Universidade Federal São João del Rei
Curso de Ciência da Computação

Trabalho Prático 2: Documentação

Redes de Computadores I

Discente: Julio Cesar da Silva Rodrigues

Docente: Rafael Sachetto Oliveira

Novembro
2022

Conteúdo

1 Introdução **2**

2 Estrutura Geral **2**

3 Estruturas de Dados e Funções **3**
3.1 Função *socket_creation* 3
3.2 Função *read_header* 3
3.3 Função *client_response* 4
3.4 Função *http_parser* 4
3.5 Função *laod_file* 4
3.6 Função *iterative_server* 4
3.7 Função *fork_server* 4
3.8 Funções *thread_server* e *consumer* 5
3.9 Função *concurrent_server* 5

4 Análise de Resultados **5**

5 Limitações **7**

6 Compilação e Execução **7**

7 Considerações Finais **8**

1 Introdução

Como especificado no enunciado do trabalho prático proposto, serão apresentados de forma sumarizada, alguns aspectos abordados no desenvolvimento desta tarefa. Veremos as etapas envolvidas, soluções, obstáculos encontrados, entre outras observações, cujo objetivo eram estimular um melhor entendimento de programação de protocolos da camada aplicação e de transporte de forma prática. Para isto, é necessário novamente realizar a comunicação entre processos pela rede, utilizando informações como endereço de *IP*, nome de domínio, número de porta, hospedeiro, entre outras.

Diante desta introdução, a tentativa foi de implementar tais protocolos da camada de aplicação e transporte adaptados para um servidor *web*, ou seja, implementar um programa que seja capaz de lidar com um conjunto pequeno das mensagens que possuem o formato *HTTP*, enviadas como requisições pelos clientes. E como pressuposto, o programa também deve ser capaz de formular as respostas adequadas no mesmo formato para formar o par de comunicação completo entre cliente e servidor.

2 Estrutura Geral

Toda a concepção das estruturas de requisições e respostas, assim como as funções que às manipulam foram implementadas em linguagem C (excetuando a análise de resultados gráfica, que foi gerada utilizando a linguagem *Python*). A implementação foi desenvolvida com o auxílio da ferramenta de controle de versionamento GitHub, e encontra-se disponível publicamente no repositório `https://github.com/juliorodrigues07/web_server_c`.

A organização do código se apresenta de forma bastante simples. Existe apenas um diretório fonte que corresponde basicamente à todas as operações executadas pelos servidor. Neste diretório, existem apenas os *scripts* responsáveis pelo funcionamento dos servidores como um todo (.c e .h), e o principal que apenas direciona ao tipo de servidor desejado pelo usuário na utilização.

3 Estruturas de Dados e Funções

No *script server.h* (*header* do servidor), encontram-se as estruturas de dados que definem cada requisição e cada resposta, e estas contém as seguintes variáveis:

- *request*
  - *bytes_read*: Inteiro responsável por armazenar a quantidade de *bytes* lidos de uma requisição qualquer;
  - *header*: *String* que armazena o conteúdo da mensagem de requisição em si.
- *response*
  - *response_buffer*: *String* responsável por armazenar o conteúdo da mensagem de resposta;
  - *message_length*: Inteiro responsável por armazenar o tamanho da mensagem de resposta;
  - *status*: Inteiro responsável por armazenar o estado da mensagem de resposta (200, 404, ...).

Também são definidas algumas constantes importantes como número da porta, quantidade máxima de conexões, tamanho de cabeçalho, caminho e buffer, além claro de variáveis para controle das *threads*.

3.1 Função *socket_creation*

Esta função, baseada fortemente no código de apoio disponibilizado junto ao enunciado deste trabalho prático (*server_tcp.c*), é responsável apenas por configurar a preparação da conexão (*Opening, Binding, Listening...*) para a comunicação via *socket TCP* do servidor com processos de outras máquinas, ou neste caso, com processos da máquina local.

3.2 Função *read_header*

O propósito desta função é bastante simples, esta apenas realiza a leitura da mensagem de requisição *HTTP* enviada pelo cliente.

3.3 Função *client_response*

Nesta função, ocorrem o envio das respostas às requisições dos usuários. Atuando em conjunto com a função *read_header* mencionada anteriormente, esta primeiramente obtém a mensagem *HTTP* de requisição, para então submetê-la ao "analisador"de mensagens *HTTP* (*parser*). Após realizada esta análise, o servidor envia a resposta correspondente na conexão *socket TCP* existente entre o par.

3.4 Função *http_parser*

Esta função tem como principal funcionalidade executar a análise das mensagens *HTTP* de requisição que são enviadas pelos usuários. Por esta implementação apenas lidar com o método *GET*, a primeira verificação feita é se este esta presente na linha de requisição da mensagem. Caso não estiver, é enviada ao usuário um código indicando que a requisição enviada está mal formulada (400 - *Bad Request*).

