Inspetor HTTP baseado em Proxy Server

João Victor de Souza Poletti
15/0132425
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Brasil
jvpoletti@gmail.com

João Marcelo Nunes Chaves
15/0132085
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Brasil
jmarcelonunes738@gmail.com

I. OBJETIVO

O presente trabalho teve como objetivo desenvolver um servidor *PROXY HTTP* para inspecionar o tráfego da *Web*. Nesse inspetor, o usuário, por meio do *browser*, envia um *URL HTTP* e o programa desenvolvido captura o tráfego para realizar as funções de *SPIDER* e *DUMP*.

II. INTRODUÇÃO

Um servidor *proxy* é intermiador do tráfego entre o computador e a *Internet*. Tal ferramente pode ser utilizada para diversas finalidades, como por exemplo:

- **Compartilhar a conexão com a *Internet* mesmo quando existe apenas um IP disponível**: Ou seja, o servidor *proxy* é o único realmente conectado à *Web*, os outros PCs acessam através dele. Como consequência ele oculta seu endereço IP, mascarando assim sua identidade pessoa *online*.
- **Melhorar o desempenho do acesso através de um cache de páginas**: O *proxy* funciona como uma cache de páginas e arquivos, armazenando informações já acessadas. Quando alguém acessa uma página que já foi carregada, o *proxy* envia os dados que guardou na cache, sem precisar acessas a mesma página repetidamente. Como consequência, acaba-se economizando bastante em termos de banda e fluxo na rede como um todo, bem como gera um acesso mais rápido para o cliente.
- **Bloquear o acesso a determinadas páginas**: Com isso, é possível ter um controle do fluxo do conteúdo acessado na rede, podendo import restrições do que pode ou não ser acessado, uma vez que a porta de acesso à rede local é o próprio servidor *proxy*. Ou seja, pode-se impor limitações de acesso com base no horário, *login* e endereço IP da máquina, por exemplo.

Para fins ilustrativos, a funcionalidade de um servidor *proxy* pode ser observada por meio da Figura 5. Com a imagem fica bastante visível o cunho intermediador do servidor *proxy* entre o tráfego da *Internet* e dos usuários.

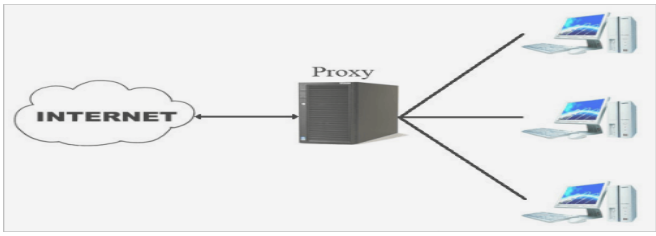

Fig. 1: Exemplificação de um servidor *proxy*

III. EMBASAMENTO TEÓRICO

As redes de computadores podem ser dividias em 5 camadas de acordo com o modelo TCP/IP. Sua estrutura está visível na Figura 2.

Fig. 2: Camadas de rede TCP/IP.

O projeto desenvolvido focou, principalmente, nas camadas de transporte e de aplicação, uma vez que os protocolos *TCP* e *HTTP* se encontram nas camadas 4 e 5, respectivamente.

A. PROTOCOLO TCP

O protocolo TCP (*Transmission Control Protocol*) está presente na camada de transporte e é orientado à conexão, ou seja, antes de uma aplicação conseguir enviar dados para outro processo de aplicação, os dois processos devem efetivar sua comunicação por meio de um *handshake* - devem enviar segmentos preliminares entre si para estabelecer parâmetros da transferência de dados que irá ocorrer a seguir. Além disso, este protocolo roda, apenas, nos *end-systems* e não

é utilizado nos nodes intermediários (logo, não mantém estado da conexão TCP). Ademais, é um protocolo *FULL-DUPLEX*, podendo realizar o envio de segmentos, bem como a recepção de pacotes transmitidos pelo outro lado ao mesmo tempo. Por fim, é um protocolo POINT-TO-POINT, ou seja, entre um único *sender* e um único *receiver*, não permitindo *multicasting*.

