera por uma requisição do browser, a interceptação ocorrerá na primeira requisição.



ex: Configuração do proxy HTTP em um Firefox.

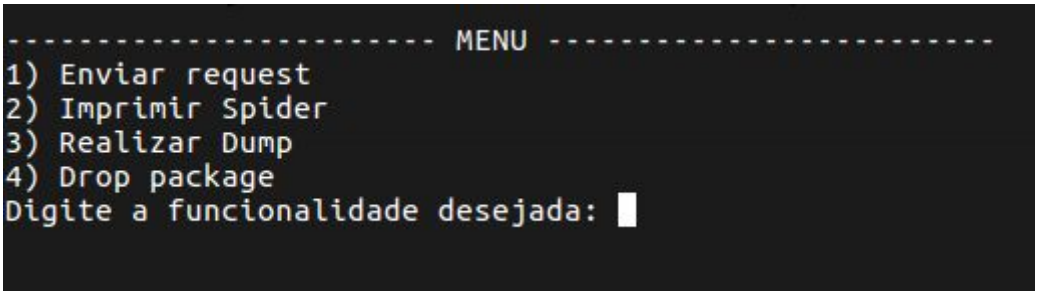
Para que a requisição seja realizada, deve-se alterar as configurações de proxy HTTP no seu navegador ou sistema operacional, como na imagem acima.

Com a interceptação da requisição, a captura será mostrada no terminal em conjunto com a opção de se editar essa requisição ou não. Caso seja selecionado que sim, abrirá uma instância do editor de texto “nano” para que haja a edição.

```
[~/Dropbox/UnB/Teleinformática e Redes 2/inspetor-http/src$ ./aracne
Porta escolhida: 8228
------------------------------------------------------
                  HTTP INSPECTOR
------------------------------------------------------
Aguardando browser...

Request do browser:

GET http://pudim.com.br/ HTTP/1.1
Host: pudim.com.br
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:63.0) Gecko/20100101 F
irefox/63.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Range: bytes=725-
If-Range: "353-527867f65e8ad"

Deseja editar o pedido antes de enviar para o servidor? (y/n):
```

```
  GNU nano 2.0.6            File: files/request.txt

GET http://pudim.com.br/ HTTP/1.1
Host: pudim.com.br
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:63.0) Gecko/20100101 $
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Range: bytes=725-
If-Range: "353-527867f65e8ad"
```

Logo após aparecerá um menu com as funcionalidades disponíveis:



1) **Enviar request**

Nesta opção, o programa enviará o HTTP interceptado e mostra a resposta do sistema. Esta resposta também é editável, antes de enviá-la de volta para o navegador.

2) **Imprimir Spider**

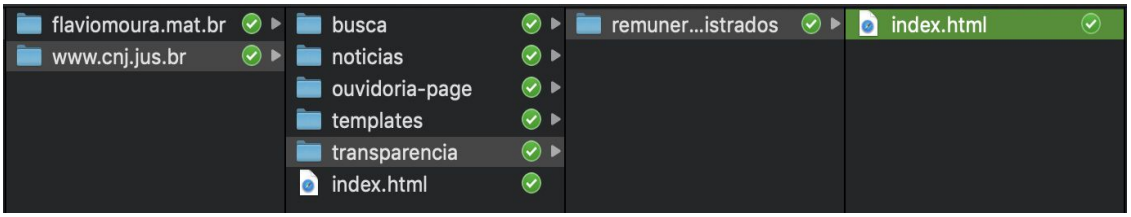
Nesta opção, o programa analisa o HTTP requisitado, chama a função '**Spider**' e mostra na tela a árvore de referências a partir do request do navegador.

A árvore é determinada pela ordem sequencial e "tab" antes do link

```
SPIDER:
.........................................

http://sitesinuteis.com.br/
        http://sitesinuteis.com.br/js/html5shiv.min.js
        http://sitesinuteis.com.br/css/style.css
        http://sitesinuteis.com.br/js/jquery.min.js
                http://sitesinuteis.com.br/
        http://sitesinuteis.com.br/
        http://sitesinuteis.com.br/js/script.js
```

3) **Realizar Dump**

Essa opção realiza o download de todos os arquivos do site referenciados pela url passada pelo navegador ao chamar a opção ‘dump’, similar à árvore recursiva do spider, porém realizando o download de todos os arquivos encontrados. Os arquivos são salvos na pasta ‘../cache/’ e são referenciados pela pasta com o nome do domínio da url passada como argumento, como visto na imagem abaixo.



4) **Drop package**

Essa opção apenas descarta o request feito pelo navegador, e volta para o estado de aguardando o browser em seguida.

Bugs e limitações

1. **Não é possível abrir com sucesso um arquivo baixado pelo dump como por exemplo do formato ‘.pdf’ ou ‘.jpg’. Arquivos com formato ‘.html’, ‘.css’, ‘.js’ e todos os tipos de arquivo que usam apenas texto funcionam corretamente.**

2. **Não foi feita uma interface gráfica para o programa. Foi apenas feito uma interface no terminal.**

3. **O cache só armazena arquivos na função ‘dump’, e não a cada request feito.**