



Universidade Federal
de São João del-Rei

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
REDES DE COMPUTADORES 2024/1

IMPLEMENTAÇÃO DE UM SERVIDOR WEB COM DIFERENTES TÉCNICAS DE PROGRAMAÇÃO EM SOCKET

Gustavo Henriques da Cunha

São João del-Rei

2024

Lista de Figuras

1	Saída do programa com o tipo de servidor iterativo no terminal.	2
---	---	---

Sumário

1	INTRODUÇÃO	1
1.1	Objetivo	1
1.2	Motivação	1
2	Servidores	1
3	Programa	2
4	Implementação	2
4.1	Módulo Servidor	3
4.2	Módulo Servidor Iterativo	3
4.3	Módulo Servidor com Thread	4
4.4	Módulo Servidor com Fila	4
4.5	Módulo Servidor Concorrente	5
5	Testes	5
6	Conclusão	7
	REFERÊNCIAS	7

1 INTRODUÇÃO

Este é um trabalho prático da disciplina de Redes de Computadores no curso de Ciência da Computação na UFSJ, tendo como docente o professor Rafael Sachetto Oliveira.

1.1 Objetivo

Este trabalho tem como objetivo implementar um servidor *web* em C, utilizando quatro diferentes técnicas de programação em *socket*, com o intuito de compreender as suas diferenças, bem como testar e comparar os seus desempenhos.

1.2 Motivação

O trabalho é importante para praticar os conceitos introduzidos em sala de aula, buscando uma melhor compreensão do funcionamento das camadas de aplicação e transporte, além da programação em *socket*.

2 Servidores

A implementação do programa consiste em um servidor *web* com a possibilidade de escolha entre quatro implementações de diferentes técnicas, que são:

1. Servidor Iterativo, onde cada *socket* é aberto individualmente e a conexão é encerrada após o processamento da mesma.
2. Servidor utilizando *threads*, onde para cada conexão aceita, é criada uma *thread* para processá-la.
3. Servidor utilizando *threads* e fila de tarefas, que simula o modelo produtor-consumidor, onde, após o processo principal aceitar uma conexão, esta é enfileirada em uma fila de tarefas.
4. Servidor Concorrente, que utiliza o *select* para aguardar em todos os *sockets* abertos.

3 Programa

Para utilizar o programa, os arquivos devem ser devidamente compilados. Para compilar o programa, deve ser utilizado o comando `make`.

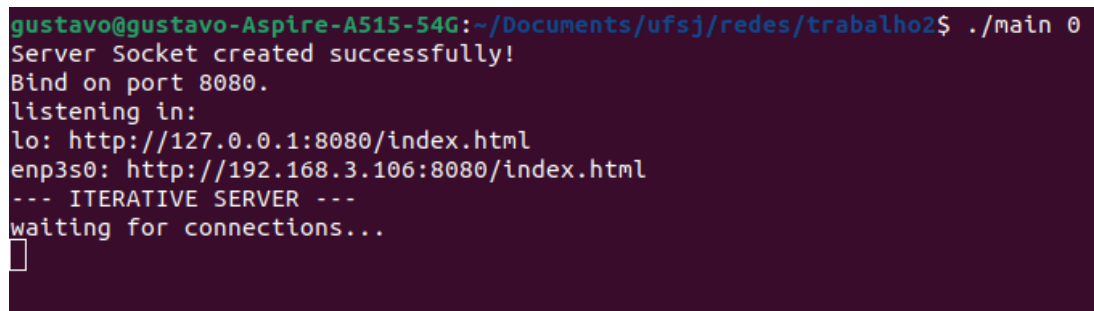
Primeiro, deve-se executar o servidor, juntamente com um número entre 0 e 4 que escolherá o tipo de servidor a ser utilizado, conforme a ordem descrita na secção anterior.

Com o servidor em execução, será exibida no terminal a URL para acesso à página. Basta copiar o endereço e utilizá-lo num *browser* para visualizar o *site*.

O servidor trabalha apenas com requisições do tipo `GET`, ignorando outras requisições. Para aceder ao *site* em `index.html`, apenas as requisições `GET` são necessárias.

A porta utilizada é fixa, sendo o número 8080, definida dentro do código.

A Figura 1 mostra a saída no terminal do servidor em execução com o tipo de servidor iterativo.

