UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Elétrica e de Computação

THIAGO MAXIMO PAVÃO

EA872K - Projeto final: Servidor HTTP

Campinas

Segundo semestre de 2023

1 Introdução

Este projeto foi desenvolvido ao longo do semestre e consolidado em um servidor Web. O servidor projeta na rede uma pasta do computador, perimitindo que seus arquivos sejam acessados pela rede. O nome dessa pasta é *webspace*. O sistema funciona recebendo conexões via socket e respondendo requisições HTTP 1.1 atráves dele.

O cabeçalho da requisição é interpretado internamente com o uso dos analisadores *flex* (léxico) e *yacc/bison* (sintático.) Os analisadores geram uma lista ligada contendo cada campo do cabeçalho, o que facilita que a requisição seja lida e interpretada. Para então ser respondida.

O servidor também conta com multithreading, com cada conexão aberta sendo atendida por uma thread. Isto é ótimo, visto que grande parte das comunicações envolvem muita entrada/saída. Desta forma o servidor também é capaz de manter diversas conexões abertas simultâneamente, sem que outras conexões precisem aguardar para serem atendidas.

Por fim, o acesso aos arquivos do *webspace* pode ser controlado, através de arquivos *.htaccess* no diretório que se deseja proteger. Cada subdiretório também fica protegido e o servidor oferece um formulário de troca de senha de usuários.

2 Funcionalidades

Aqui segue uma lista de funcionalidades que foram implementadas no servidor, seguindo a especificação dada.

- Servidor executado com passagem de parâmetros via linha de comando. Os parâmetros são: Endereço do webspace, porta para abrir o servidor, arquivo de log, URL para troca de senha em diretórios protegidos, número máximo de threads que podem ser abertas pelo programa e o tipo de codificação dos arquivos de texto. Este último argumento é opcional.
- Conexões realizadas via socket, seguida pela leitura de uma ou mais requisições. Nos testes foram feitas conexões pelo comando *telnet* e pelo navegador.
- Leitura dos dados do socket feita para um buffer auxiliar na memória e posterior tokenziamento pelo *flex*. Assim a passagem dos dados do socket para os analisadores foi feita sem o uso de arquivos auxiliares.
- Análise do cabeçalho de requisições feita via *flex* e *yacc/bison*, com a geração de uma lista ligada contendo as informações necessárias.
- Processamento de requisições HTTP dos tipos GET, HEAD, OPTIONS, POST e TRACE. Métodos além destes são respondidos com o status 501 *Not Implemented*
- Envio de páginas de erro para informar melhor ao cliente o que ocorreu. Foram feitas páginas para os erros: 403 *Forbidden*, 404 *Not Found*, 405 *Method Not Allowed*, 500 *Internal Server Error* e 503 *Service Unavailable*. Também existem páginas de erro para cada possibilidade de problema que venha a ocorrer ao realizar a troca de senha.
- Verificação de confinamento do diretório informado, certificando-se de que não há tentativa de acesso à uma pasta externa ao *webspace* antes de buscar o recurso.
- Uso de threads de forma eficiente. Apenas foi necessária uma região crítica: Para o acesso à variável de contagem do número de threads em execução. Todas as variáveis globais utilizadas pelos analisadores foram removidas, isto foi feito transformando o sistema de *parsing* em reentrante.
- Proteção de acesso à diretórios contendo um arquivo *.htaccess*. Os arquivos contém um caminho que pode ser absoluto ou relativo para o arquivo de senhas. Cada senha é um par na forma *user:password*, onde password é criptografado pela chamada de sistema *crypt*.

- Possibilidade de troca de senha a partir de um formulário. Este formulário não está presente no *webspace*, fica junto com o servidor e é entregue por ele a depender do final da URL requisitada.

  Além destas, mais algumas funcionalidades merecem destaque.

- O *salt* da criptografia da senha foi feito da maneira mais genérica possível, o servidor é capaz de lidar com senhas de *salt* simples (2 caracteres) e da forma avançada: '$n$salt$'. Desta forma é possível criar um banco de senhas com criptografia mais avançada, aumentando a segurança.

- Suporte a diversos tipos de arquivo. O campo *Content-Length* deve ser configurado com o tipo correto de acordo com o arquivo, isto foi feito a partir da extensão do arquivo e é configurado corretamente para 21 tipos, com fácil expansão se for necessário.

- Proteção do arquivo *.htaccess*. Qualquer requisição à este recurso é respondida com o código 403 *Forbidden*, de forma à aumentar a segurança do servidor.

- Limpeza do servidor ao fechar. O servidor roda em *loop* atendendo requisições, mas é possível encerrar sua execução com o comando no terminal Ctrl + C. Ao detectar isto, o programa fecha o *socket* do servidor aberto, e fecha o arquivo de log.

3 Implementação

3.1 Organização dos arquivos

O programa é composto pela junção de diversos códigos em C. Além destes foram criados códigos com extensão .l e .y, que são os analisadores léxico e sintático, respectivamente.

Os arquivos desenvolvidos e suas descrições são as seguintes.

- *base64.c* e *base64.h*: Módulo para decodificação do par *user:password* recebido no cabeçalho em requisições que demandam Autenticação.
- *get.c* e *get.h*: Módulo de acesso ao sistema de arquivos, fazendo a busca de recursos e verificando cada condição necessária (permissão, autenticação, ...). Também contém a lógica de execução da troca de senha.
- *util.c* e *util.h*: Funções utilitárias, utilizadas de forma compartilhada pelos módulos.
- *lex.l*: Código para construção do analisador léxico.
- *sint.y*: Código para construção do analisador sintático.
- *server.c*: Programa principal do servidor, contendo a inicialização e o loop que atende cada nova conexão.

São utilizadas as ferramentas de geração de código *flex* e *bison* nos arquivos *lex.l* e *sint.y* para gerar arquivos *lex.yy.c* e *sint.tab.c*, que são efetivamente os analisadores léxico e sintático que são compilados no programa final. Os comandos de compilação foram configurados para gerar também um arquivo de cabeçalhos (extensão *.h*), para que as definições criadas por eles possam ser utilizadas em outros arquivos.

Também existem duas pastas importantes para o projeto. O diretório *serverPages*, presente no mesmo local que o programa, contém todas as páginas que são enviadas pelo servidor, e que não fazem parte do *webspace*. São elas:

- Páginas informativas sobre erros, como *error_403.html*, *error_404.html* e outras.
- Formulário de troca de senha. Esta página é enviada quando um cliente requisita a URL de troca de senha em um diretório protegido.
- Páginas informativas sobre erros na troca de senha.

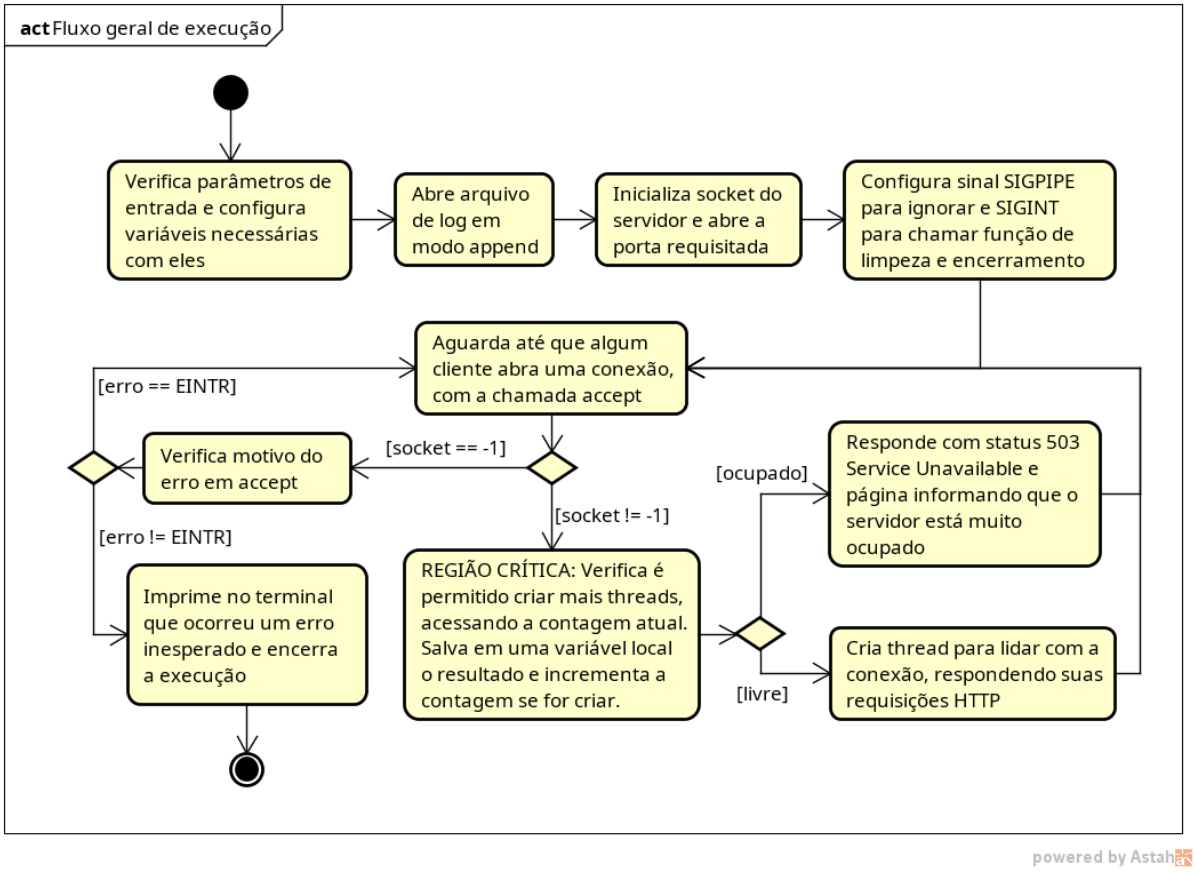

Figura 1 – Diagrama de fluxo de execução da theread principal

- Página de sucesso de troca de senha.

A outra pasta se chama *passwords* e também está presente no diretório do programa. Nela estão contidos os arquivos .htpassword especificados por caminho relativo no *.htaccess*.

3.2 Fluxo geral de execução

O fluxo principal está presente na função main, e conta com a ordem de operação presente na Figura 1. Este é o fluxo executado pela thread principal, que aguarda conexões e cria threads para atendê-las.

Vale destacar que o mínimo de tempo é gasto dentro da região crítica, o sistema determina se será permitido criar outra thread e já incrementa a variável se for criar, saindo da região crítica assim que é possível. Outro ponto é o sinal SIGPIPE, que foi ignorado. Isto foi necessário pois há situações em que o cliente fecha a conexão antes do servidor terminar de enviar a resposta. Isto faz com que o socket para o qual ele está escrevendo seja inválido, o que gera o sinal SIGPIPE.