Em casos mais gerais, esta função, com o auxílio de uma segunda que será explanada na próxima seção, fará a interpretação da mensagem de requisição *HTTP* recebida, para então direcionar as solicitações desejadas como arquivos, ou informar outros códigos de erro presentes na especificação do protocolo de mensagens.

3.5 Função *laod_file*

Esta função será a responsável por retornar as mensagens de erro ou os arquivos desejados, conforme o conteúdo especificado na linha de requisição da mensagem *HTTP* pelo usuário.

3.6 Função *iterative_server*

O tipo de servidor mais simples presente na implementação, este atende a requisição de apenas um cliente por vez (única conexão) de forma bloqueante, ou seja, um vez realizada a conexão, o servidor deverá aguardar indefinidamente até que o usuário envie-o uma mensagem de requisição *HTTP*;

3.7 Função *fork_server*

O modo de operação que apresentou maiores problemas, o servidor utilizando *fork* se mostrou bastante ineficiente devido à falta de ajuste no controle de quantos processos são criados ao longo do tempo à medida que chegam

mensagens de requisição. A partir do momento que é realizada a conexão, é criado um processo filho para responder à tal, mas há grandes deficiências na implementação deste tipo de servidor que serão discutidas mais adiante.

3.8 Funções *thread_server* e *consumer*

Talvez o tipo de servidor que apresentou o maior desafio na implementação, este modo de operação também possui defeitos, mas aparentemente não tão graves quanto o servidor baseado em *fork*. Aqui, um número fixo de *threads* é criado, utilizando como parâmetro uma constante definida no *header*. Após a criação das *threads*, uma fila de tarefas também de tamanho fixo é mantida, para serem alocadas às *threads* de acordo com disponibilidade para consumir as requisições.

3.9 Função *concurrent_server*

Surpreendentemente o tipo de servidor que não apresentou tantos obstáculos na implementação, este se difere em relação ao modo mais básico, apenas na sua forma de operação não bloqueante (não totalmente). Este tipo de servidor suporta várias conexões simultâneas, mas somente irá bloquear para responder as mensagens de requisição *HTTP*, quando de fato houverem dados a serem processados enviados por um cliente.

4 Análise de Resultados

Nesta seção, discutiremos algumas métricas obtidas utilizando a ferramenta *Siege*, na execução de testes com os diferentes servidores presentes na implementação.

É importante citar que os testes foram realizados mantendo o tempo fixo (dez segundos) em todos os modos de servidores e com quaisquer números de clientes simultâneos. Foram também utilizados parâmetros padrão da ferramenta, ou seja, o número máximo de usuários concorrendo é limitado à 255, e não foi atribuído nenhum intervalo específico o qual os usuários deveriam respeitar entre o envio das requisições.

Por fim, os testes foram executados sempre utilizando o mesmo arquivo como parâmetro (*cat.jpg*) na requisição, e também foram executados cinco vezes com cada número de clientes em cada modo de servidor, para então calcular a média e exibir as informações aproximadas. O gráfico relacionando o número de requisições respondidas por segundo, com a quantidade de usuários concorrente em cada modo de servidor é exibido na Figura 1.

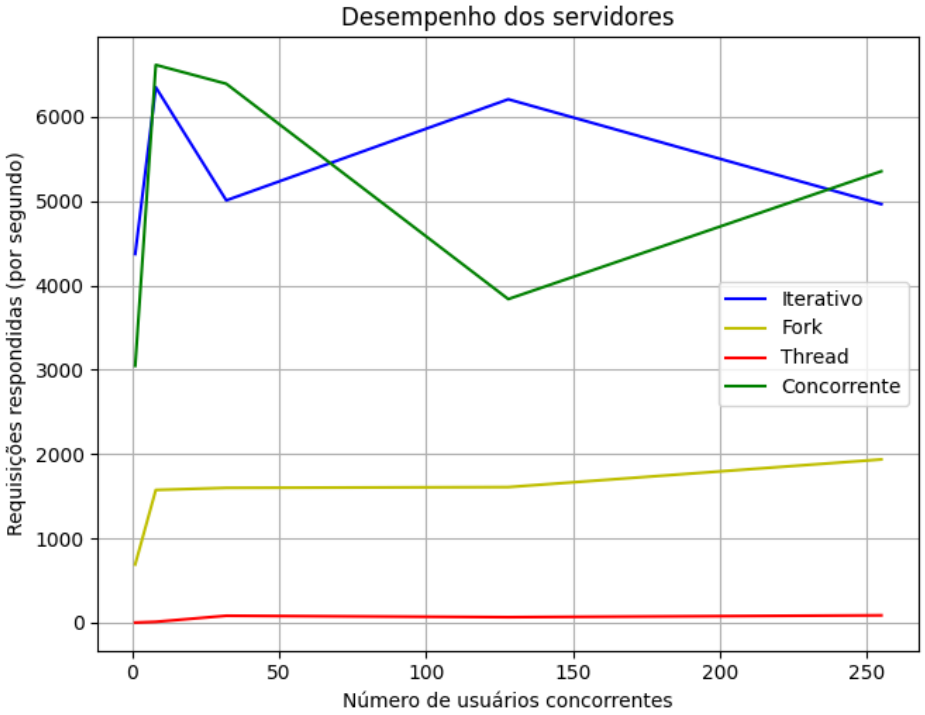

Figura 1: Gráfico comparativo do desempenho dos servidores

Podemos observar nitidamente no gráfico, que os dois tipos de servidores que presumidamente deveriam ser os que obteriam o melhor desempenho, apresentaram níveis bem abaixo da expectativa em relação aos seus concorrentes mais simples. Esperamos discutir melhor essa disparidade na próxima seção, apontando possíveis falhas que levaram ao desempenho muito abaixo do esperado pelos servidores *fork* e *thread*.

