Teleinformática e Redes 2 - Projeto

Paulo Borges Teixeira Neto - 15/0143753
Luiz Gustavo Rodrigues Martins - 13/0123293

Como compilar e executar

Duas alternativas

- Abrir o projeto pelo QT creator (precisa reconfigurar o projeto por causa dos .user) e executar;
- Entrar no diretório, compilar com “make” e executar: **qtproxy -p <porta>**

Caso a porta não seja informada corretamente, a porta padrão será 8228.

Para maior eficiência do *proxy*, é recomendado testar os requests com *Connection: close*, pois a resposta demora para chegar quando se deixa o *keep-alive*.

Apesar do *dump* tratar imagens e binários, o *proxy* não trata. Apagar a linha do request que aceita gzip para o funcionamento correto.

No *spider* e *dump*, escrever a URL correta com http://

Introdução teórica

Proxy server:

Um servidor *proxy* é usado para interceptar requisições e respostas HTTP. As requisições de um *browser* são interceptadas pelo servidor *proxy* em uma determinada porta e podem ocorrer diversas operações, como edição de campos e dados da requisição. O servidor *proxy* então manda essa requisição, modificada ou não, para o servidor de destino, que gera um *response* que também é interceptado pelo servidor *proxy*, que pode modificar e realizar operações com essa resposta, antes de enviar para o *client* (*browser*).

TCP:

O TCP é um protocolo da camada de transporte responsável por uma conexão lógica entre aplicações diferentes. Então, sistemas diferentes mantêm o mesmo estado na camada de transporte (por meio de *sockets*). As camadas abaixo (rede, enlace e física) desconhecem essa comunicação.

Este protocolo, diferente do UDP, é orientado à conexão, sendo necessário um *handshake* entre as duas aplicações para se estabelecer uma conexão antes da comunicação em si. Com isso, a comunicação passa a ser bidirecional.

O TCP provê confiabilidade na entrega dos dados, tendo mecanismos de detecção de erros, retransmissão e controle de fluxo. As informações do TCP são encapsuladas em datagramas IP.

HTTP:

HTTP é o protocolo da camada de aplicação para a WEB. Segue o modelo cliente servidor de forma que o cliente normalmente é um *browser* que manda requests para um servidor e recebe e exibe respostas para o usuário. O servidor é um servidor *web* que envia objetos em resposta a *requests* que recebe. Este protocolo roda em cima do TCP e é sem estado. O *server* não mantém informação sobre *requests*.

A mensagem tem a forma:

- Linha do *request*
- Cabeçalhos
- Linha vazia
- *Body* (opcional, muito usado no POST)

Funcionalidades implementadas

Implementamos, usando C++ e interface gráfica QT 5 um software que possui 3 principais funcionalidades:

- *proxy* HTTP
    - intercepta requisições e respostas HTTP em uma porta pré-definida que por padrão é 8228, mas pode ser definida ao executar o programa com o comando: **qtproxy -p <porta>.** O software permite edição das requisições e respostas recebidas
- *Spider*
    - Lista as urls encontradas em um site, dada uma determinada profundidade especificada na interface.
- *Dump*
    - Realiza o download das páginas e recursos como imagens e scripts a partir da resposta de um *spider*. Também é possível especificar uma profundidade.

Arquitetura do sistema desenvolvido

O sistema foi desenvolvido em C++ utilizando QT5 pela IDE QT Creator. Utilizou-se uma arquitetura *multithreading* na qual o *thread* principal é a interface gráfica, outro *thread* para o *proxy* e outro para as *webtools*, que seriam o *proxy* e o *spider*.