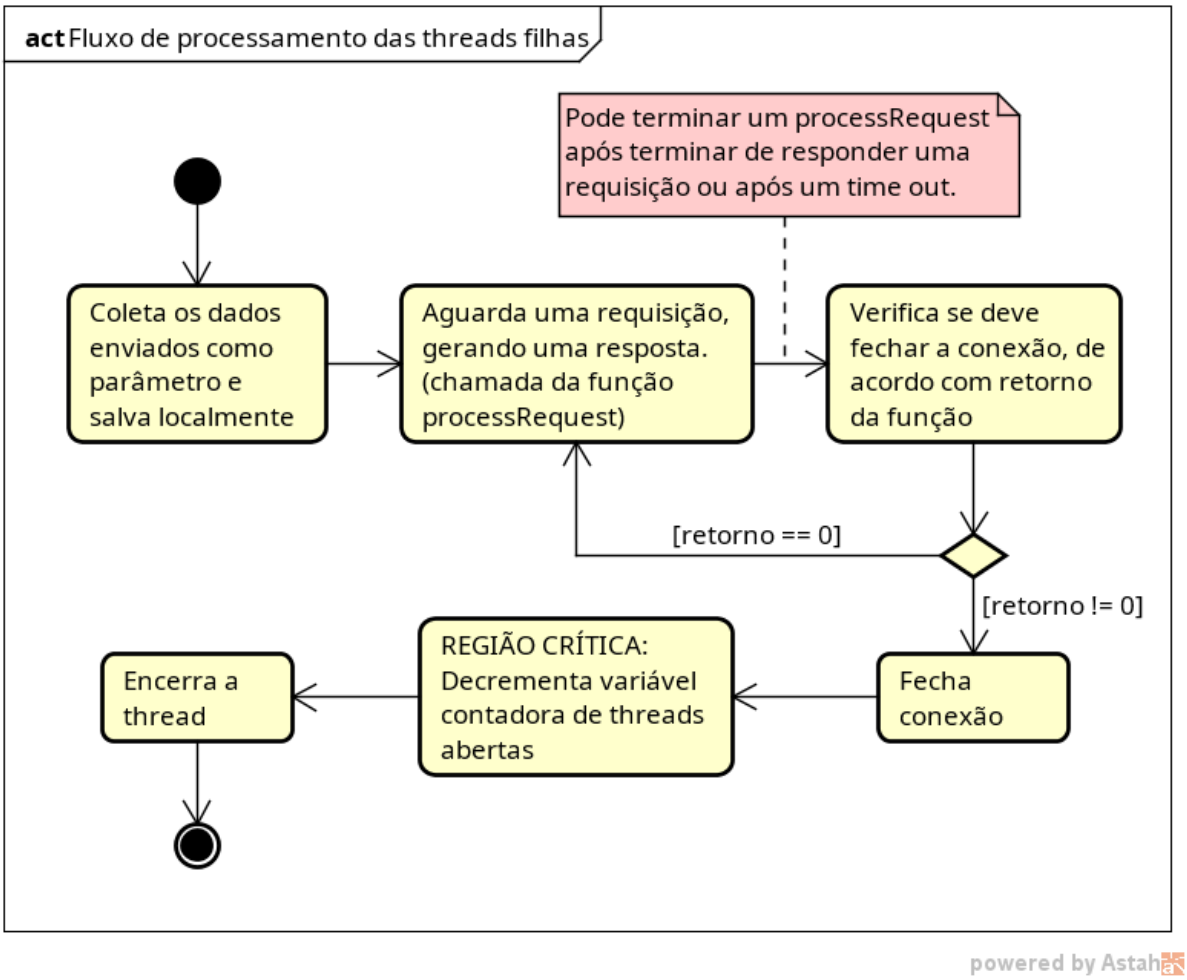

Figura 2 – Fluxo de operação das threads que lidam com as requisições

3.3 Fluxo de execução das threads filhas

Cada thread é criada para lidar com uma conexão recebida pela thread principal. Ela recebe como parâmetro o número do *socket* da conexão, o local do *webspace* e o *file descriptor* do arquivo de log. Então, ela é resposável por responder requisições vindas do socket até que seja a hora de fechá-lo. Isto é feito com auxílio da função *processRequest* que será detalhada posteriormente. Na figura 2 é possível ver os passos de operação das threads auxiliares.

3.4 Processamento de requisições

A função *processRequest* realiza a leitura do socket e responde à uma requisição, então ela retorna um valor igual a 0 quando deve-se manter a conexão aberta para atender novas requisições. Para isso basta chamar novamente a função. O retorno é diferente de 0 se a conexão deve ser fechada. Isto é determinado pelo campo do cabeçalho enviado pelo cliente: *Connection*. Este é o fluxo comum, mas há casos especiais como erros no *parsing* e

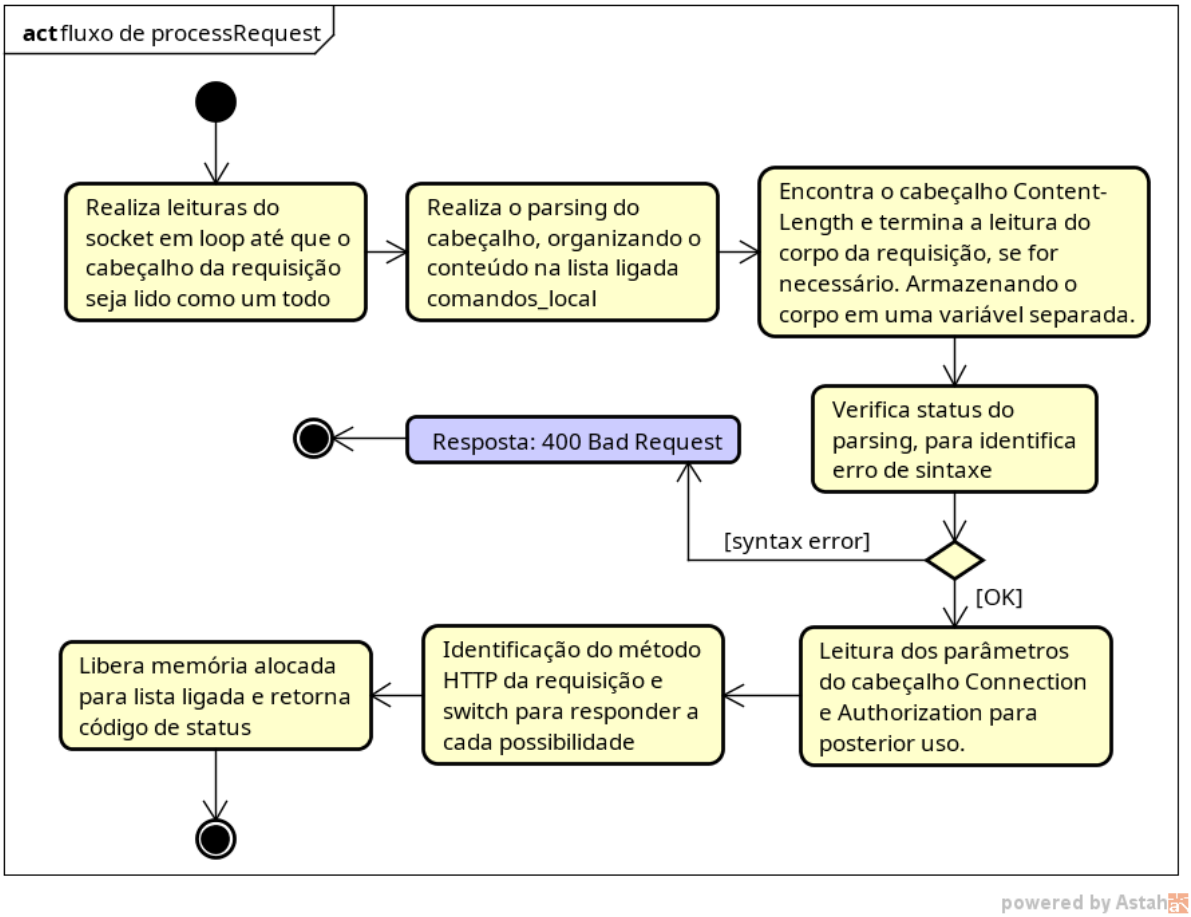

Figura 3 – Fluxo de execução da função *processRequest*.

outros. O diagrama da Figura 3 mostra qual a sequência de operações da função.

Cada um dos métodos HTTP identificados é tratado e respondido em casos da estrutura *switch* da linguagem C. O switch determina além dos métodos suportados, outros dois casos:

- INVALID: O caso em que o método HTTP presente na lista ligada não é valído, ou seja, um método que não faz parte do padrão HTTP 1.1. Na prática este caso não deve ser executado, já que um erro deste tipo é reconhecido pelo analisador sintático (primeiro token na gramática precisa ser do tipo método).
- default: O caso padrão é executado quando o método HTTP é válido, mas não é suportado pelo servidor, por ser diferente de GET, HEAD, OPTIONS, TRACE e POST. Então a resposta neste caso é o status 501 Not Implemented.

Também vale notar que durante as leituras do socket para montar o cabeçalho ou terminar de ler o corpo da requisição é utilizada a chamada de sistema *poll* para aguardar que haja conteúdo no socket para ser lido. Este poll é feito com 5 segundos de *timeout* para que o servidor não aguarde eternamente até que o cliente decida enviar algo.

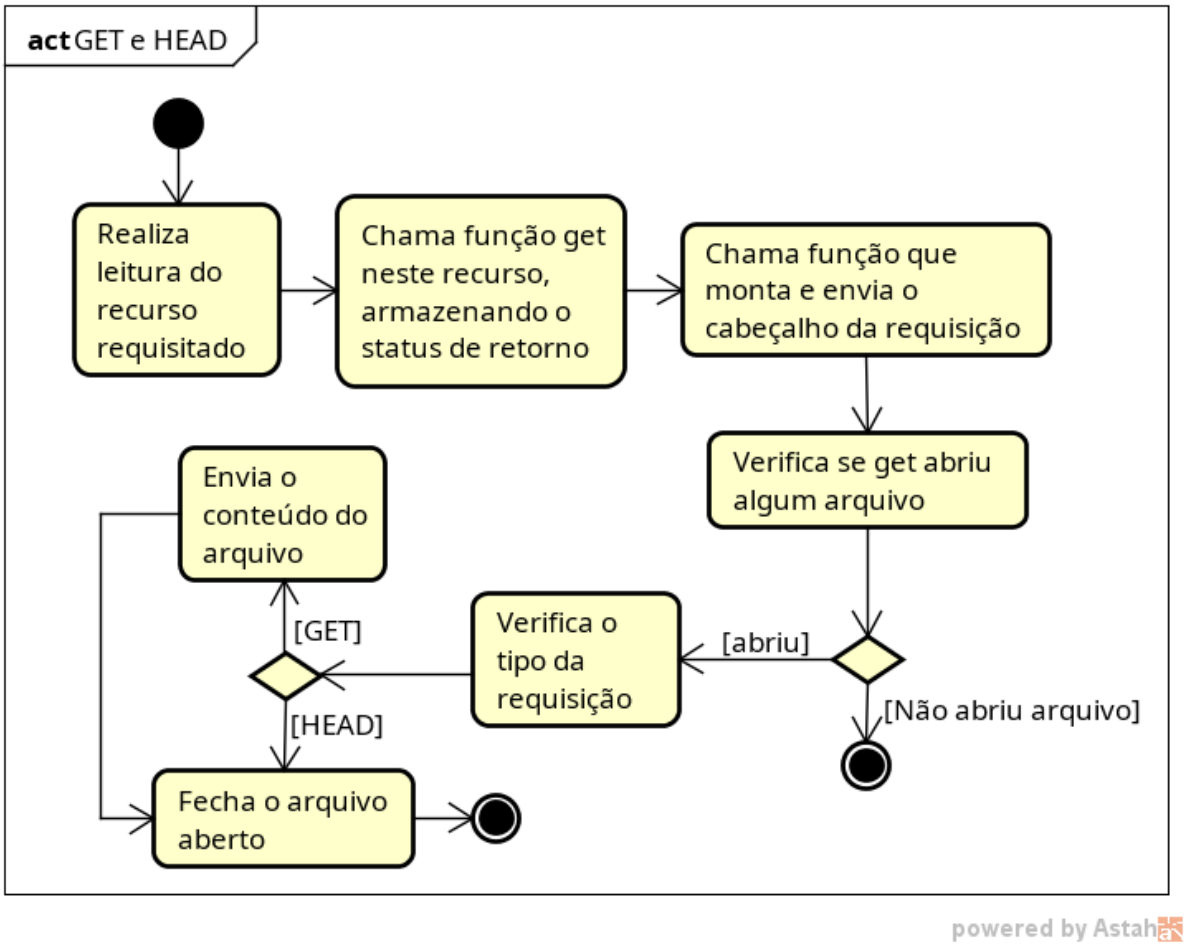

Figura 4 – Fluxo de execução no caso de requisição do tipo GET e HEAD

Quando é identificado que o retorno de poll foi nulo, a função retorna um valor indicando para a thread que a conexão deve ser fechada.

3.4.1 GET e HEAD

Estes dois métodos são respondidos de forma muito parecida, pois GET tem como resposta um recurso requisitado, enviando um cabeçalho da requisição e o arquivo no conteúdo. Por outro lado, o HEAD deve enviar o mesmo cabeçalho que GET enviaria, mas sem enviar o conteúdo do arquivo. O fluxo de execução neste caso é mostrado na figura 4.,

A função get é responsável por buscar o recurso no sistema de arquivos, identificando qualquer possível erro no caminho. Ela abre o recurso requisitado, ou uma página de erro, e retorna o status da resposta. O diagrama mostrando seu fluxo de execução está presente na Figura 5.