B. PROTOCOLO HTTP

Por outro lado, o protocolo HTTP (*HyperText Transfer Protocol*) é implementado na camada de aplicação TCP/IP. Esse protocolo se caracteriza por possuir duas programas: cliente e servidor. Esses dois programas trocam informações utilizando o protocolo HTTP que define uma série de regras sobre como essas mensagens trocadas devem estar escritas. Esse protocolo, como já dito anteriormente, se utiliza do protocolo TCP para principalmente se beneficiar da transfêrencia confiável de dados. A transmissão entre um cliente e um servidor, inicialmente, se dá por uma conexão TCP entre o cliente e o servidor. A partir daí, são utilizados os sockets (figura 3). O cliente envia uma requisição HTTP para seu socket que passa essa mensagem via TCP para o socket do servidor que recebe esse pacote. Da mesma forma, o servidor pode enviar mensagens de resposta para os clientes através do seu socket.

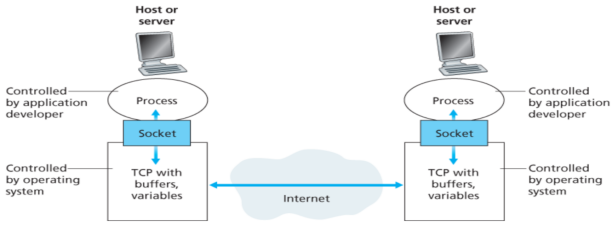

Fig. 3: Exemplo do funcionamento dos sockets cliente-servidor

O protocolo HTTP é definido por certos padrões impostos nas mensagens trocadas entre cliente servidor. Na primeira linha da mensagem, temos o chamado request line. Após isso, todas as outras linhas são parte do cabeçalho da mensagem. Nesse cabeçalho, encontramos o nome do host (hostname), o tipo de conexão (persistente ou não persistente), a língua do objeto requerido ou enviado e mais algumas informações. Abaixo, nas figuras 4 e 5, encontramos um exemplo request e um de response no protocolo HTTP.

```
GET http://aeridinae.e-monocot.org/ HTTP/1.1
Host: aeridinae.e-monocot.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:63.0) Gecko/201001$
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
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Fig. 4: Exemplo de uma mensagem do tipo request em HTTP