Possui 4 principais módulos:

mainwindow: é responsável pela UI e as conexões de campos e botões da interface com as ferramentas (observar os *connects* no construtor). A principal função desenvolvida neste módulo é a setScreen, responsável por terminar a *thread* atual e trocar de tela, iniciando a nova *thread*. Utilizou-se o *stacked widget* do QT para essa organização de páginas diferentes.

proxy: é a classe que implementa o *proxy*. Possui todas as funções de operação de *sockets* e sinais que se comunicam com a GUI, as conexões são feitas em mainwindow, mas os sinais são emitidos em doWork, que é onde o *proxy* é implementado. A forma utilizada pelo *proxy* para esperar o botão é "travando" a execução com a função implementada waitButton, que é um loop que só termina quando buttonFlag for *true*, o que acontece quando se aperta o botão. Para evitar excesso de processamento, utilizou-se msleep e essa função também processa a fila de eventos em cada iteração para que o mecanismo de *signals* e *slots* funcione.

webtools: é a classe em que se implementa o *spider* e o *dump*, as funções que executam essas funcionalidades são runSpider e runDump respectivamente.

httputils: Funções para manipulação de dados e operação dos protocolos, como um *parse* para o *request*, iniciar uma conexão com o servidor, enviar dados, enviar uma requisição, encontrar URLs em um html, uma função que corrige os links para diretórios locais (para o *dump*) e um *parse* para URLs.

Documentação

- **mainwindow.h**

enum class ScreenName: enum com o nome das telas possíveis

construtor MainWindow: inicialmente coloca na tela do proxy e realiza todas as conexões que serão usadas no programa utilizando o mecanismo de signals e slots

proxy: instância da classe Proxy que será enviada para uma thread que roda doWork.

proxyThread: É um qthread...

webtools

webToolsThread

setScreen: método responsável por trocar de tela, inicializar a interface para essa tela e iniciar o thread correto.

signals

sendRequestAndReply: emitido ao mandar *request* ou *response* da interface para a *Thread*.

sendUrlAndDepth: emitido quando manda URL e profundidade(Depth) da interface do *spider* ou *dump* para a Thread.

slots

on_actionproxy_triggered(): Define que a tela atual é do Proxy.
on_actionspider_triggered(): Define que a tela atual é do Spider.
on_actiondump_triggered(): Define que a tela atual é do Dump.

on_pushButton_clicked(): Botão da tela do Proxy é clicado, logo manda para a interface os textos do request/reply e bloqueia o botão.

on_pushButton_2_clicked(): Botão da tela do Spider é clicado, logo limpa o campo de texto da interface, chama a execução do Spider e envia os campos de URL e depth da interface para o objeto Webtools para ser utilizado no Spider como parâmetros.

on_pushButton_3_clicked(): Botão da tela do Dump é clicado, logo chama a execução do Dump e envia os campos de URL e depth da interface para o objeto Webtools para ser utilizado no Dump como parâmetros.

on_requestReceived(): mostra o texto do request quando o Proxy o recebe, permitindo edição pelo usuário

on_requestSent(): bloqueia a área de texto do request na interface, não permitindo edição quando o Proxy envia o request.

on_replyReceived(): mostra reply recebido pelo Proxy na interface, sendo possível editá-lo e libera o botão da interface para liberar o reply para o cliente.

on_replyRetrieved(): quando o reply é enviado para o cliente, bloqueia áreas de texto e botão da interface do Proxy.

on_nodeSpider(): quando recebe uma url do Spider a mesma é mostrada na tela do Spider juntamente com as anteriores, montando a árvore hipertextual do Spider.

- **proxy.h**

construtor Proxy: inicializa variáveis stopFlag e buttonFlag com false ao instanciar um novo Proxy.

void doSetup(): inicia/reinicia a conexão entre o start da Thread e o Slot doWork() do Proxy.

void stop(): define a variável stopFlag = true para parar a execução da Thread.

Variáveis de controle:
bool stopFlag;
bool buttonFlag;

Variáveis para armazenar request e reply do Proxy:
string request = "";
QString reply = "";

Variável para obter parâmetros informados no comando de execução do código:
QStringList mainArgs = QApplication::arguments();

void waitButton(): função onde trava a execução do código, liberando após o clique do botão na interface.

int initServerSocket(string): inicializa o socket do servidor do Proxy.

signals

requestReceived(): emite sinal de que um request foi recebido para que a interface apresente a informação do request e permita sua edição.

requestSent(): emite sinal de que request foi enviado para que a interface bloqueie todos os campos esperando o recebimento de um reply.