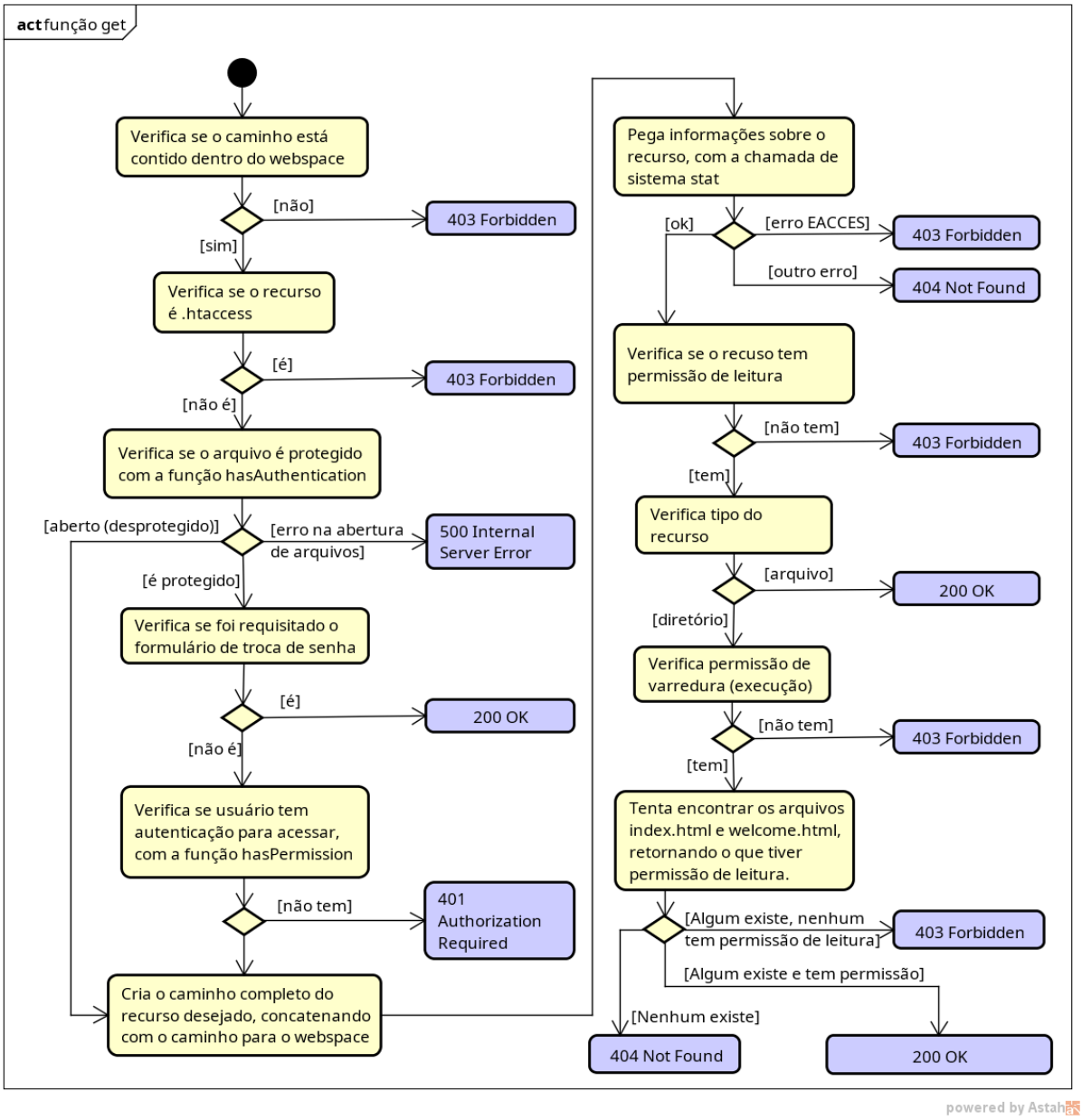

Figura 5 – Fluxo da função get.

3.4.2 OPTIONS

Este método é respondido com uma primeira linha contendo o cabeçalho *Allow*, seguido dos parâmetros GET, HEAD, OPTIONS, TRACE.

No caso de requisições na URL de troca de senha, o cabeçalho também contém o método POST, já que este é o método para realizar efetivamente a troca de senha.

Além deste primeiro cabeçalho, são enviados outros que são padrão em qualquer resposta como *Connection*, *Date*, *Server* e outros.

3.4.3 TRACE

Este método serve para realizar um teste de conexão, sua resposta é um status 200 OK juntamente com os mesmos cabeçalhos que foram enviados na requisição.

3.4.4 POST

O processamento geral deste caso é muito parecido com o do GET/HEAD, porém ao invés de buscar o recurso é chamada uma função que realiza todo o trabalho de verficar a validade da requisição até efetivamente trocar a senha. A função retorna o status da resposta e abre um arquivo informando sobre o que aconteceu, como uma página avisando que as senhas não coincidem, usuário ou senha não reconhecido, a página informando o sucesso na troca de senha e outras.

Recebendo isto, basta gerar o cabeçalho e enviar o arquivo, assim como era feito para o GET. O processo da função processPost está presente no diagrama da Figura 6.

3.4.5 Função Cabeçalho

Uma função utilizada na resposta de todos os métodos HTTP suportados é a função de geração e envio do cabeçalho. Ela é útil pois muitas das respostas tem campos em comum, portanto seu código é reutilizável e a chamada pode ser personalizada para gerar o cabeçalho de qualquer resposta.

Ela recebe como parâmetro um inteiro indicando o status da resposta (Ex: 200, 404, etc), o descritor do arquivo que está sendo enviado (opcional), o número do *socket* e do arquivo de log. Com isto e mais alguns parâmetros não tão importantes ela realiza os seguintes passos, enviando o cabeçalho da resposta tanto para o *socket* da conexão quando para o arquivo de log.

- Status da resposta, juntamente com um texto correspondente. Ex: 404 Not Found.
- Data e hora atual.

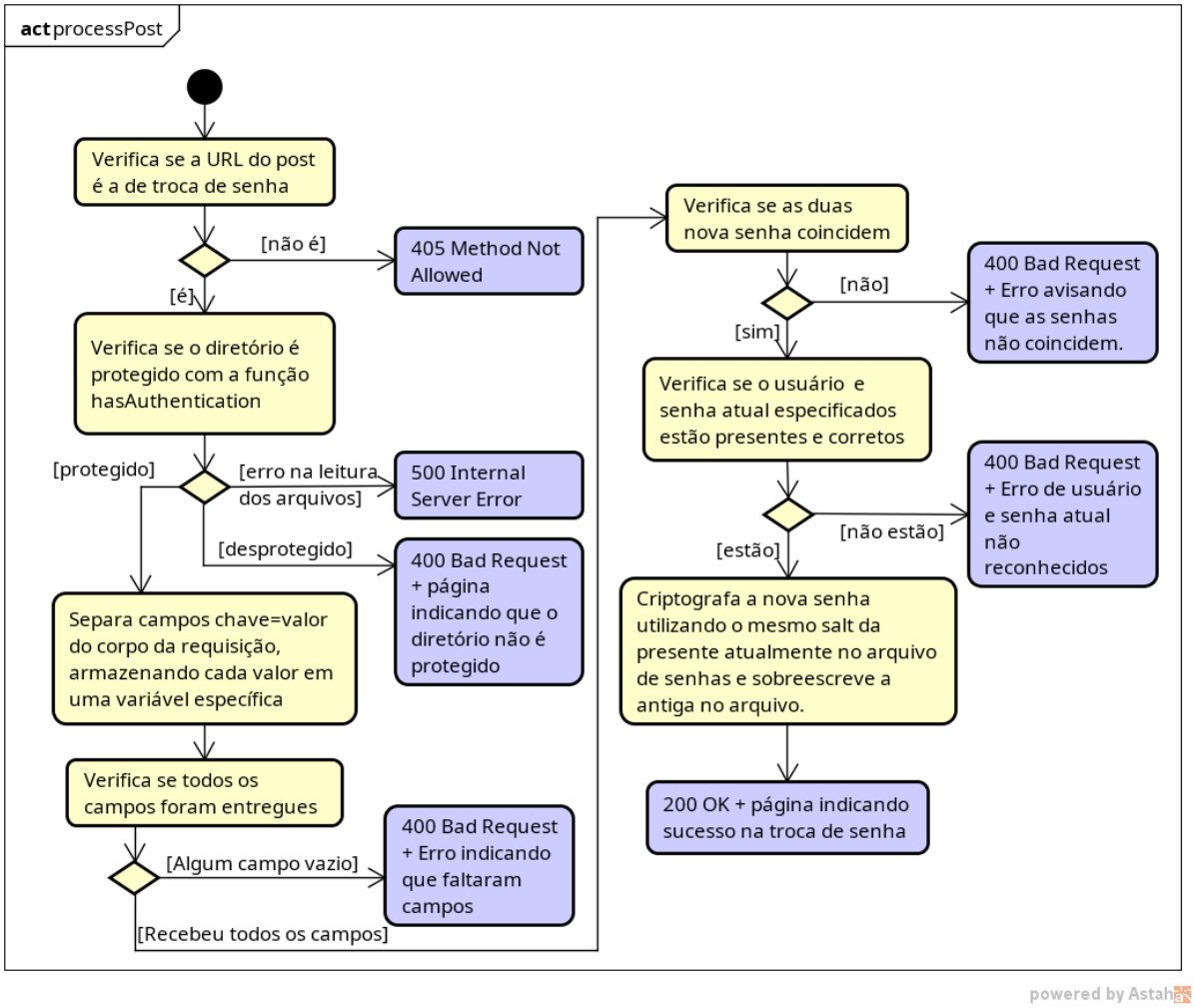

Figura 6 – Fluxo da função que processa o POST.

- Nome do servidor.
- Caso status = 401, Cabeçalho *WWW-Authenticate* para browser solicitar autenticação do tipo básico.
- Campo indicando status da conexão *keep-alive* ou *close*, igual ao que tiver sido enviado pelo cliente em sua requisição. Se não tiver sido enviado, utiliza *close* como padrão.
- Se tiver sido enviado um arquivo também:
  - Data e hora de última modificação.
  - Tamanho em bytes do arquivo.
  - Tipo do arquivo, juntamente com o tipo de codificação especificado como argumento na chamada do programa para aquivos do tipo text.
- Caso contrário, envia 0 como *Content-Length* e tipo *text/html*.

3.5 Analisadores Léxico e Sintático

A verificação do formato do cabeçalho da requisição recebida e organização é feita pelo conjunto do *flex* e do *yacc*, que são os analisadores léxico e sintático, respectivamente. Por padrão, eles utilizam de variáveis globais para realizar a comunicação entre eles. Ao desenvolver a primeira versão do servidor com *threads* foi necessário envolver todo o processo do *parsing* em uma região crítica.

Isto não é ideal, porque o processo de computação para realizar o *parsing* é muito grande. Ele é necessário sempre que o sistema precisa ler uma requisição, portanto a probabilidade de muitas threads ficarem esperando sua vez de utilizar o mecanismo é grande.

Para resolver este problema, os analisadores foram adaptados para funcionarem de forma reentrante. Este nome vem do fato de que mesmo que o sistema já esteja realizando o *parsing* em algum *thread* o mecanismo pode ser chamado novamente por outra *thread*.

A essencia das modificações necessárias é fazer com que as variáveis que anteriormente eram globais ou até mesmo omitidas fiquem locais e sejam passadas entre o programa e os analisadores como parâmetro, de forma que não sejam usadas variáveis compartilhadas entre as *threads* no processo.

Para a função que chama o *parser*, a modificação necessária foi criar um *scanner* do flex, uma estrutura que guarda informações importantes para que o analisador léxico consiga ler e retornar os tokens. Este scanner é enviado como parâmetro para *yy_scan_string*, função utilizada para enviar o buffer que se deseja tokenizar ao analisador, e para *yy_parse*, para que o parser utilize-o para obter os tokens.

O programa também foi modificado para que o ponteiro da lista ligada também fosse enviado por parâmetro, então quando ela é construída seu valor também é enviado para uma variável local da thread, sem o uso de variáveis globais da primeira versão.

3.5.1 Tokens

Os tokens reconhecidos pelo analisador léxico são os seguintes, juntamente com sua expressão regular e explicação.

- Nome do token | Expressão regular | Explicação
- WORD | `[^,\t\n\r]*` em modo <parametros> ou `[^ \t\n\r:,]+` em modo normal | Sequência de caracteres qualquer, podendo ou não incluir espaços, à depender do modo de execução.
- FIELD_SEPARATOR | `:` | Separador de nome do cabeçalho e seu valor.

- OPTION_SEPARATOR | `,` | Separador de valores do cabeçalho.
- METODO | `GET|HEAD|OPTIONS|TRACE|POST|PUT|DELETE` | Métodos válidos do HTTP.
- NEWLINE | `[\r][\n]` | Quebra de linha em requisições da forma CRLF.

O modo de reconhecimento do token WORD é alterado porque na primeira linha da requisição e antes do separador de nome do cabeçalho e valor uma palavra é uma sequência de caracteres que não inclui espaços em branco. Já no reconhecimento de 'opções' de um comando cada opção é separada por vírgulas, então pode conter espaços em seu interior.