```
HTTP/1.1 200 OK
Date: Mon, 10 Dec 2018 01:58:13 GMT
Server: Apache
X-Content-Type-Options: nosniff
X-Powered-By: PHP/5.6.38
X-Drupal-Cache: MISS
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Cache-Control: public, max-age=10800
X-Content-Type-Options: nosniff
Content-Language: en
X-Frame-Options: SAMEORIGIN
X-Generator: Drupal 7 (http://drupal.org)
Last-Modified: Mon, 10 Dec 2018 01:58:13 GMT
Vary: Cookie,Accept-Encoding
Content-Type: text/html; charset=utf-8
X-Varnish: 1038582633 1032127055
Age: 4248
Via: 1.1 varnish-v4
ETag: W/"1544407093-1"
```

Fig. 5: Exemplo do header de um response em HTTP

C. COMPUTER NETWORKING: A Top-Down Approach, 7th Edition

A teoria foi estudada com ajuda do livro: Computer Networking: A Top-Down Approach, 7th Edition. Ele foi utilizado diversas vezes para solucionar dúvidas e para o entendimento do protocolos já listados acima.

D. SLIDES DE SALA

Por fim, os slides disponibilizados pelo professor também foram utilizados como guias, bem como para a solução de dúvidas.

IV. TECNOLOGIAS UTILIZADAS

Para a efetivação deste trabalho foram utilizados programas de edição de código, como o *Atom* e o *Xcode*. Além disso, o código foi desenvolvido na linguagem de programação C - utilizando as bibliotecas de *socket* - e versionamento feito utilizando o *GitHub* (link: https://github.com/joao96/inspetorHTTP). Por fim, foi utilizado o *browser* Mozilla Firefox para executar o papel de cliente.

V. ARQUITETURA DO SOFTWARE

O software desenvolvido foi separado em módulos. Primeiramente, criamos um arquivo header para realizar a conexão dos diferentes arquivos .c existentes e para declarar variáveis globais necessárias para o desenvolvimento do software. Feito isso, foram criados três arquivos: main.c, spider.c e dump.c. Cada um desses arquivos realiza unicamente um tipo de funcionaliade, por exemplo, o arquivo spider.c apenas terá a codificação relacionada à implementação do spider. Já no arquivo main.c o proxy está implementado juntamente com a funcão do inspector. Por fim, no dump.c temos a codificação relacionada ao dump (download dos objetos de certa página).

VI. IMPLEMENTAÇÃO

A. MAIN

O desenvolvimento do programa pode ser divido em 3 partes. Inicialmente foram implementados *sockets* para realizar a conexão com o servidor na porta determinada no início do programa. Para sua criação, utilizamos a biblioteca <sys/socket.h>. Com ela utilizamos as funções *socket*(), para a criação do socket em si, *bind*() para associar o *socket* com o endereço local. A seguir, colocamos o *socket* para escutar por meio da função *listen*(). Ao chegar uma requisição feita pelo *browser*, um *socket* é criado e reservado para o uso do cliente

e a conexão é aceita. Com a conexão aberta, recebemos a resposta enviada do servidor ao *browser*.

A descrição acima pode ser encontrada no módulo **main.c**. Claro, essas funcionalidades se encontram dentro de um *do-while*, visto que o servidor *proxy* pode receber várias requisições, uma vez que foi implementado o procolo *HTTP* persistente. Uma vez feita a conexão entre cliente e o servidor, escreve-se o *request* em um arquivo .txt para, então, realizar o *parsing* dessa mensagem. É válido lembrar, ainda, que a funcionalidade de permitir o usuário editar o *request* também foi implementada, utilizando a função *system* da linguagem C.

- **void parsing(char *, char *, char *)**: esta função recebe o *request* como parâmetro, uma *string* correspondendo ao *URL* e outra ao *Host*, retira o *URL* da mensagem - percorrendo-a utilizando a função *strstr* de C -, bem como o *Host* e salva-os nessas variáveis de entrada.

Feito o tratamento da mensagem de *request*, chama-se outra função para obter o IP do *Host* encontrado na mensagem para, então, escrever um *GET* requisitando a página .html do *browser*.

- **int get_host_by_name(char *, char*)**: recebe o *URL* e o *Host* retirados do *request* e checa se o *Host* pode ser alcançado. Caso seja possível, cria-se um novo *socket* para esse cliente e tenta-se realizar a conexão. Uma vez estabelecida, cria-se uma requisição com o método GET para esse *URL* e *Host*. Feito isso, utiliza-se a função *write* de C para enviar pelo *socket*.