A terminal window with a dark purple background. The text is as follows:

```
gustavo@gustavo-Aspire-A515-54G:~/Documents/ufs/j/redes/trabalho2$ ./main 0
Server Socket created successfully!
Bind on port 8080.
listening in:
lo: http://127.0.0.1:8080/index.html
enp3s0: http://192.168.3.106:8080/index.html
--- ITERATIVE SERVER ---
waiting for connections...
█
```

Figura 1: Saída do programa com o tipo de servidor iterativo no terminal.

4 Implementação

O código foi inteiramente implementado em C, modularizado para separar cada tipo de servidor, além de terem sido utilizadas as funções padrão do C.

A estrutura básica do programa consiste num módulo chamado *server*, onde são implementadas as funções básicas do servidor, como a criação do *socket*, aceitação de conexões, leitura de requisições, envio de mensagens e conteúdo. Essas funções são implementadas por todos os tipos de servidores utilizados.

O módulo principal irá receber a indicação de qual servidor será utilizado e chamar a função correspondente.

Em caso de erros, uma mensagem é exibida no terminal e o programa é encerrado.

4.1 Módulo Servidor

O módulo *server* é responsável pela criação dos *sockets* e pela comunicação com o cliente.

Nele, são implementadas as seguintes funções:

- `void error(const char *msg)`: função que receberá uma mensagem de erro, a exibirá e encerrará a execução.
- `int start_socket(int portno, int maxpending)`: função que irá criar e configurar um *socket*, retornando o seu descritor de arquivo, com base no número da porta e no número de requisições que podem ser enfileiradas no *listen*.
- `int accept_connection(int sockfd)`: função que recebe como parâmetro o descritor de arquivo do *socket* do servidor e aceita uma nova conexão, retornando o descritor de arquivo do *socket* do cliente.
- `void connection_handler(int cli_sockfd)`: função responsável por conduzir a conexão com o cliente, cujo descritor de arquivo do *socket* é fornecido.
- `void send_response(int cli_sockfd, const char *code, const char *type, const char *content, size_t content_len)`: função utilizada para enviar uma resposta HTTP ao cliente.
- `char *return_mime(const char *path)`: função utilizada para obter o tipo *MIME* dado o caminho passado.
- `void request_handler(int cli_sockfd, const char *buffer)`: função que irá gerir a requisição feita pelo cliente, verificando se é válida ou não, obter o *MIME* com a função correspondente e enviar os dados, com mensagens de sucesso ou erro, ao cliente, podendo utilizar a função `send_response`. Esta é chamada pela função `connection_handler`.

4.2 Módulo Servidor Iterativo

O módulo *iterative_server* é utilizado para implementar o servidor iterativo, sendo chamado quando o utilizador especifica 0 na execução do programa.

Este módulo possui a seguinte função:

- `int iterative_server(int sockfd)`: ao receber o descritor de arquivo do *socket* do servidor, esta função proporciona todo o funcionamento do servidor iterativo, aceitando um *socket* aberto de cada vez. Ela chama as funções do módulo *server* para tratar da lógica de comunicação com o cliente.

4.3 Módulo Servidor com Thread

O módulo *thread_server* é utilizado para implementar o servidor utilizando *thread*, sendo chamado quando o utilizador especifica 1 na execução do programa.

Este módulo possui as seguintes funções:

- `int thread_server(int sockfd)`: ao receber o descritor de arquivo do *socket* do servidor, esta função proporciona o funcionamento do servidor com *thread* ao utilizar a função de aceitação de conexões do módulo *server* e criar uma *thread* para cada uma delas.
- `void *thread_connection_handler(void *args)`: função passada na criação da *thread* que chama a função de gestão de conexões do módulo *server* para o respetivo *socket*.

4.4 Módulo Servidor com Fila

O módulo *queue_server* é utilizado para implementar o servidor utilizando *threads* e fila de espera, sendo chamado quando o utilizador especifica 2 na execução do programa.

Este módulo mantém, como variáveis globais, a fila de *sockets*, o seu tamanho, as *threads* responsáveis por responder às requisições dos clientes (com um número fixo de 4, definido no código), e as variáveis de controlo para as *threads*, *mutex* e *cond*.

Este módulo possui as seguintes funções:

- `int server_with_queue(int sockfd)`: ao receber o descritor de arquivo do *socket* do servidor, esta função proporciona o funcionamento do servidor com fila de espera, ao alocar as *threads* e a fila, enfileirar os *sockets* ao receber novas conexões, e sinalizar quando a fila não estiver mais vazia.
- `void *queue_thread_connection_handler(void *args)`: função passada na criação da *thread* que chama a função de gestão de conexões do módulo *server* para tra-

tar uma nova requisição e remove o *socket* da fila quando a requisição é concluída, sinalizando quando a fila não estiver mais cheia.