Em modo de reconhecimento normal, a sequência "`GET /recurso HTTP/1.1`" é tokenizada como METODO, WORD, WORD.

Já no reconhecimento de parâmetros de um comando, por exemplo, "`User-Agent: Mozilla/5.0 (X11; Linux x86\_64; rv:102.0)`" os tokens WORD FIELD_SEPARATOR WORD seriam gerados, o que é correto mesmo que existam espaços no meio.

Isto é possível pois ao reconhecer o separador de campos o modo é alterado, e este modo aceita que espaços participem de um token do tipo WORD. Neste modo, o separador é a vírgula, então ela não é permitida dentro de um token do tipo WORD e a sequência "`Accept: image/avif,image/webp`" geraria os tokens WORD FIELD_SEPARATOR WORD OPTION_SEPARATOR WORD.

Outro benefício desta estratégia é permitir que parâmetros tenham caracteres que seriam reconhecidos como outros tokens, se não houvesse a mudança no modo, a string "Host: localhost:2339" seria tokenizada como WORD FIELD_SEPARATOR WORD FIELD_SEPARATOR WORD. O que causaria um erro de sintaxe no analisador sintático. Com a mudança de modo, o caractere ':' pode ser reconhecido como WORD, então a tokenização gerada é WORD FIELD_SEPARATOR WORD, que é válida.

3.5.2 Gramática

A gramática definida com a ferramenta *Yacc/Bison* é composta pelos símbolos terminais

- METODO
- WORD
- FIELD_SEPARATOR
- OPTION_SEPARATOR
- NEWLINE

Além disso, os símbolos não terminais criados são

- linha
- linhas
- opcoes
- opcoes-metodo

Onde *linhas* é o símbolo inicial da gramática. E as regras de transformação são as mostradas a seguir.

- linhas → linha linhas
- linhas → linha

- linha → WORD FIELD_SEPARATOR opcoes NEWLINE
- linha → WORD FIELD_SEPARATOR NEWLINE
- linha → METODO opcoes-metodo NEWLINE
- linha → NEWLINE

- opcoes → WORD OPTION_SEPARATOR opcoes
- opcoes → WORD

- opcoes-metodo → WORD opcoes-metodo
- opcoes-metodo → WORD

Cada uma dessas regras também é combinada com um bloco de código, que é responsável por gerar e incrementar as listas ligadas de acordo com o que foi reconhecido. Os valores dos tokens são passados pelas transformações utilizando $$, e lidos com $1, $2, etc. Que são indicativos para o *yacc* do valor que deve ser atribuído a cada símbolo.

Ao fim de um parsing bem sucedido é construída uma lista ligada que pode ser utilizada para acessar e procurar cabeçalhos e valores da requisição. Se ocorrer um erro de sintaxe, o valor do retorno de *yyparse* é negativo, o que é tratado no código de processamento de requisições.

3.6 Sistema de Autenticação e Troca de Senha

A organização do sistema de autenticação é dada da seguinte forma. Um diretório contendo um arquivo *.htaccess* é protegido, assim como todos os subdiretórios abaixo dele. No caso de um *.htaccess* dentro de um diretório já protegido por outro (mais acima na árvore) o que ocorre é que este mais para dentro protege a subárvore de diretórios que ele contém.

Ao buscar um recurso, cada diretório é visitado à procura de um arquivo *.htaccess*, partindo do que contém o recurso até a raíz. O primeiro que for encontrado é o que protege aquele recurso, visto que ele é o mais próximo do recurso. Se a árvore for percorrida até a raíz do webspace sem encontrar o arquivo, então o recurso é acessado e entregue sem verificação de autenticação.

Cada arquivo *.htaccess* deve conter um endereço para outro arquivo, que é o arquivo contendo os usuários e senhas que tem autorização para acessar aquele diretório. O caminho para o arquivo pode ser dado de forma absoluta (caminho começa com '/') ou relativo, neste segundo caso, o programa procura o arquivo considerando como raíz o diretório 'passwords' no mesmo local que o arquivo.

O conteúdo do arquivo de senhas deve ser pares de usuário e senha na forma *user:password*, um por linha do arquivo. *user* é o nome de usuário que deve ser utilizado para fazer login e *password* é a saída da chamada de sistema *crypt* para uma senha e um salt.

Vale notar que o sistema foi implementado para suportar criptografia básica, onde o salt é apenas dois caracteres, e também criptografia avançada, podendo ser dos tipos MD5, SHA-256 e SHA-512. A criptografia escolhida altera a forma que a senha é gerada, pois o salt tem outro formato.

Também destaca-se que a forma de terminação das linhas nos arquivos (*.htaccess* e arquivo de senhas) pode ser do tipo LF ou CRLF, pois o programa se adapta ao tipo escolhido. No entanto é importante que o arquivo *.htaccess* contenha apenas a linha com o nome do arquivo. Esta linha pode terminar com uma quebra de linha ou não mas não é permitido que existam outros caracteres na linha seguinte.

As funcionalidades de acesso à recursos protegidos e de troca de senha são feitas utilizando as funções *hasAuthentication* e *hasPermission*, como é possível ver no fluxo de execução da função *get* (Figura 5) e de *processPost* (Figura 6). O funcionamento interno dessas funções são mostrados a seguir.

3.6.1 Função hasAuthentication

Esta função é responsável por identificar se um recurso é protegido por algum arquivo *.htaccess*. Se ele for, ela também abre o arquivo de senhas daquele diretório e retorna para que possa ser lido ou escrito posteriormente.

3.6.2 Função hasPermission

Esta função utiliza o arquivo de senhas aberto e o usuário e senha informados. Ela procura o usuário pelo arquivo e se encontrar compara a senha armazenada nele com a senha gerada criptografando-se a senha enviada pela requisição com o mesmo salt da presente no arquivo.

Se algum passo desta função falhar, a função retorna que o usuário não tem permissão de acesso. Apenas retornando que o acesso é permitido se todos os passos forem executados com sucesso.

Além disso, a função também pode estar sendo chamada para verificar se o dados de um usuário querendo realizar a troca de senha estão corretos. Para facilitar o resto do processo de troca de senha, a função envia em parâmetros de saída a string do *salt* da senha no arquivo e a quantidade de bytes no arquivo até a linha que o usuário se encontra. Desta forma é fácil gerar a nova senha e posicionar o cursor para sobrescrevê-la.

Estes dados só são enviados caso haja sucesso na verificação de permissão. Pois os valores não serão utilizados caso o usuário e/ou senha atual estiverem errados.

3.6.3 base64

Ao requisitar autenticação para algum recurso, o servidor envia o cabeçalho "WWW-Authenticate: Basic ...". Isto faz com que o par *user:password* seja enviado codificado em base64, que é um tipo de codificação reversível utilizado para transformar qualquer sequência de bytes em caracteres imprimíveis.

Desta forma garante-se que os dados de login não são interpretados de alguma forma na transmissão e que eles chegam da mesma forma que foram enviados pelo cliente.

Com isso, também faz-se necessário descodificar o conteúdo de login antes de chamar a função *hasPermission*, isto é feito pela função *base64decode*, que foi obtida de <https://stackoverflow.com/a/6782480>.

4 Testes

4.1 Conteúdo do Webspace de Teste

O webspace de testes foi construído adaptando-se o que foi entregue na página da disciplina. O resultado foi o webspace descrito a seguir.

Estão presentes na raíz do webspace:

- *a.txt*: Arquivo contendo a frase "Este é um arquivo texto."
- *configura_permissoes.sh*: Script de ajuste das permissões, dos arquivos e diretórios. Deve ser executado após a extração do webspace.
- *desfaz_configura_permissoes.sh*: Script que desfaz as alterações feitas pelo script acima. Devolvendo as permissões.
- *feec.gif*: Imagem com o logo da FEEC.
- *index.html*: Página principal, contendo hyperlinks para todos os arquivos/diretórios para testes.
- *lab.zip*: Pasta compactada contendo uma página index.html
- *teste.pdf*: Arquivo de teste de envio de pdf.
- *teste.txt*: Arquivo sem permissão de leitura contendo a frase "Este é o arquivo teste.txt que deveria estar sem permissão de leitura e, portanto, inacessível."
- *unicamp.gif/unicamp.jpg/unicamp.png/unicamp.tif*: Logo da Unicamp presente em diversos formatos diferentes de imagem.
- *welcome.html*:Página contendo apenas o parágrafo "This is welcome.html"

Também existem diretórios, que serão detalhados a seguir. O nome de cada subseção é o nome da pasta.

4.1.1 dir

Diretório sem permissão de execução contendo um arquivo: *texto_de_teste.txt* que contém o texto "Este é o arquivo texto1.txt."

4.1.2 dir1

Contém um arquivo *index.html* e um diretório dir11. Este por sua vez contém outra página index.html, e um arquivo .htaccess, protegendo o diretório.

Também há dentro de dir11 a pasta dir111. Esta também tem uma página index.html e um .htaccess, que referencia outro arquivo de senhas e portanto exige outra autenticação.

Cada página html descreve o que ela é e onde está localizada.

4.1.3 dir2

Contém apenas a página *welcome.html*, que apenas informa seu propósito.

4.1.4 dir3

Contém páginas *index.html* e *welcome.html*, porém a primeira delas não tem permissão de leitura.

4.1.5 dir3.1

Mesmo do caso anterior porém não possui o arquivo *welcome.html*, apenas *index.html* sem permissão de leitura.

4.1.6 dir4

Diretório com páginas *index.html* e *welcome.html*, ambos sem permissão de leitura.

Também contém o subdiretório dir41, protegido internamente por um .htaccess e outro subdiretório em dir41: dir411. Este último não contém um .htaccess mas também é protegido pois o diretório acima conta com um.

dir41 e dir41/dir411 também contam com uma página *index.html* cada.

4.1.7 dir5

Contém apenas o arquivo *texto_para_testes.txt*, que por sua vez é formado apenas pela frase “Este é um outro texto para testes.”

Não contém *index.html* nem *welcome.html*

4.1.8 dir6

Contém página index.html de apresentação e um arquivo *.htaccess*, mas ele aponta para um arquivo de senhas inexistente.

4.1.9 Senhas

Os arquivos de senhas (*.htpassword...*) não são mantidos dentro do *webspace* por questões de segurança. Da forma que o servidor/*webpsace* foi configurado os arquivos das senhas estão presentes na mesma pasta que o servidor, em 'passwords'.

Foram criados três arquivos, para satisfazer cada *.htaccess* comentado acima. Em cada *.htpassword* foram configurados três usários, utilizando tipos de salt diferentes e tipos de criptografia diferentes, para certificar-se do funcionamento generalizado.

.htpassword_dir11

```
dir11user1:EAwPY2TFEeU6g
dir11user2:$5$AGRWNOZP$wnZAmkJzmHJGeAGdTueP6HqMfvvg/Nc9pwNVCrnDfz8
dir11user3:$6$ADitmSLw$iTUAM3XqCMr<cortado>Lm1GnAqjK9SnCX9EPOlpMFRlzIy.
```

Os usuários e senha são

- dir11user1 : dir11user1Password
- dir11user2 : dir11user2Password
- dir11user3 : dir11user3Password

.htpassword_dir111

```
dir111user1:arbp/B0G3pHUk
dir111user2:$5$lPW7RJPe$Tj4PsFgLKRt3B5w7TRoO5lHD7g8poxvhjjdWk0bDKG.
dir111user3:$6$BzxyHbMr$C4o5<cortado>lyJrrTODG2ZvBY9pQoxfe0asoUnYoL/j.
```

Os usuários e senha são

- dir111user1 : dir111user1Password
- dir111user2 : dir111user2Password
- dir111user3 : dir111user3Password

.htpassword_dir41

```
dir41user1:BhOccvQjI13zg
dir41user2:$5$9Vug8fyh$CCsWfY3CYu.gmOVlHsbM3zH43H36nYqcN5XWbRriRJ2
dir41user3:$6$qz1Y0X06$6A9x<cortado>p/63nC/k04uiIOTjL5sgQunKYOBf8Y1
```

Os usuários e senha são

- dir41user1 : dir41user1Password
- dir41user2 : dir41user2Password
- dir41user3 : dir41user3Password

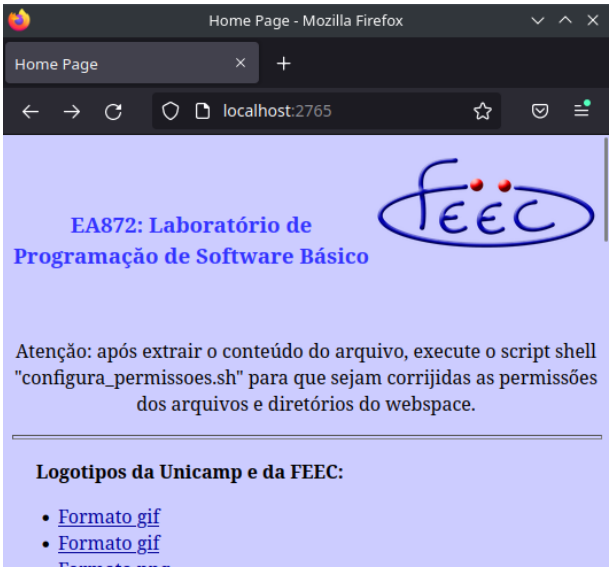

Figura 7 – Acessando recurso '/' retorna a página index.html na raíz do projeto

4.2 Execução e Resultado dos Testes

A maior parte dos testes foi executada pelo navegador. Utilizando as ferramentas de desenvolvedor foi possível extrair os cabeçalhos da requisição e da resposta, quando necessário. Em alguns casos também foi necessário utilizar o comando *telnet* para se conectar e enviar requisições de forma manual.