Realizado o passo anterior, lê-se a mensagem de resposta e a escreve em um arquivo .txt. Esta mensagem irá conter o *.html* da página. Claro, a funcionalidade de permitir a edição dessa resposta também foi implementada utilizando a função *system* da linguagem C. Agora, como o *proxy* deve ser transparente e permitir o fluxo de ida e volta, o código reenvia esse .html para *browser*, uma vez que todo o tratamento foi finalizado.

Em seguida, o usuário entra em um menu para decidir se deseja realizar a função *SPIDER* ou *DUMP* dessa conexão. Caso não queira, basta digitar 4 no menu e o programa fechará essa conexão, bem como o *socket* desse cliente e estará pronto para receber outra (servidor continua a ouvir o *socket*).

Ainda na **main.c**, temos as funções **imprime_arvore** e **zera_arvore**.

- **void imprime_arvore(struct Arvore *, int)**: Esta função é utilizada mais para frente durante a execução do programa, mas ela efetiva o percorrimento da estrutura de dados utilizada para manter controle da hereditariedade dos *hrefs* encontrados e retirados das páginas *.html*. Além disso, é utilzada também para realizar o preenchimento de arquivos .txt permitindo, assim, uma visualização mais clara da árvore hipertextual.
- **void zera_arvore(struct Arvore *, int)**: Por outro lado, essa função é responsável pela eliminação de *hrefs* da árvore, liberando o conteúdo de cada nó da árvore. Isso é feito quando a conexão com um determinado cliente é encerrada, logo a árvore hipertextual dele dele ser apagada da estrutura de dados com o objetivo de não interferir na execução de uma nova conexão com outro cliente.

Fig. 6: Imagem tirada do inspector. Aqui, "Pular para o conteúdo principal"é editado para "Pular para o conteúdo principal - TR2"

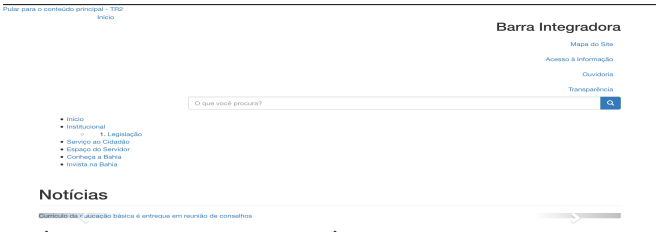

Fig. 7: Imagem tirada do browser após a alteração

Fig. 8: Imagem tirada do menu em execução

B. SPIDER

A funcionalidade do spider, consiste em listar todas as url's subjacentes à uma página já baixada e realizar sua arvóre hipertextual. Para implementar tal funcionalidade, criamos uma arvóre que possa ter múltiplos filhos, para que assim, possamos definir uma herança com relação às url's em cada página. Um dos passos principais nessa etapa do programa é varrer o html de uma página e retirar dele todas as referências a outras url's, ou seja, é necessário procurar todos os href's contidos em uma página.

- **void Spider(char *, char*, char*, struct Arvore, char *)**: A função spider recebe o *url* e o *hostname* extraídos do *response HTTP* recebido. Ela também recebe como parâmetro uma *string* com o nome do diretório que o *html* baixado será armazenado, a árvore a ser utilizada e, por fim, o diretório raiz que nada mais é que o nome do diretório da página a qual o *spider* será feito.

Nessa função, são abertos dois arquivos: index.txt e html_tree.txt. O primeiro corresponde ao *html* baixado de cada página, nesse arquivo será feita a varredura de *href's* já listada acima. Já o segundo arquivo será onde todos os *href's* dessa página serão armazenados. Durante a execução dessa função são feitas checagens para achar os *href's* nos arquivos e esses *href's* são escritos no arquivo html_tree.txt.

- **int walk_tree(char *, struct Arvore *)**: A função walk_tree recebe uma string contendo um href já encontrado e também recebe a raiz da árvore. Essa funcão percorre a árvore e checa a existência do href, recebido como parâmetro, na árvore. Caso a árvore ja possua essa ocorrência ela retorna TRUE. Por outro lado, caso não encontre nenhuma referência retorna FALSE.
