Servidor HTTP em C

Caio Henrique Suzuki Polidoro[1], **Davi Ferreira Santo**[1]

[1]Faculdade de Computação – Universidade Federal de Mato Grosso Sul (UFMS)
Caixa Postal 549 – 79.070-900 – Campo Grande – MS – Brazil

{caio.polidoro,davi.santo}@aluno.ufms.br

1. Introdução

Neste trabalho foi implementado um servidor HTTP simples em linguagem C, o servidor permite conexões múltiplas de clientes via navegador que desejam realizar o *download* de arquivos. Estabeleceu-se como objetivo dos alunos envolvidos expandir seu conhecimento a respeito do protocolo HTTP e o tratamento das requisições.

2. Funcionalidades implementadas

Nesta seção serão brevemente explicadas as funcionalidades realizadas com alguns detalhes de implementação.

2.1. Funcionamento em dois modos com porta padrão

Implementou-se tanto o funcionamento do servidor via *fork* ou *thread*, utilizando parâmetros passados pelo usuário no terminal para selecionar a opção desejada. Os parâmetros seguem o padrão:

- httpd -f (porta)
  Desta forma, o usuário seleciona o funcionamento via *fork*, foi implementada também uma função de tratamento do sinal SIG_CHLD, para evitar processos zumbis.
- http -t N (porta)
  Desta forma, o usuário seleciona o funcionamento via *thread*, note que é necessário escolher a quantidade de *threads* a serem criadas. Este modo **não** funciona corretamente, veja a Seção 3.

Ambos os modos contam com uma porta padrão (8080), caso o usuário não forneça a porta desejada.

2.2. Base: recebimento de requisições e *download* de arquivos

Foram implementados o recebimento e interpretação das requisições enviadas por um cliente (neste caso, navegador), no qual decidiu-se dar atenção especial aos dados da primeira linha do cabeçalho de requisição, onde é possível coletar a informação de qual arquivo o usuário está solicitando e então enviá-lo através de um *buffer*, sendo assim possível enviar arquivos grandes em seções de bytes do tamanho do *buffer* escolhido (1024 bytes, mas pode ser facilmente alterado).
Vale ressaltar que foi implementado apenas o tratamento de requisições do tipo GET, caso outra requisição seja feita, o servidor não tratará a requisição, apenas exibindo erro (*Bad Request*).

2.3. Navegação em diretórios

Foi implementada a navegação de diretórios, ao acessar um diretório, é possível visualizar todos os arquivos presentes dentro do mesmo, inclusive outros diretórios, e então realizar o *download* dos arquivos, que serão exibidos, ou acessar outro diretório, exibindo seu conteúdo. Para a implementação, o que se fez foi a leitura do diretório requisitado através da função *readdir*, desta maneira foi possível listar seus arquivos e subpastas. Em posse dessa lista, o que se fez foi apenas criar um link em uma tag html que passava o endereço de cada elemento. Com esse html gerado, sempre que o usuário clicar em um dos links, o navegador gera automaticamente uma requisição ja conhecida e tratada pelo servidor, tornando o problema trivial.

2.4. CGI-BIN

Ao se deparar com uma requisição da seguinte forma:

```
GET <sp> /cgi-bin/<script>?{<var>=<val>}*{<var>=<val>}<sp>HTTP/1.1<crlf>
{Outras informações de cabeçalho}<crlf>*
<crlf>
```

O script será executado e o resultado será recebido pelo *socket*, pois foi feito o redirecionamento da saída padrão. A comunicação para captação dos parâmetros e a execução do script de acordo foi realizado através da variável de ambiente **QUERY_STRING**.

Foi escrito um programa de exemplo soma.c que recebe dois parâmetros, a e b, e exibe no navegador o resultado da soma de a e b.

2.5. Conexões persistentes

Nos cabeçalhos de requisição, o servidor verifica o campo *Connection*, que indica a intenção de manter uma conexão persistente ou não, sendo *keep-alive* para o primeiro caso e *close* para o segundo.

Na posse dessa informação o servidor toma a decisão de fechar o *socket* após atender a requisição ou manter a conexão e receber mais requisições mandando um cabeçalho de acordo.

2.6. Mensagens de erro ao usuário

Pela implementação feita, existem três casos: caso OK, no qual a requisição está no formato esperado e é possível atendê-la; caso *Bad Request*, onde a requisição não está no formato esperado, ou seja, o cabeçalho contém algum erro ou não é um GET; e ainda o caso *Not Found*, onde não é possível encontrar o arquivo/diretório desejado.

No caso OK, a requisição é atendida normalmente e exibe ao usuário no navegador; nos outros dois casos, é exibida uma tela informando ao usuário que houve um erro e qual erro foi.

3. Dificuldades de implementação

A principal dificuldade na implementação deste trabalho ocorreu na implementação do funcionamento no modo *thread*, que está funcionando parcialmente, com erros de motivos desconhecidos pelos acadêmicos. Como ocorre o erro: ao escolher o modo thread e a quantidade de threads arbitrária, o servidor atende normalmente algumas solicitações, até que após algumas requisições serem atendidas, ocorre um falha de segmentação causando com que o programa feche.

Outra dificuldade foi a implementação do CGI-BIN, que atualmente tem uma solução simples e que aparentemente é rápida, tomou-se tempo para entender exatamente como seria feito o redirecionamento da saída padrão, e após finalmente conseguir redirecioná-la, o navegador não conseguia interpretar o cabeçalho de resposta, ficando em uma tela em branco e que não era possível interagir, apesar de confirmar-se via telnet que o cabaçalho estava sendo enviado corretamente. Após a implementação da conexão persistente, a implementação do CGI-BIN passou a funcionar, não se sabe ao certo a conexão entre os eventos.

Mais um detalhe que ocorreu como dificuldade enfrentada pelos acadêmicos foi no recebimento do cabeçalho de requisição: sabe-se que um cabeçalho HTTP de requisição GET deve terminar com os quatro caracteres \r\n\r\n, utilizou-se então uma estrutura do-while para receber (recv) no *buffer* enquanto estes quatro caracteres não fossem detectados. Com isso seria possível utilizar o programa telnet para digitar cabeçalhos inteiros, pois sem esta implementação, ao apertar enter, o telnet entende como se a requisição já tivesse finalizado, isto limitava os testes à cabeçalhos de apenas uma linha. Infelizmente, a solução utilizada de detectar os 4 caracteres obrigatórios não funcionou como esperado: apesar do telnet funcionar corretamente, ao fazer testes no navegador, eventualmente ao clicar em um arquivo para baixá-lo, a requisição não chega ao servidor e o navegador fica ”carregando”.

4. Conclusão

Foi possível implementar a maior parte das funcionalidades propostas pela descrição do trabalho, com exceção do funcionamento utilizando *pool* de *threads*. O resultado é um servidor HTTP funcional para requisições GET que é capaz de exibir arquivos de diferentes extensões, navegação em pastas, execução de *scripts* CGI e que permite conexão persistente.