4.2.1 Teste 1 - Acesso ao arquivo index.html ao acessar a URL '/'

Acessando a URL `localhost:2765` após abrir o servidor nesta porta, vemos a página index. Na Figura 7

Requisição realizada:

```
GET / HTTP/1.1
Host: localhost:2765
User-Agent: Mozilla/5.0<cortado>Gecko/20100101 Firefox/102.0
Accept: text/html,<cortado>,image/webp,*/*;q=0.8
Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: cross-site
```

Figura 8 – Tentativa de acesso à um arquivo sem permissão de leitura

Cabeçalho da resposta:

```
HTTP/1.1 200 OK
Date: Wed Dec 06 07:44:06 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: keep-alive
Last-Modified: Tue Dec 05 21:31:16 2023 BRT
Content-Length: 3600
Content-Type: text/html; charset=utf-8
```

4.2.2 Teste 2 - Arquivo sem permissão de leitura

Acessando o arquivo sem permissão de leitura na raíz do webspace, *teste.txt*. Vemos corretamente a página de erro 403, como visto na Figura 8.

Requisição realizada:

```
GET /teste.txt HTTP/1.1
Host: localhost:2765
User-Agent: Mozilla/5.0<cortado>Gecko/20100101 Firefox/102.0
<cortado>
Accept: text/html,application/xhtml+xml,application
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
```

Cabeçalho da resposta:

```
HTTP/1.1 403 Forbidden
Date: Wed Dec 06 07:49:26 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: keep-alive
```

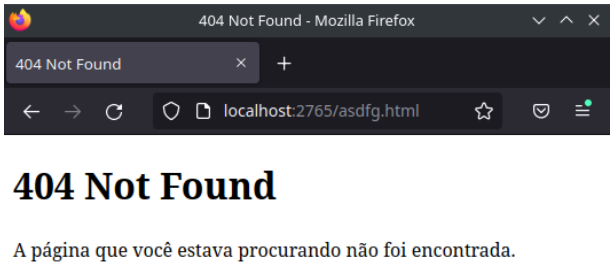

Figura 9 – Acesso à URL de arquivo inexistente.

```
Last-Modified: Thu Nov 23 19:29:22 2023 BRT
Content-Length: 201
Content-Type: text/html; charset=utf-8
```

4.2.3 Teste 3 - Arquivo inexistente

Com uma url de um recurso inexistente, vemos a página de erro 404, vista na Figura 9.

Requisição realizada:

```
GET /asdfg.html HTTP/1.1
Host: localhost:2765
User-Agent: Mozilla/5.0<cortado>Gecko/20100101 Firefox/102.0
<cortado>
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
```

Cabeçalho da resposta:

```
HTTP/1.1 404 Not Found
Date: Wed Dec 06 07:51:58 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: keep-alive
Last-Modified: Thu Nov 23 19:29:24 2023 BRT
Content-Length: 207
Content-Type: text/html; charset=utf-8
```

4.2.4 Teste 4 - Diretório sem permissão de execução

Acessando o diretório ‘dir’, vemos a página de erro mostrada na Figura 10.

Figura 10 – Acesso à diretório sem permissão de execução.

Requisição realizada:

```
GET /dir HTTP/1.1
Host: localhost:2765
User-Agent: Mozilla/5.0<cortado>Gecko/20100101 Firefox/102.0
<cortado>
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
```

Cabeçalho da resposta:

```
HTTP/1.1 403 Forbidden
Date: Wed Dec 06 07:56:00 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: keep-alive
Last-Modified: Thu Nov 23 19:29:22 2023 BRT
Content-Length: 201
Content-Type: text/html; charset=utf-8
```

4.2.5 Teste 5 - Acesso à arquivo dentro de diretório sem permissão de execução

O diretório sem permissão de execução é o mesmo do teste anterior, o arquivo acessado é 'texto_de_teste.txt'. O erro deve ser 403, pois não é permitido acessar este diretório. Como visto na Figura 11 é isto que ocorre.

Requisição realizada:

```
GET /dir/texto_de_teste.txt HTTP/1.1
Host: localhost:2765
User-Agent: Mozilla/5.0<cortado>Gecko/20100101 Firefox/102.0
<cortado>
```

Figura 11 – Acesso à um arquivo que existe mas dentro de um diretório sem permissão de execução.

Figura 12 – Acesso à recurso inexistente dentro de um diretório sem permissão de execução.

```
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
```

Cabeçalho da resposta:

```
HTTP/1.1 403 Forbidden
Date: Wed Dec 06 07:59:05 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: keep-alive
Last-Modified: Thu Nov 23 19:29:22 2023 BRT
Content-Length: 201
Content-Type: text/html; charset=utf-8
```

4.2.6 Teste 6 - Acesso à recurso inexistente dentro de diretório sem permissão de execução

Teste similar ao anterior, porém acessando um recurso que não existe. O erro deve ser o mesmo, pois o servidor não deve dar informações sobre o interior de um diretório sem permissão de execução. Isto é o que pode ser visto na Figura 12

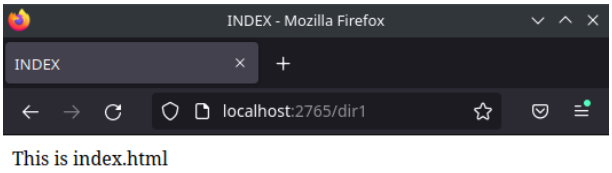

Figura 13 – Acesso à diretório contendo *index.html* apenas.

Requisição realizada:

```
GET /dir/asdkjfn HTTP/1.1
Host: localhost:2765
User-Agent: Mozilla/5.0<cortado>Gecko/20100101 Firefox/102.0
<cortado>
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
```

Cabeçalho da resposta:

```
HTTP/1.1 403 Forbidden
Date: Wed Dec 06 08:03:36 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: keep-alive
Last-Modified: Thu Nov 23 19:29:22 2023 BRT
Content-Length: 201
Content-Type: text/html; charset=utf-8
```

4.2.7 Teste 7 - Acesso à diretório com index.html

Ao requisitar um diretório, o servidor deve retornar o arquivo *index.html* que nele existir ou o *welcome.html*. Acessando o diretório 'dir1', que tem apenas o primeiro deles. Vemos a página mostrada na Figura 13

Requisição realizada:

```
GET /dir1 undefined
Host: localhost:2765
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
<cortado>
Sec-Fetch-Mode: navigate
```

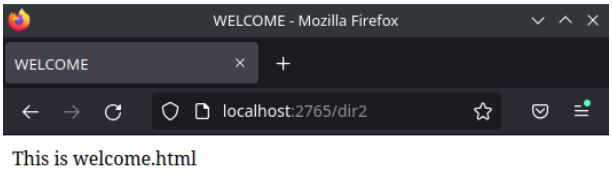

Figura 14 – Diretório com *welcome.html* apenas.

```
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
```

Cabeçalho da resposta:

```
HTTP/1.1 200 OK
Date: Wed Dec 06 08:12:23 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: keep-alive
Last-Modified: Sat Apr 07 11:24:40 2007 BRT
Content-Length: 91
Content-Type: text/html; charset=utf-8
```

4.2.8 Teste 8 - Acesso à diretório contendo apenas welcome.html

Similar ao anterior, mas para a segunda possibilidade de arquivo padrão. O resultado da requisição pode ser visto na Figura 14

Requisição realizada:

```
GET /dir2 HTTP/1.1
Host: localhost:2765
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
<cortado>
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
If-Modified-Since: Sat Apr 07 11:24:40 2007 BRT
```

Cabeçalho da resposta:

```
HTTP/1.1 200 OK
Date: Wed Dec 06 08:15:07 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
```

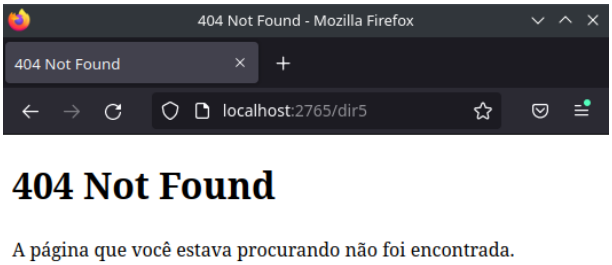

Figura 15 – Diretório sem *index.html* nem *welcome.html*

```
Connection: keep-alive
Last-Modified: Sat Apr 07 11:24:40 2007 BRT
Content-Length: 95
Content-Type: text/html; charset=utf-8
```

4.2.9 Teste 9 - Acesso à diretório sem nenhum dos arquivos padrão.

Neste caso a reposta esperada é 404, que é o que ocorre. O resultado pode ser visto na Figura 15.

Requisição realizada:

```
GET /dir5 HTTP/1.1
Host: localhost:2765
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
<cortado>
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
```

Cabeçalho da resposta:

```
HTTP/1.1 404 Not Found
Date: Wed Dec 06 08:16:55 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: keep-alive
Last-Modified: Thu Nov 23 19:29:24 2023 BRT
Content-Length: 207
Content-Type: text/html; charset=utf-8
```

Figura 16 – Diretório com *index.html* e *welcome.html*, ambos sem permissão de leitura.

4.2.10 Teste 10 - Acesso à diretório com os dois arquivos padrão sem permissão de leitura.

Aqui o erro é 403, pois os arquivos existem mas não podem ser lidos. O resultado pode ser visto na Figura 16.

Requisição realizada:

```
GET /dir4 HTTP/1.1
Host: localhost:2765
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
<cortado>
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
```

Cabeçalho da resposta:

```
HTTP/1.1 403 Forbidden
Date: Wed Dec 06 08:20:29 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: keep-alive
Last-Modified: Thu Nov 23 19:29:22 2023 BRT
Content-Length: 201
Content-Type: text/html; charset=utf-8
```

4.2.11 Teste 11 - Diretório com index.html sem permissão de leitura mas com welcome.html legível

Neste caso espera-se que o arquivo *welcome.html* seja entregue. Isto é visto na Figura 17.

Requisição realizada:

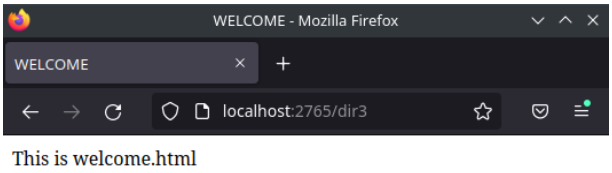

Figura 17 – diretório com *index.html* sem permissão de leitura e *welcome.htmt* liberado.

Figura 18 – Diretório com *index.html* sem permissão de leitura e sem *welcome.html*

```
GET /dir3 HTTP/1.1
Host: localhost:2765
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
<cortado>
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
```

Cabeçalho da resposta:

```
HTTP/1.1 200 OK
Date: Wed Dec 06 08:24:45 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: keep-alive
Last-Modified: Sat Apr 07 11:24:40 2007 BRT
Content-Length: 95
Content-Type: text/html; charset=utf-8
```

4.2.12 Teste 12 - Diretório com index.html sem permissão de leitura e sem welcome.html

Aqui o erro é 403, pois mesmo o arquivo *welcome.html* não existindo, esperaria-se que o servidor retornasse *index.html* e isto só não é possível porque ele não tem permissão de leitura. O resultado está na Figura 18

Requisição realizada:

```
GET /dir3.1 HTTP/1.1
Host: localhost:2765
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
<cortado>
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
```

Cabeçalho da resposta:

```
HTTP/1.1 403 Forbidden
Date: Wed Dec 06 08:27:31 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: keep-alive
Last-Modified: Thu Nov 23 19:29:22 2023 BRT
Content-Length: 201
Content-Type: text/html; charset=utf-8
```

4.2.13 Teste 13 - Método HEAD

O servidor deve retornar apenas apenas o cabeçalho da requisição. Realizando esta operação no arquivo *index.html* na raíz do projeto temos

Requisição realizada (*telnet*):

```
HEAD /index.html HTTP/1.1
```

Resposta completa:

```
HTTP/1.1 200 OK
Date: Wed Dec 06 08:33:08 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: close
Last-Modified: Wed Dec 06 08:08:11 2023 BRT
Content-Length: 3742
Content-Type: text/html; charset=utf-8
```

Realizando agora a mesma operação, mas no diretório '/', temos

Requisição realizada (*telnet*):

```
HEAD / HTTP/1.1
```

Resposta completa:

```
HTTP/1.1 200 OK
Date: Wed Dec 06 08:35:06 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: close
Last-Modified: Wed Dec 06 08:08:11 2023 BRT
Content-Length: 3742
Content-Type: text/html; charset=utf-8
```

Que é a mesma resposta, conforme o esperado. Portanto HEAD funciona tanto em diretórios quando arquivos.