replyReceived(): emite sinal de que um reply foi recebido pelo Proxy para que a interface apresente a informação do reply e permita sua edição antes do envio ao cliente.

replyRetrieved(): emite sinal de que um reply foi enviado para o cliente para que a interface se prepare para receber um novo request.

slots

doWork(): inicia a execução das ações do Proxy

on_buttonPressed(): informa que o botão da interface foi clicado e armazena os campos de request e reply da interface

- **webtools.h**

enum class Operation: enum com as operações possível: Spider ou Dump

construtor WebTools

doSetup(): inicia/reinicia a conexão entre o start da Thread e o Slot runSpider() ou runDump() dependendo da funcionalidade do Webtools escolhida.

stop(): define a variável stopFlag como true

Getters para URL e Depth de um objeto Webtools:
getUrl()
getDepth()

Variáveis de controle:
stopFlag;
buttonFlag;

depth: armazena depth informado pelo usuário na interface
url: armazena a URL informada pelo usuário na interface

waitButton(): função onde trava a execução do código, liberando após o clique do botão na interface.

spider(): chamada pela função runSpider(), para executar a ação do Spider em si.

dumpURL(): para cada URL retornada pelo Spider, irá criar o diretório para armazenar a página a ser baixada.

downloadPage(): download da página pelo Dump.

signals

nodeSpider(): emite sinal de que recebeu um nó do Spider para que o mesmo seja mostrado na interface para o usuário.

slots

runSpider(): Inicia execução da funcionalidade do Spider.
runDump(): Inicia execução da funcionalidade do Dump.

on_buttonPressed(): informa que botão da interface foi clicado e define as variáveis url e depth com os respectivos valores informados pelo usuário na interface.

- **httputils.h**

class Request: classe usada para fazer o parse e separar dados do request tais como método HTTP e host.

initServerSocket(): recebe o host, faz a resolução DNS, cria socket, conecta com o servidor e retorna o socket

sendData(string, int): envia todos os dados da string para algum socket

sendDataChar(char*, int, int); envia todos os dados de um array de caracter para um socket (para binários)

sendGet(int* serverSock,string host, string uri): manda uma requisição GET com connection close e retorna a resposta

fixUrl(string str, string host) : normaliza o formato da URL ou um path para sempre mostrar http://www…, usado no spider e consequentemente no dump.

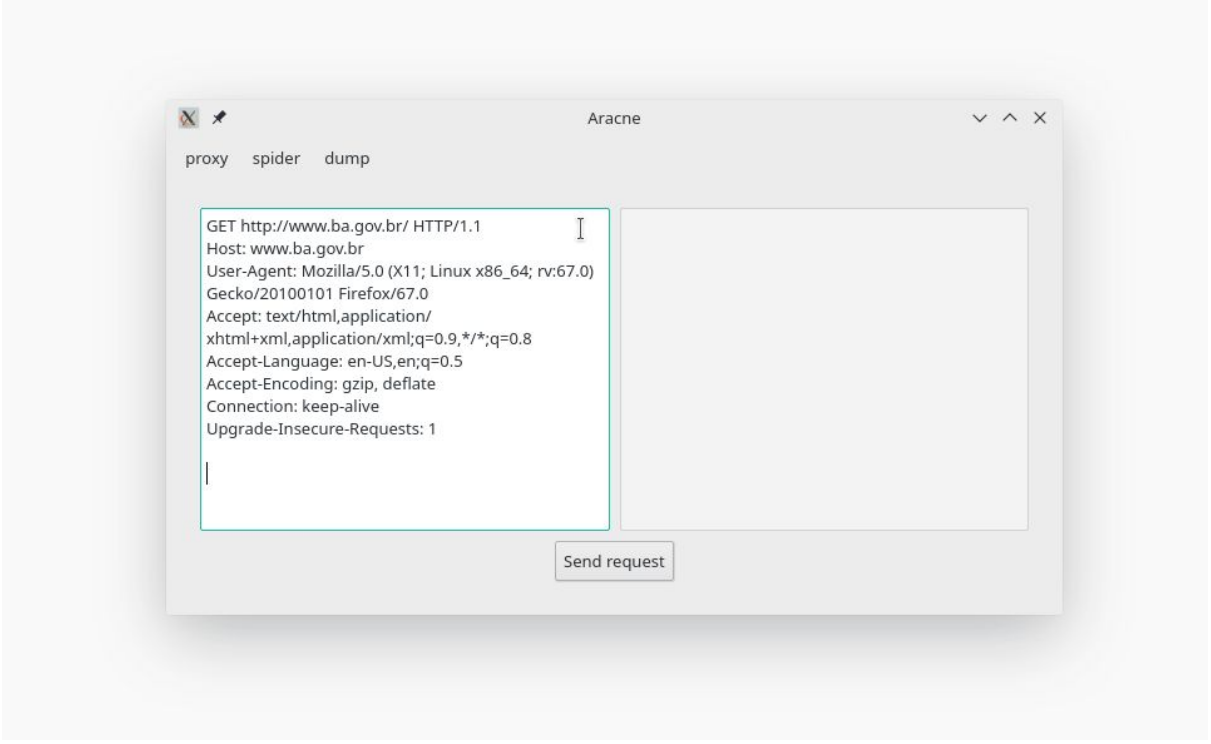
findPaths(std::vector<string>& paths, std::string response, string host): pega um html de resposta, procura todos os href e src e insere no vector paths