O resumo exibindo as médias dos valores brutos obtidos a partir dos testes de dez segundos são sumarizados na Tabela 1:

Tabela 1: Número de requisições respondidas por cada servidor (10 segundos)

Usuários	**Iterativo**	**Fork**	**Thread**	**Concorrente**
1	43728	6912	10	30468
8	63511	15743	96	66165
32	50090	15992	822	63921
128	62085	16085	646	38386
255	49643	19361	874	53546

5 Limitações

A principal falha encontrada até o momento na implementação atual, se encontra no servidor que utiliza *forks*. Esperar o término do processo filho não seria uma opção, já que isto faria basicamente com que seu funcionamento seguisse o princípio do servidor iterativo. O tempo de espera inserido (*timeout*) também não se mostra adequado, já que aparentemente vários processos "zumbis"são criados, acarretando em aumento demasiado no processamento e no nível de ocupação da memória principal (*memory leak*), à medida que são ampliados os parâmetros como tempo e número de clientes concorrentes.

A segunda falha está no servidor que utiliza *threads*. O tempo de ociosidade escolhido também não se mostra adequado, mantendo o servidor a maior parte do tempo omisso à resposta das requisições. Assim como o servidor *fork*, deve-se encontrar o equílibrio nos parâmetros com testes intensivos para evitar a criação de *forks* ou *threads* em excesso ou de forma escassa.

6 Compilação e Execução

Antes de discutir quaisquer detalhes de compilação e execução, é importante citar que a execução dos vários tipos de servidor pode requerer uma mudança no número da porta para possibilitar a abertura da conexão no *socket TCP*. Esta alteração deve ser feita diretamente no código (*server.h*) onde está definida a constante do número de porta (*PORT_NUMBER*).

Para compilar o programa, basta executar o seguinte comando via terminal:

make

Na execução, basta executar o comando no seguinte modelo via terminal:

make $<$ *nome do servidor* $>$

$<$ *nome do servidor* $>$ $\rightarrow$ [iterative, fork, thread ou concurrent]

Para testar utilizando o *software Siege*, basta digitar o comando no seguinte modelo em outra aba de terminal:

siege -twS -cx http://localhost:y/z

Em que:

- w: Tempo em segundos;
- x: Quantidade de usuários simultâneos;
- y: Número da porta;
- z: Nome ou caminho do arquivo.

Segue um exemplo:

siege -t10S -c128 http://localhost:2000/cat.jpg

Também pode-se testar individualmente pelo navegador, executando a seguinte pesquisa exemplo:

http://127.0.0.1:2000/index.html

Por fim, os arquivos presentes no diretório *files* são exibidos na Tabela 2:

Tabela 2: Arquivos presentes no diretótio *files*

400.html	404.html	413.html
index.html	ca.jpg	f1.gif
	batman.mp4	

7 Considerações Finais

Neste trabalho, foi construído parte da comunicação entre um par (*Cliente-Servidor*) para o envio de requisições e respostas no formato de mensagem *HTTP* em rede. Portanto, o objetivo principal era tentar compreender de forma mais clara, como ocorrem os processos de comunicação e transmissão de dados na camada de aplicação e de transporte.

Pela manipulação de estruturas como *threads* não ter se mostrado trivial, considero que a implementação do servidor com fila de tarefas e *threads* ofereceu o maior desafio, juntamente à utilização da ferramenta *Siege* na tentativa de obter boas métricas para demonstrar o comportamento de cada tipo de servidor e seu desempenho bruto.

Com este exemplo didático, podemos observar de forma nítida, a complexidade por trás da transmissão de dados, mesmo que em volume pequeno, que podem ser requisitados de forma constante em quaisquer redes de computadores, sejam estas locais, regionais, ou globais como a *internet*.

Referências

- Speed testing your website performance with Siege: Part One: `https://www.euperia.com/wrote/speed-testing-your-website-with-siege-part-one`;
- Series of posts on HTTP status codes: `https://evertpot.com/http/`;
- Load Testing Web Servers with Siege Benchmarking Tool: `https://www.tecmint.com/load-testing-web-servers-with-siege-benchmarking-tool/`;
- How to install Siege on Ubuntu: `https://linuxhint.com/install-siege-ubuntu/`;
- Material disponível no portal didático na disciplina de Redes de Computadores I.