4.2.14 Teste 14 - Método OPTIONS

A saída do método OPTIONS depende do recurso que é solicitado, pois quando a requisição é feita na URL de troca de senha também é possível realizar requisições do tipo POST.

Requisição realizada (*telnet*):

```
OPTIONS / HTTP/1.1
```

Resposta completa:

```
HTTP/1.1 200 OK
Allow: GET, HEAD, OPTIONS, TRACE
Date: Wed Dec 06 08:36:53 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: close
Content-Length: 0
Content-Type: text/html
```

Requisição realizada (*telnet*):

```
OPTIONS /dir1/dir11/change_password.html HTTP/1.1
```

Resposta completa:

```
HTTP/1.1 200 OK
Allow: GET, HEAD, POST, OPTIONS, TRACE
Date: Wed Dec 06 08:38:04 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: close
Content-Length: 0
Content-Type: text/html
```

Note que neste segundo caso o POST também está presente.

4.2.15 Teste 15 - Método TRACE

Este método responde com um status 200 e com os mesmos cabeçalhos que foram enviados na requisição. Por exemplo:

Requisição realizada (*telnet*):

```
TRACE / HTTP/1.1
Host: localhost:2765
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
```

Resposta completa:

```
HTTP/1.1 200 OK
Host: localhost:2765
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
```

4.2.16 Teste 16 - Método não implementado no servidor

Para este foi enviado o método HTTP, PUT.

Requisição realizada (*telnet*):

```
PUT / HTTP/1.1
```

Resposta completa:

```
HTTP/1.1 501 Not Implemented
Date: Wed Dec 06 08:42:15 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: close
Content-Length: 0
Content-Type: text/html
```

Not Implemented indica que o servidor entendeu que o método é válido e indica que mesmo que seja, ele não foi implementado no servidor.

4.2.17 Teste 17 - Requisição inválida

Enviar algo estranho para o navegador, uma sequência de caracteres que não forme um cabeçalho válido, causa erro de sintaxe e o status 400 é retornado.

Requisição realizada (*telnet*):

```
SADFKFBSAJDBHF
```

Resposta completa:

```
HTTP/1.1 400 Bad Request
Date: Wed Dec 06 08:44:55 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: close
Content-Length: 0
Content-Type: text/html
```

4.2.18 Teste 18 - Servidor ocupado

Limitando o número de threads à 2, foi feito o seguinte teste: Abrindo duas conexões via *telnet* e a terceira no browser. A saída vista no browser é a mostrada na Figura 19.

Requisição realizada:

```
GET / HTTP/1.1
Host: localhost:2764
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
<cortado>
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
```

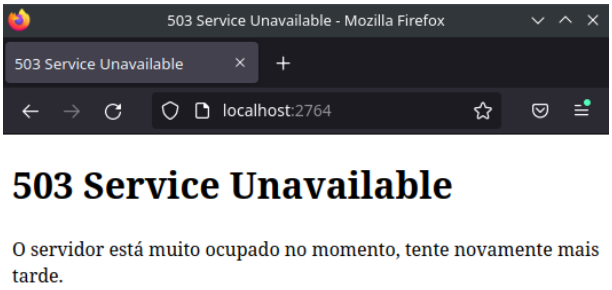

Figura 19 – Erro de servidor muito ocupado.

Cabeçalho da resposta:

```
HTTP/1.1 503 Service Unavailable
Date: Wed Dec 06 08:48:13 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: close
Last-Modified: Thu Nov 23 19:29:28 2023 BRT
Content-Length: 239
Content-Type: text/html; charset=utf-8
```

4.2.19 Teste 19 - Acesso invasivo, fora do webspace

Utilizando '..' na URL é possível subir diretórios, pode-se tentar fazer isto para sair do webspace e acessar arquivos pela máquina. O servidor não deve permitir que isto ocorra, e realmente não deixa.

Requisição realizada (*telnet*):

```
GET /dir1/../../teste HTTP/1.1
```

Cabeçalho da resposta:

```
HTTP/1.1 403 Forbidden
Date: Wed Dec 06 08:53:33 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: close
Last-Modified: Thu Nov 23 19:29:22 2023 BRT
Content-Length: 201
Content-Type: text/html; charset=utf-8
```

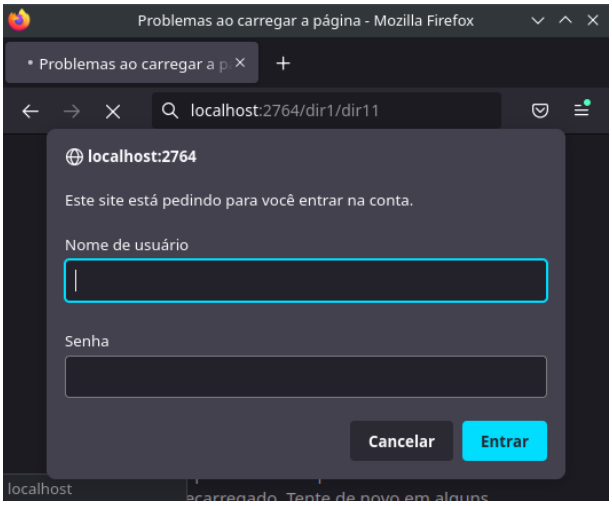

Figura 20 – Autenticação requisitada

4.2.20 Teste 20 - Uso de '..' na URL com sucesso

Também é possível utilizar '..' na URL sem ser barrado, desde que o recurso especificado esteja sempre dentro do webspace. A URL não deve sair do webspace em momento algum ao ser percorrida.

Requisição realizada (*telnet*):

```
HEAD /dir1/../dir2/ HTTP/1.1
```

Cabeçalho da resposta:

```
HTTP/1.1 200 OK
Date: Wed Dec 06 08:56:16 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: close
Last-Modified: Sat Apr 07 11:24:40 2007 BRT
Content-Length: 95
Content-Type: text/html; charset=utf-8
```

4.2.21 Teste 21 - Acesso à diretório com .htaccess

O diretório contém um arquivo *index.html*, então este deve ser retornado. Como ele é protegido, primeiramente é requisitada autenticação, como é possível ver na Figura 20. Foram inseridas as informações do usuário `dir11user1`. A senha deste usuário foi criptografada com salt de dois caracteres então o teste também mostra que este tipo de salt é reconhecido corretamente.

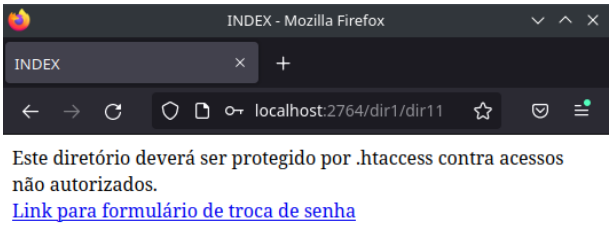

Figura 21 – Página *index.html* visível após preencher os dados de login.

Após inserir os dados e enviar a autenticação é possível ver a página index.html do diretório, isso é mostrado na Figura 21.

No cabeçalho da requisição neste segundo caso é possível ver o campo com os dados da autenticação.

```
GET /dir1/dir11 HTTP/1.1
Host: localhost:2764
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
<cortado>
Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Authorization: Basic ZGlyMTF1c2VyMTpkaXIxMXVzZXIxUGFzc3dvcmQ=
<cortado>
Sec-Fetch-Site: cross-site
```

4.2.22 Teste 22 - Acesso a diretório filho do anterior mas com outro .htaccess

Aqui, o servidor deve aceitar apenas os dados de autenticação do *.htaccess* interno, mais próximo do recurso requisitado. Então basta logar com os dados do usuário `dir111user2` e ver se ele é aceito. Este usuário não está presente no outro *.htpassword* então não seria aceito por causa disso.

Além disso, a senha do usuário 2 foi criptografada em SHA-256, então o teste já identifica se esta criptografia também é reconhecida.

Novamente, acessando o diretório é recebida a tela de requisição de login (Figura 22) e ao preencher os dados e enviar vemos a página *index.html* (Figura 23).

4.2.23 Teste 23 - Arquivo protegido por .htaccess em algum diretório acima

Aqui é necessário que seja requisitada autenticação, pois a subárvore em que o arquivo se encontra é protegida, mesmo que não haja um arquivo *.htaccess* diretamente

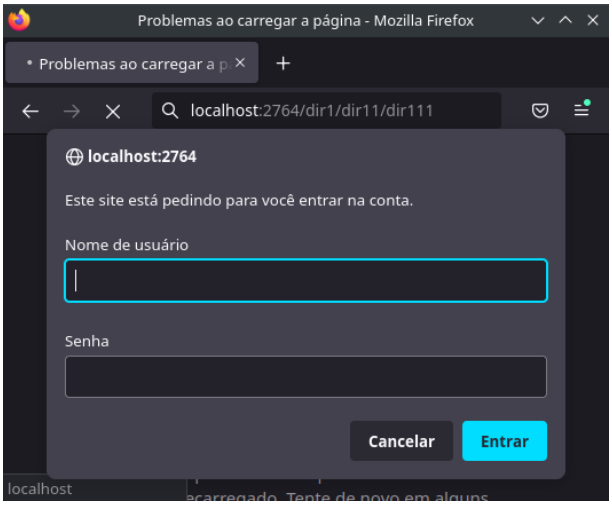

Figura 22 – Autenticação requisitada no diretório interno.

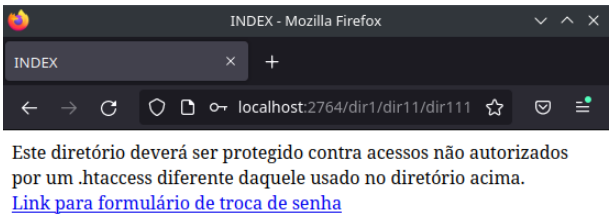

Figura 23 – Página recebida após autenticar-se

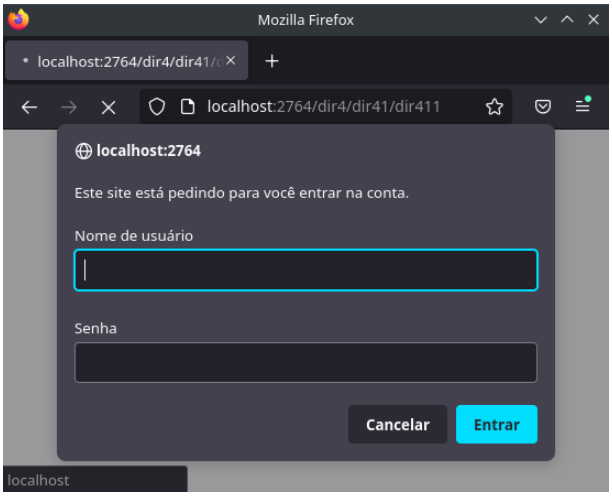

Figura 24 – Autenticação requisitada ao acessar recurso em subárvore protegida

no diretório que ele se encontra. Executando o teste, foi inserido os dados do usuário `dir41user3`, testando agora se a criptografia SHA-512 também é reconhecida.

Novamente é mostrada a tela de autenticação (Figura 24) e ao logar vemos a página *index.html* (Figura 25).

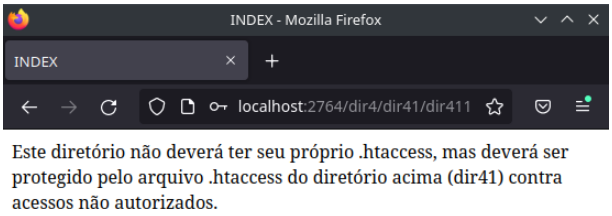

Figura 25 – Página recebida após autenticar-se

Figura 26 – Acesso negado ao *.htaccess*.