As 4 próximas funções são para extrair dados de uma URL:
extractHost(string): extrai o www.host.com sem o http e o path
extractPath(string): extrai tudo que tem depois do host
extractDirectory(string): extrai o diretório em que se encontra o arquivo
extractFileName(string): extrai apenas o nome do arquivo

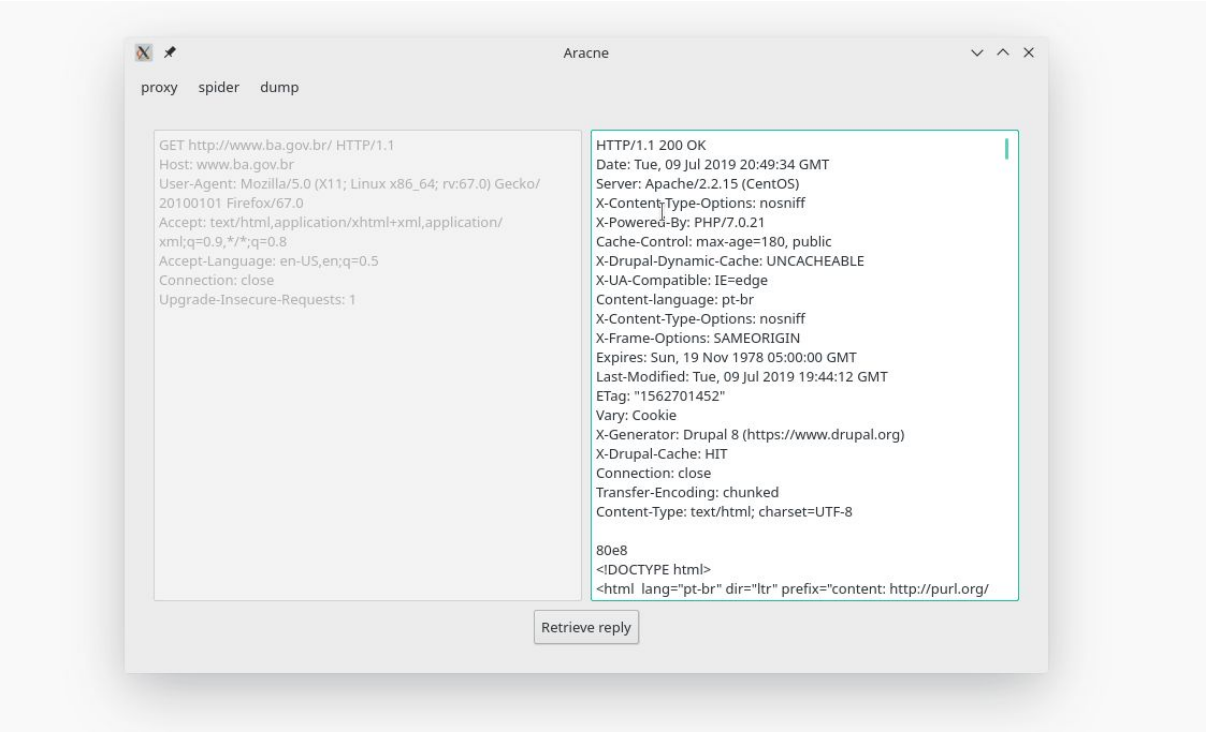
Screenshots do funcionamento

Proxy:

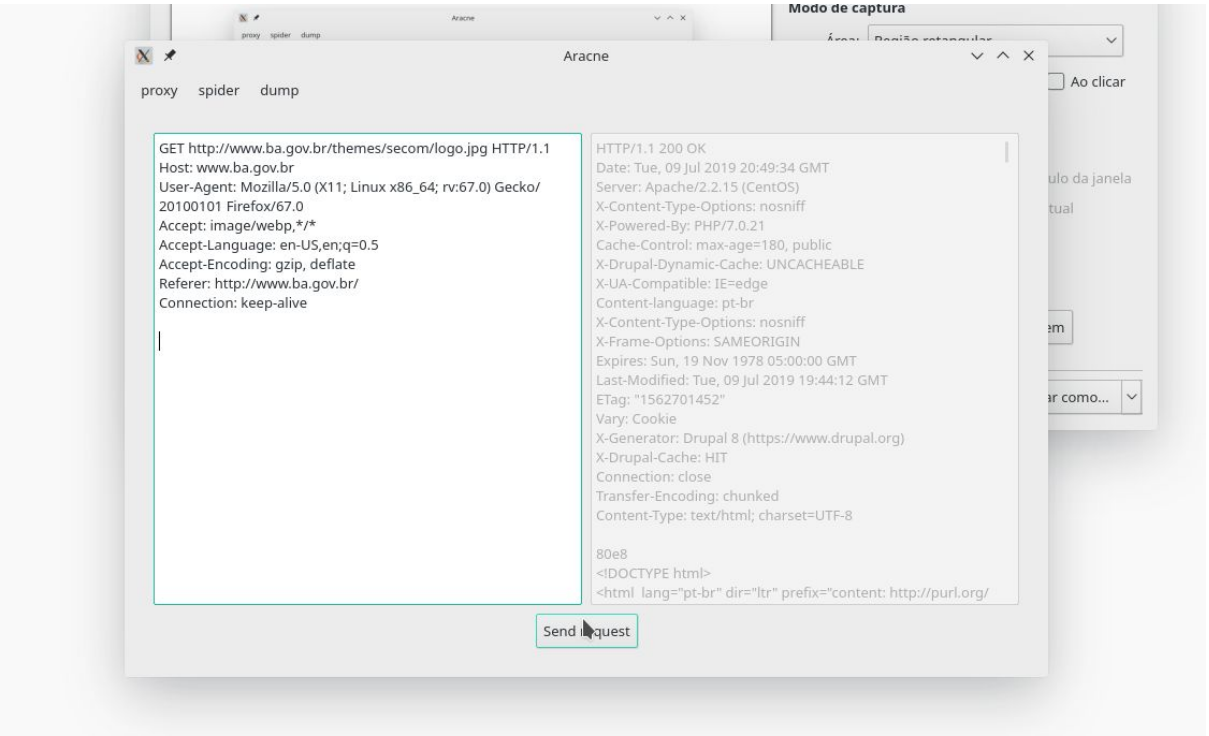
Com o browser configurado com proxy na porta 8228.
Ao entrar no site: http://www.ba.gov.br/



O primeiro request já é capturado pelo proxy e é possível editá-lo e enviá-lo ao servidor:

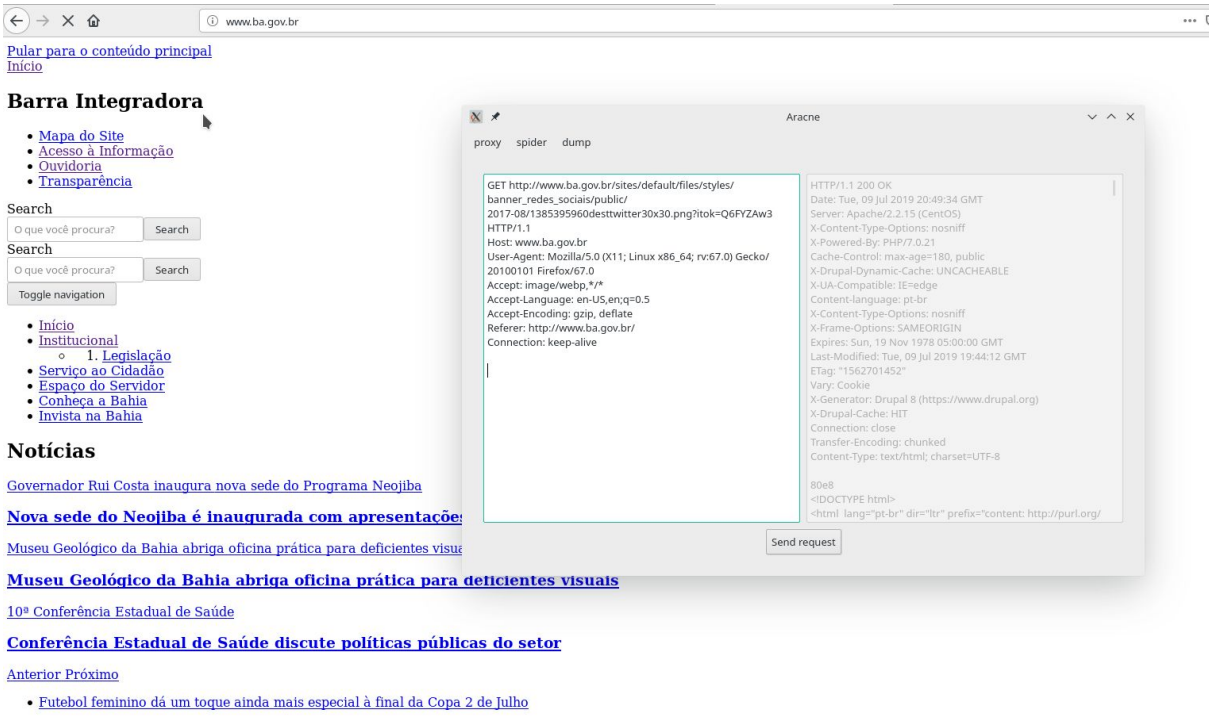


A resposta então é recebida e podemos editar e mandar de volta ao browser clicando em Retrieve reply



Após mandar o response para o browser, o proxy já captura o próximo request, e assim por diante.

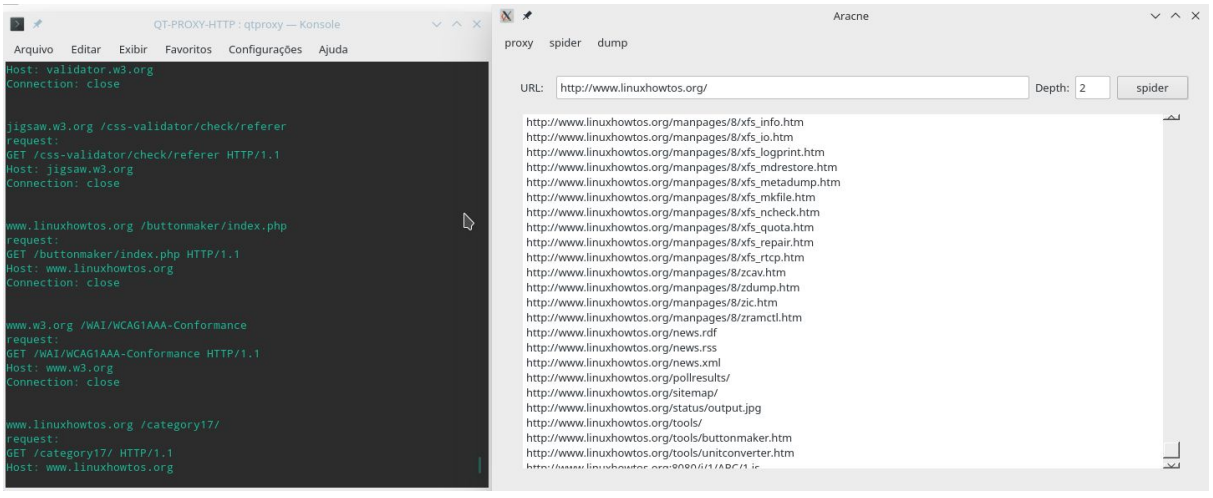
Após alguns requests, já é possível ver a página.