- **int make_tree(char *, struct Arvore *, char *, char *, char *)**: A função make_tree recebe uma string contendo um *href* encontrado, recebe a raiz da árvore, a string contendo o nome do arquivo a ser criado para salvar as referencias (*href*'s), o *hostname*, a *string* contendo o diretorio atual e o diretorio raiz desse *SPIDER*. Essa função cria novos filhos na árvore caso não exista a referência de *href* passada como parâmetro, já presente na árvore (isso se torna possível com a utilização da walk_tree). Feito isso, essa função cria um request *HTTP* solicitando o objeto *html* referente a um determinado *href*. Feito isso, é criado um novo diretório para o armazenamento do *html* e do html_tree.txt dessa referência encontrada. Por fim, a função *spider* é chamada até no máximo o número de níveis N (setado no arquivo functions.h).
- **void initialize_node(struct Arvore *)**: Por fim, essa função tem como único objetivo inicializar todos os filhos de um determinado pai (passado como parâmetro) com NULL.

Fig. 9: Exemplo de tree gerada

C. DUMP

O *dump* consiste em realizar o *download* dos arquivos descritos no *spider* para visualizar a *url* e sua árvore localmente. Para isso, foi implementado o dump.c que basicamente recebe a árvore do spider e realiza o *download* de todas as refêrencias contidas no *url*.

- **void dump(char *, char *)**: Esta função recebe a *string* da *url* do objeto a ser baixado e a *string* do *hostname* da *url* raíz. Primeiramente, o arquivo html_tree.txt é aberto e varrido para obter todos as *url's* referentes à *url* raiz. Feito isso, foi criado um socket para realizar a comunicação entre o cliente e o servidor. Com isso, os *requests* foram feitos tratando *strings* e utilizando os dados dos *url's* obtidos no arquivo html_tree.txt e sempre utilizando o *hostname* que foi passado como parâmetro da função. Por fim, utilizamos a função *write*, da biblioteca de *sockets* para enviar a requisição, via TCP, para o servidor. Agora, utilizamos a função *read*, também da biblioteca de *sockets* para continuamente ler o *buffer* e extrair as informações passados pelo servidor. Essas informações sao extraídas e salvas de acordo com sua extensão utilizando um *fopen*, também é válido salientar que cada arquivo é salvo no diretório de acordo com o nome presente em sua *url*.

VII. CONCLUSÃO

Por meio do presente trabalho foi possível observar e colocar em prática muitos dos conceitos aprendidos durante as aulas. Além disso, unir esses princípios de redes de computadores para desenvolver a ferramenta de inspeção *HTTP* ajudou bastante para o entendimento geral do funcionamento dos protocolos *HTTP* e *TCP*. No entanto, é justo descretizar as difulcudades e limitações do código desenvolvido. Uma das principais limitações do programa é o funcionamento do *SPIDER*. Apesar dele retornar uma árvore hipertextual no arquivo criado, essa árvore está incompleta, não contendo alguns *hrefs* que foram recolhidos. Acreditamos que esse problema tenha ocorrido porque não estamos realizando o percorrimento da estrutura de dados de forma correta, ou seja, não percorremos efetivamente todos os nós da árvore. Uma consequência desse funcionamento indesejado do *SPIDER* é que o próprio *DUMP* usará essa árvore incompleta para realizar o *download*. No entanto, o *DUMP* funciona de forma correta, estando limitado apenas pelo *SPIDER*, uma vez que ele utiliza a saída deste como entrada. Outra particularidade é que o programa é encerrado ao receber um método *POST*. Por último, deve-se tomar cuidado ao examinar os arquivos obtidos por meio do *dump*, uma vez que esses arquivos não estão separados dentro de pastas, ou seja, não existe uma pasta de um cliente com o seu conteúdo baixado, estando todos os arquivos dentro do próprio diretório raiz do projeto.

VIII. EXECUÇÃO DO PROGRAMA

Para executar o código desenvolvido basta acessar `https://github.com/joao96/inspetorHTTP` e clonar o repositório para a máquina local. Feito isso, abrir o terminal no diretório criado e digitar *gcc -o aracne main.c spider.c dump.c*. Uma vez compilado, basta executar o programa com o comando *./aracne.*. Caso deseja-se alterar a porta que será utilizada (por padrão é a 8228) é só digitar o comando anterior da seguinte forma: *./aracne -p <porta>*.

Os sites utilizados para teste foram:

- **www.ba.gov.br**;
- **http://aeridinae.e-monocot.org/**;
- **http://flaviomoura.mat.br/paa.html**.