4.2.24 Teste 24 - Acesso direto ao arquivo .htaccess

Por questões de segurança, o servidor jamais deve entregar o arquivo *.htaccess*, mesmo que seja enviada autenticação correta. O resultado do teste está presente na Figura 26.

Requisição realizada:

```
GET /dir4/dir41/.htaccess HTTP/1.1
Host: localhost:2764
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) <cortado>
Accept-Encoding: gzip, deflate, br
Authorization: Basic ZGlyNDF1c2VyMzpkaXI0MXVzZXIzUGFzc3dvcmQ=
<cortado>
Sec-Fetch-User: ?1
```

Cabeçalho da resposta:

```
HTTP/1.1 403 Forbidden
Date: Wed Dec 06 09:24:05 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: keep-alive
Last-Modified: Thu Nov 23 19:29:22 2023 BRT
Content-Length: 201
Content-Type: text/html; charset=utf-8
```

4.2.25 Teste 25 - Erro na autenticação

Ao inserir dados incorretos, o servidor envia novamente o cabeçalho requisitando autenticação, então o formulário de login aparece novamente. Observando a requisição e resposta é possível ver o ocorrido.

Requisição realizada:

```
GET /dir4/dir41 HTTP/1.1
Host: localhost:2764
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
<cortado>
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Authorization: Basic c2FkZmFzZGZhc2RmOmFzZGZhc2RmYXNkZmFzZGY=
```

Cabeçalho da resposta:

```
HTTP/1.1 401 Authorization Required
Date: Wed Dec 06 09:37:06 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
WWW-Authenticate: Basic realm="/dir4/dir41"
Connection: keep-alive
Content-Length: 0
Content-Type: text/html
```

Como é possível ver, mesmo que a requisição tenha dados de autenticação, o servidor retorna o cabeçalho *WWW-Authenticate*, pois os dados estão incorretos. Isto ocorre de forma independente com a existência ou não do usuário, para aumentar a segurança do servidor.

4.2.26 Teste 26 - Diretório com .htaccess referenciando .htpassword inexistente

Este teste serve para mostrar um erro de configuração do servidor, em que o arquivo de senhas especificado em *.htaccess* não consegue ser encontrado. Ao tentar entrar neste diretório vemos o erro da Figura 27

O mesmo erro é visto ao tenter entrar no formulário de troca de senha do diretório. Nos dois casos nem é exibida a tela de autenticação para o cliente, pois o servidor detecta que o arquivo de senhas não pode ser aberto mesmo quando o usuário não envia dados de login.

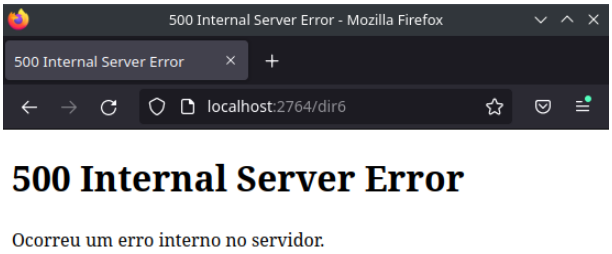

Figura 27 – Servidor mal configurado: *.htpassword* inexistente.

O mesmo erro seria visto caso ocorrese algum erro na abertura do arquivo .htaccess, por exemplo se ele não tivesse permissão de leitura.

4.2.27 Teste 27 - Troca de senha com sucesso

Para realizar este teste, primeiramente o formulário de troca de senha de 'dir1/dir11' foi acessado e preenchido com os dados do usuário `dir11user1` (Figura 28). A nova senha configurada foi 'password'. Então, foi exibida a página de sucesso na troca de senha (Figura 29). Retornando ao index.html do mesmo diretório, foi requisitada autenticação novamente (Figura 30) e ao inserir os novos dados (nova senha) o acesso é liberado (Figura 31).

Acessando o arquivo de senhas do *.htaccess* daquele diretório, vemos que a senha criptografada foi alterada. É mais fácil ver a mudança com a ferramenta *diff* pelo VSCode (Figura 32).

Em seguida, foram realizados os mesmos testes, para alterar a senha do usuário `dir111user3`, presente em outro arquivo de senhas. A sequência foi bem sucedida e ao seu fim é possível ver a alteração feita no outro arquivo (Figura 33).

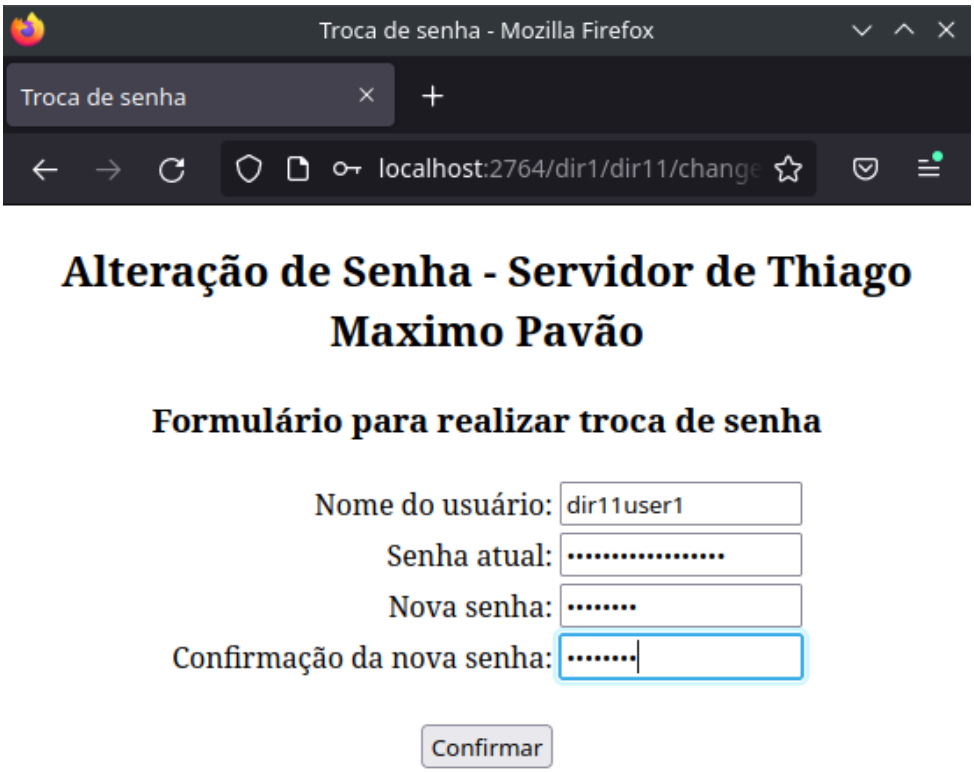

Figura 28 – Formulário de troca de senha preenchido com os dados de dir11user1 e nova senha 'password'.

Figura 29 – Sucesso na troca de senha.

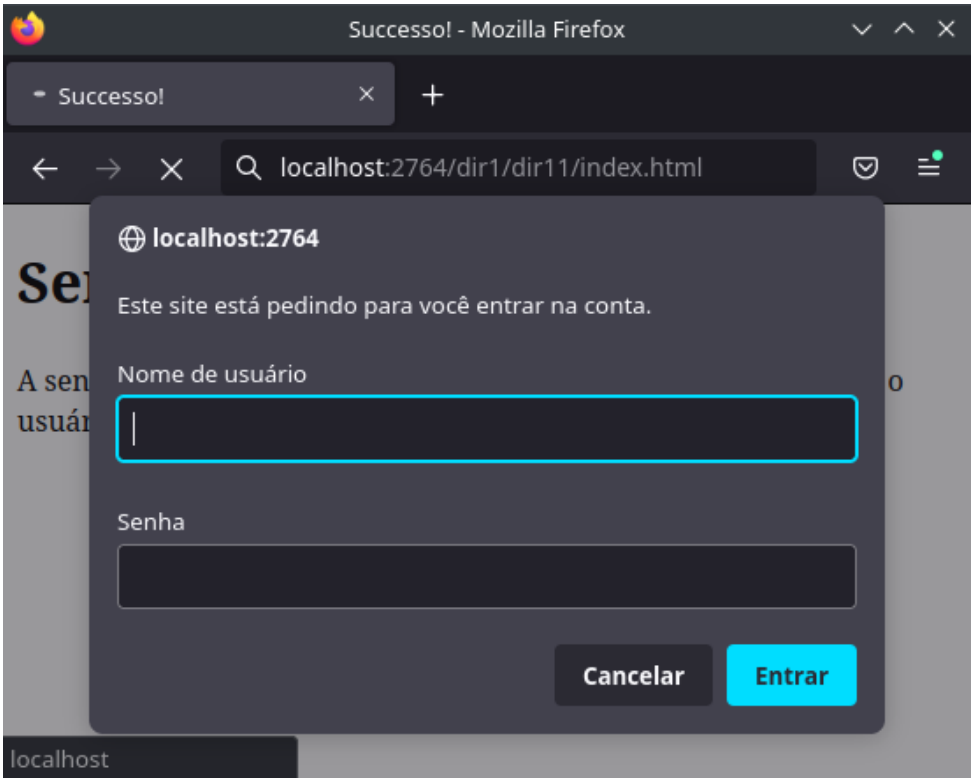

Figura 30 – Autenticão requisitada novamente, normalmente esta tela não seria mostrada pois eu já havia logado com o usuário anteriormente e o navegador salva os dados no cache. Ela aparecer indica que o navegador tentou utilizar os dados para requisitar a página mas não teve sucesso.

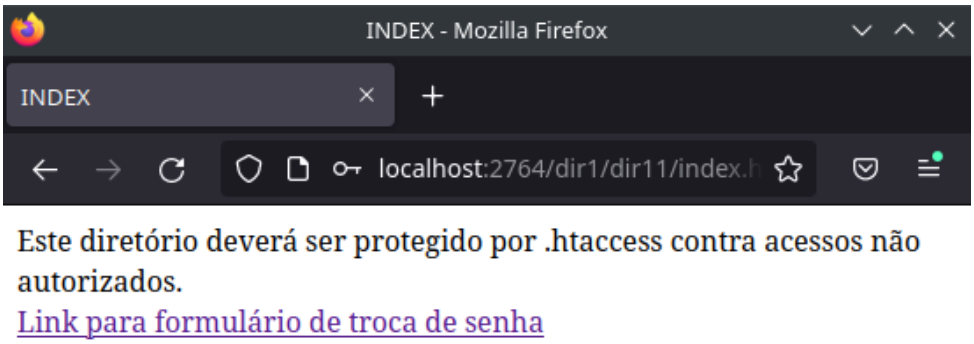

Figura 31 – Nova autenticação aceita com sucesso.

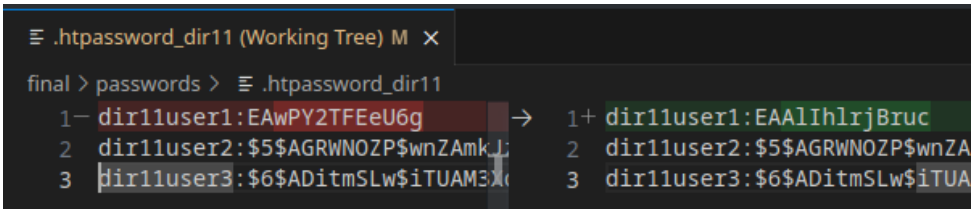

Figura 32 – Senha alterada, salt de 2 caracteres mantido.

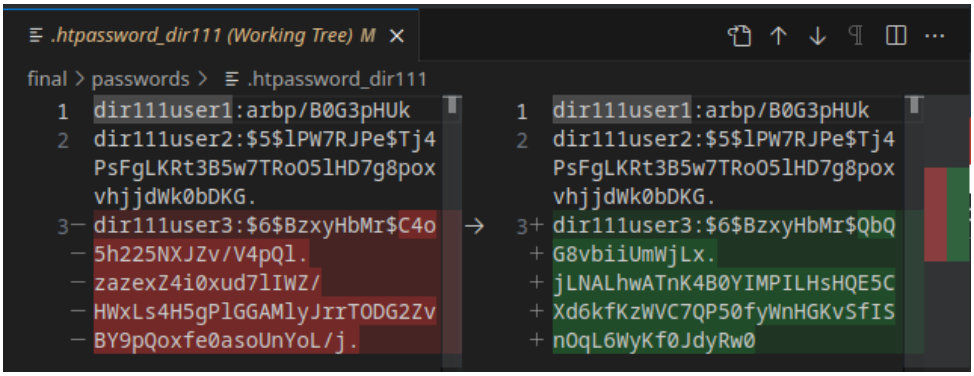

Figura 33 – Senha SHA-512 alterada, salt mantido.