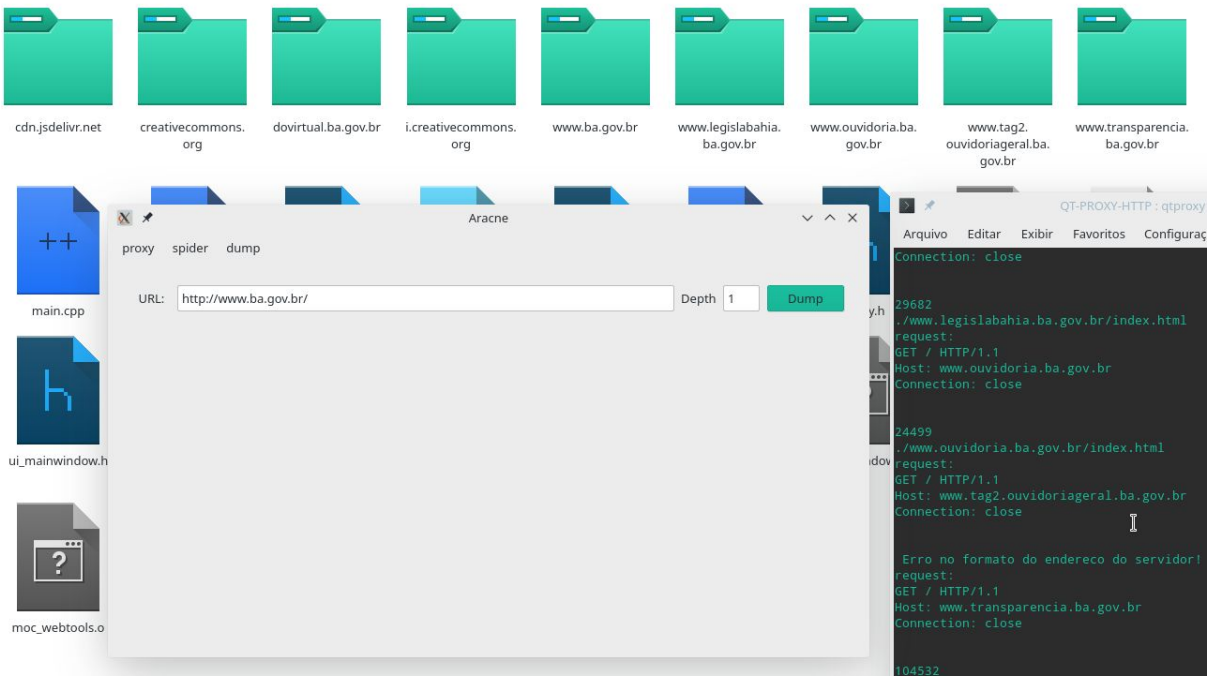
A funcionalidade de proxy implementada não trata imagens e outros binários, esse recurso foi apenas utilizado no dump.

Spider:

Ao inserir a url http://www.linuxhowtos.org/ com profundidade 2, ao clicar em spider, o terminal mostra todos os requests realizados e ao final, a interface mostra as URLs encontradas.