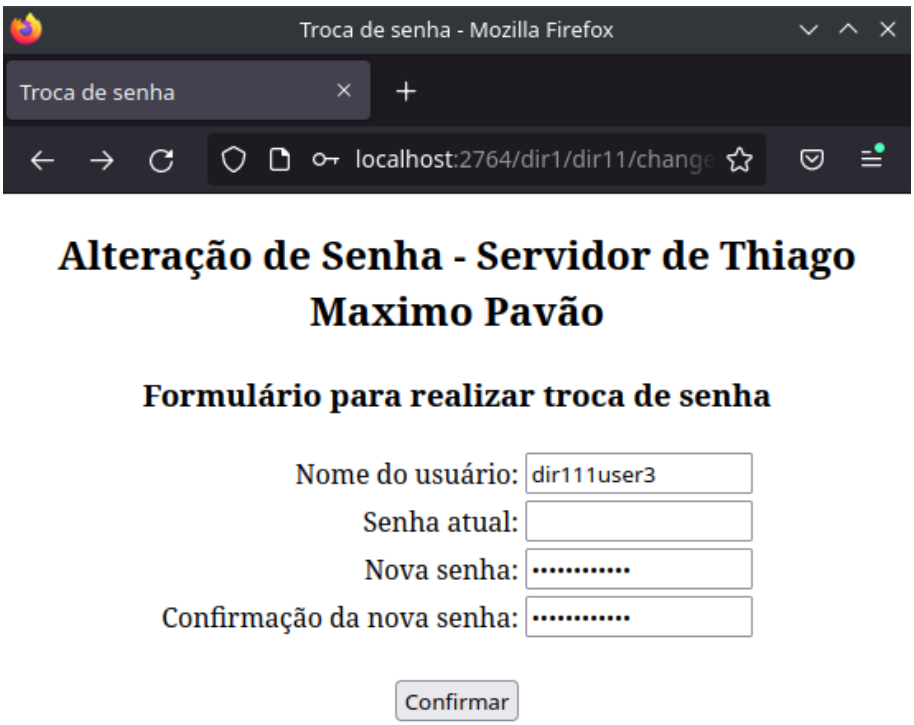

Figura 34 – Formulário preenchido faltando um campo (senha atual)

4.2.28 Teste 28 - Erro na troca de senha: Campo não preechido

Enviar o formulário sem um dos campos preenchidos (Figura 34) causa um erro no momento em que o servidor lê os campos do corpo da requisição, é enviada de volta uma página indicando o problema (Figura 35). Nenhum arquivo de senha é alterado.

4.2.29 Teste 29 - Erro na troca de senha: Autenticação inválida

Enviar o formulário com um usuário inválido (Figura 36) faz com que seja enviada de volta uma página indicando o problema (Figura 37). Nenhum arquivo de senha é alterado.

Figura 35 – Resposta à falta de campos no preenchimento do formulário

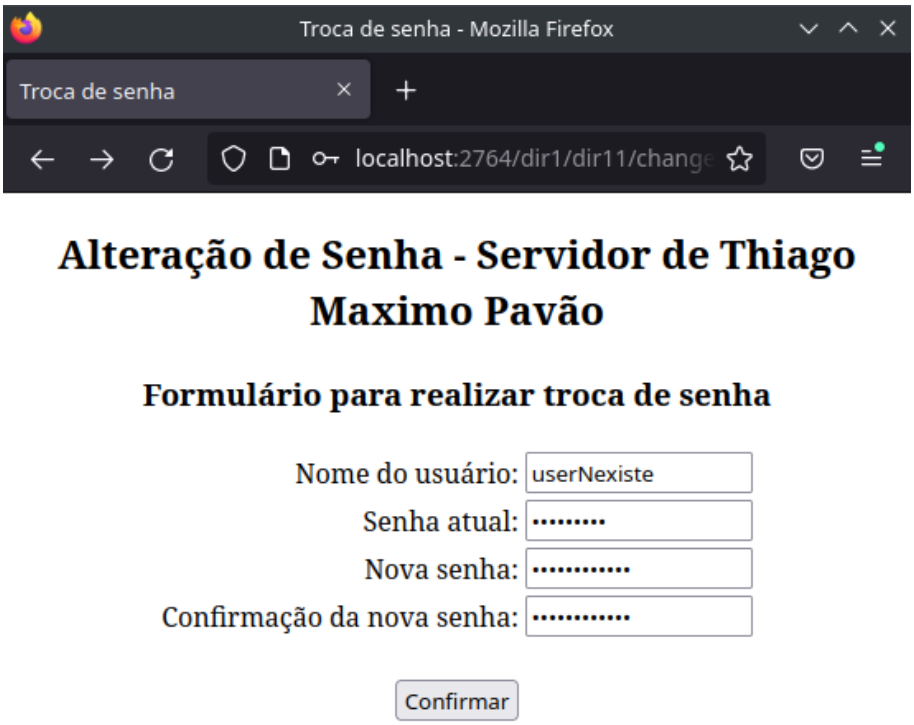

Figura 36 – Autenticação atual inválida no formulário de troca de senha.

Figura 37 – Resposta à autenticação inválida no formulário.

Figura 38 – Campos de nova senha com valores diferentes no formulário.

4.2.30 Teste 30 - Erro na troca de senha: Senhas não coincidentes

Enviar o formulário com senhas diferentes nos campos de nova senha e repetição de nova senha (Figura 38) faz com que seja enviada de volta uma página indicando o problema (Figura 39). Nenhum arquivo de senha é alterado.

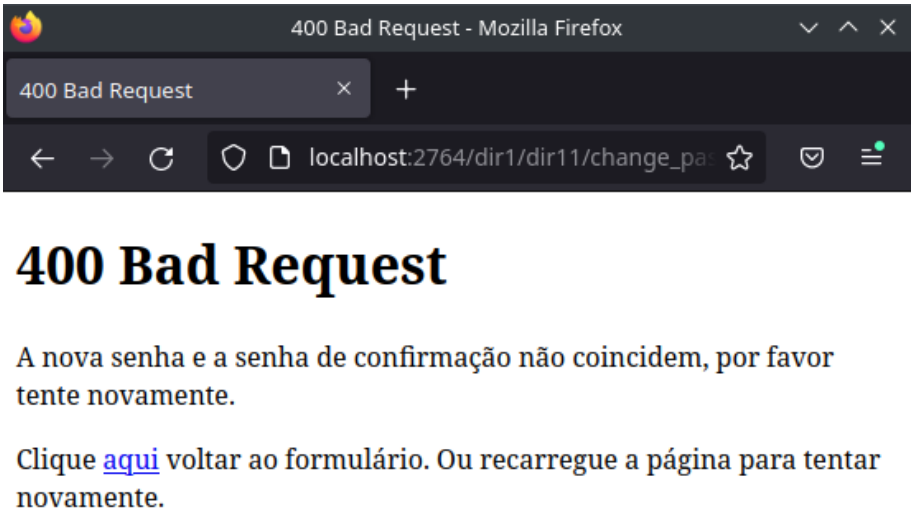

Figura 39 – Resposta à campos de nova senha com valores diferentes no formulário.

4.2.31 Teste 31 - POST em URL que não é a de troca de senha.

Neste caso o erro não é 501 *Not Implemented*, pois o método POST existe, não sendo suportado apenas na URL enviada.

Requisição realizada (*telnet*):

```
POST / HTTP/1.1
```

Cabeçalho da resposta:

```
HTTP/1.1 405 Method Not Allowed
Date: Wed Dec 06 12:05:10 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: close
Last-Modified: Thu Nov 23 19:48:13 2023 BRT
Content-Length: 208
Content-Type: text/html
```

Também é enviada uma página explicando o erro.

4.2.32 Teste 32 - POST em URL de troca de senha mas em diretório desprotegido.

Este erro ocorre se ao tentar encontrar o arquivo de senhas para realizar a troca de senha o servidor descobrir que no diretório não há sequer um arquivo *.htaccess*.

Requisição realizada (*telnet*):

```
POST /change_password.html HTTP/1.1
```

Cabeçalho da resposta:

```
HTTP/1.1 400 Bad Request
Date: Wed Dec 06 12:07:52 2023 BRT
Server: Servidor HTTP ver. 1.0 de Thiago Maximo Pavao
Connection: close
Last-Modified: Thu Nov 23 20:03:36 2023 BRT
Content-Length: 240
Content-Type: text/html
```

Também é enviada uma página explicando o erro.

5 Compilação e Execução

5.1 Download

O código do projeto final pode ser encontrado e baixado localmente do github. Em <https://github.com/ThiagoMaxPavao/EA872---Unicamp> na pasta 'final'.

5.2 Compilação

O comando executável é chamado 'server'. Para gerá-lo é preciso utilizar primeiramente as ferramentas *flex* e *yacc* e então combinar todos os arquivos *.c* em um executável.

É necesário executar os comandos a seguir.

1. `bison -o sint.tab.c --defines=sint.tab.h sint.y` : Cria tabela de definições de tokens para o flex e o analisador sintático sint.tab.c
2. `flex -o lex.yy.c --header-file=lex.yy.h lex.l` : Cria o analisador léxico lex.yy.c
3. `gcc get.c util.c lex.yy.c sint.tab.c server.c base64.c -o server -lfl -ly -lcrypt -lpthread` : Combina os arquivos contendo os códigos dos analisadores, utilitários e principal do servidor

O comando do gcc conta com diversas flags, são elas

- -lfl : para o flex
- -ly : para o bison/yacc
- -lcrypt : para a chamada de sistema de criptografia crypt
- -lpthread : para o uso de chamadas de sistema relativas à threads

5.3 Execução

Para executar o servidor, basta chamar o programa *server* que é gerado na compilação. Sua forma de uso é

Uso: `./server <Web Space> <Porta> <Arquivo de Log> <URL de troca de senha> <Max threads> [charset (tipo de codificacao)]`

Exemplo: `./server ~/webspace 1234 registro.txt change_password.html 10 utf-8`

Explicação do exemplo:

- Projeta para a web a pasta na home do usuário: webspace
- Abre o servidor na porta 1234
- Salva todos os logs em um arquivo chamado registro.txt
- Envia o formulário de troca de senha na URL terminada em change_password.html para qualquer subdiretório protegido
- Abre no máximo 10 threads para atender clientes
- Envia como codificação para arquivos do tipo texto charset=utf-8, no cabeçalho Content-Type

6 Considerações finais

6.1 Limitações da implementação

A primeira limitação é a impossibilidade de criação de usuários no webspace de forma remota. Eles devem ser criados manualmente e é possível apenas realizar a troca de senha pela rede.

Outra limitação é a forma de implementação da requisição POST. O código foi feito para aceitar POST apenas nas URLs de troca de senha e sabe como processar este tipo de requisição. A forma de implementação não permite que outros processamentos para outros tipos de POST sejam feitos de forma simples.

Também foi possível perceber um desempenho abaixo do esperado na velocidade do servidor, mesmo com a implementação dos analisadores reentrantes e da redução de regiões críticas ao máximo. Talvez parte da lógica seja feita de forma pouco eficiente ou o computador em que foram feitos os testes tenha um baixo desempenho.

6.2 Possíveis melhorias

Algumas melhorias possíveis são

- Configurar o encerramento da thread quando o seu cliente desconecta-se. Pois existem casos em que ao receber o cabeçalho “Last-Modified”, o cliente percebe que não precisa do arquivo (pois já tem em cache) e encerra a conexão.
- Criar uma forma de deletar e inserir usuários nos arquivos de senha pela rede, sem a necessidade manual de editar o arquivo de senhas.
- Melhorias de despenho do servidor, como carregar as páginas de erro na memória e não precisar abri-las toda vez que um erro ocorre.
- Implementar mais métodos do HTTP, como DELETE, PUT e outros. Expandindo as funcionalidades do servidor
- Interpretar melhor os cabeçalhos enviados pelo cliente. Por exemplo, o cliente especifica o tipo de arquivo que aceita com o cabeçalho “Accept”, este poderia ser lido para tomar decisões mais precisas. O memso poderia ser feito para os outros cabeçalhos que são ignorados.

6.3 Comentários sobre a disciplina

Sinto que aprendi muito com as aulas e com as atividades desenvolvidas ao longo da disciplina.

Minha primeira sugestão seria a melhor especificação do que deve ser entregue em cada atividade, para que não hajam desentendimentos como falta do *webspace* ou do *código* na entrega.

Outra coisa seria dar mais ênfase na importância de cada atividade para as demais, incentivando que elas sejam bem desenvolvidas para poderem ser bem reutilizadas depois.

No mais, agradeço pelas aulas e pelo semestre :D