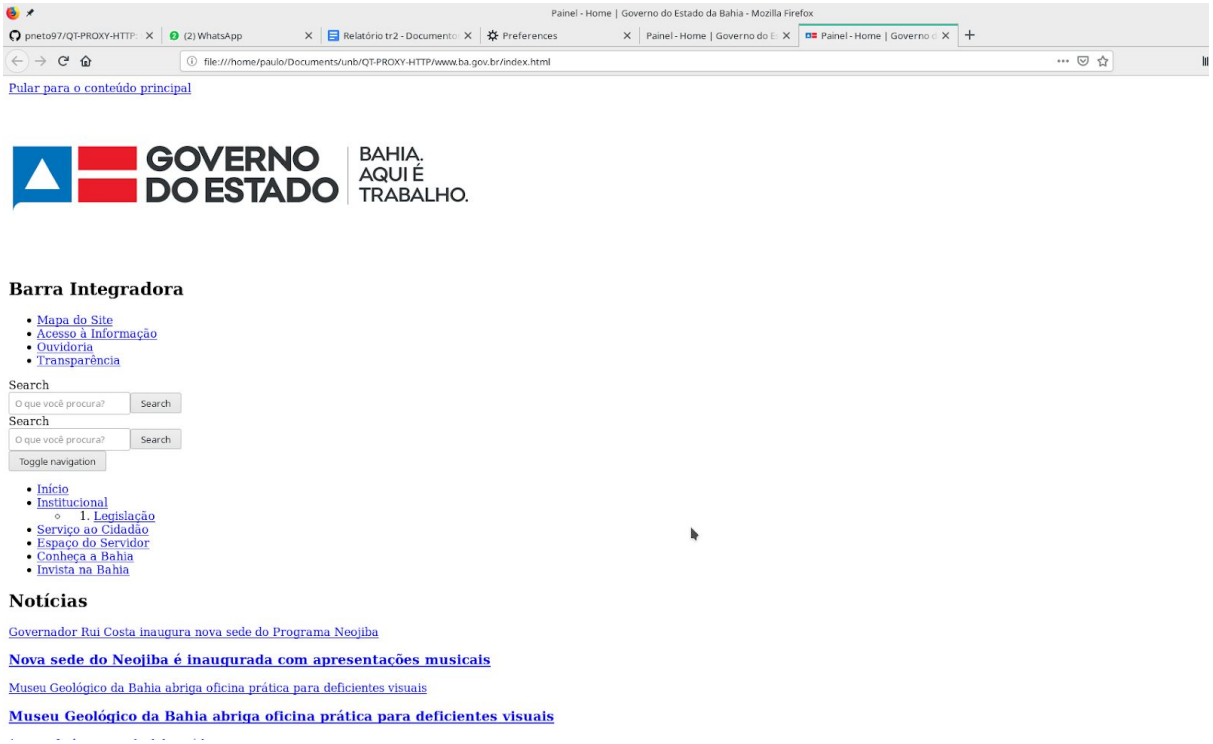


Dump:



Ao executar o DUMP, observe que o terminal mostra os requests realizados, mostra quando deu algum erro de resolução DNS (sem finalizar o programa) e para cada host encontrado, cria-se uma pasta no diretório em que se executa o programa.

Cada diretório de host armazena os arquivos do site.



Ao executar o index.html, mostra-se o que foi baixado do site. HTMLs, Imagens, javascript e css foram baixados. É possível navegar no site, apesar do css não ter sido carregado corretamente.

Bugs conhecidos

- No proxy, quando o firefox manda um request automático na inicialização, às vezes não recebe a resposta. É recomendado desativar o proxy para esses requests.
- Não tratamos imagens e binários no proxy, caso a requisição aceite gzip, remova essa linha para funcionar corretamente.
- No spider, algumas URLs quebram no meio e ocupam duas posições do array de urls aparecendo um estranho hexadecimal no meio, não acreditamos que o problema seja do parser que extrai a URL dos hrefs pois essas URLs variam em execuções diferentes.
- No dump, por causa do bug anterior, tenta baixar dados de uma URL quebrada e cria algumas pastas que não deviam, mas nada que dê crash no programa.
- No dump, apesar de funcionar na maioria dos sites e inclusive baixar imagens corretamente, em um site ocorreu *segmentation fault*, acreditamos que seja na hora de extrair dados binários do response para algum formato.

- No dump, em uma url o diretório de uma imagem em um site não foi extraído corretamente e ao invés de criar o arquivo .jpg ele atribuiu essa extensão no nome do diretório que ficou: www.pudim.com.brpudim.jpg