4.5 Módulo Servidor Concorrente

O módulo *concurrent_server* é utilizado para implementar o servidor concorrente, sendo chamado quando o utilizador especifica 3 na execução do programa.

Este módulo possui a seguinte função:

- `int concurrent_server(int sockfd)`: ao receber o descritor de arquivo do *socket* do servidor, esta função proporciona todo o funcionamento do servidor concorrente e, através do *select*, aguarda nos *sockets* abertos.

5 Testes

Para testar a performance de cada servidor, foi utilizado o *software Siege*, que permite simular a interação de múltiplos utilizadores simultâneos, recolhendo dados sobre a performance do servidor.

A Tabela 1 mostra os resultados obtidos a partir de testes realizados com 10 utilizadores simultâneos, 1 repetição e um atraso entre 0 e 10 segundos entre a requisição de cada utilizador. No teste, o Servidor 0 corresponde ao servidor iterativo, o Servidor 1 ao servidor utilizando *threads*, o Servidor 2 ao servidor utilizando fila, e o Servidor 3 ao servidor concorrente.

Parâmetro	Servidor 0	Servidor 1	Servidor 2	Servidor 3
Transações	70	70	70	70
Disponibilidade (%)	100.00	100.00	100.00	100.00
Tempo decorrido (s)	9.11	10.10	10.06	10.13
Dados transferidos (MB)	34.52	34.52	34.52	34.52
Tempo de resposta (s)	0.06	0.02	0.01	0.04
Taxa de transações (trans/s)	7.68	6.93	6.96	6.91
Throughput (MB/s)	3.79	3.42	3.43	3.41
Concorrência	0.43	0.13	0.09	0.31
Transações bem-sucedidas	70	70	70	70
Transações falhadas	0	0	0	0
Transação mais longa (s)	1.02	0.04	0.13	1.02
Transação mais curta (s)	0.00	0.00	0.00	0.00

Tabela 1: Comparação de desempenho entre os servidores.

Podemos observar que os servidores suportaram bem as transações, no sentido de que, em todos os casos, não houve transações falhadas e mantiveram 100% de disponibilidade.

O desempenho entre os servidores não foi muito diferente, com o *Siege* a gastar menos tempo para completar o teste com o servidor iterativo, que apresentou a melhor taxa de transações. Por outro lado, os dois servidores que utilizam *threads* obtiveram um melhor tempo de resposta, um menor valor na sua transação mais longa e um menor valor na concorrência. O servidor iterativo, por sua vez, apresentou um valor elevado na concorrência.

Realizando um teste mais extenso, com 100 utilizadores e execuções de 20 segundos, foi elaborada a Tabela 2.

Parâmetro	Servidor 0	Servidor 1	Servidor 2	Servidor 3
Transações	8277	9038	9748	8394
Disponibilidade (%)	100.00	100.00	100.00	100.00
Tempo decorrido (s)	19.03	19.44	19.71	19.01
Dados transferidos (MB)	4076.03	4452.21	4800.29	4124.67
Tempo de resposta (s)	0.02	0.17	0.02	0.02
Taxa de transações (trans/s)	434.94	464.92	494.57	441.56
Throughput (MB/s)	214.19	229.02	243.55	216.97
Concorrência	6.97	78.93	9.06	7.98
Transações bem-sucedidas	8277	9038	9748	8394
Transações falhadas	0	0	0	0
Transação mais longa (s)	0.08	14.85	6.21	0.05
Transação mais curta (s)	0.00	0.00	0.00	0.00

Tabela 2: Comparação de desempenho entre os servidores com teste maior.

Desta vez, foi possível observar como o servidor com *threads* e o servidor com fila de espera conseguiram atender a um maior número de transações, apresentando ambos, especialmente o servidor com fila, um maior *throughput* quando comparados aos outros servidores que funcionam numa única *thread sequencial*.

Outro aspeto interessante foi o valor de concorrência do Servidor 1, com um valor muito superior aos demais. No entanto, também apresentou um maior tempo de resposta.

No geral, podemos observar melhores resultados no Servidor 2, implementado com fila de espera. Contudo, é importante notar que, em caso de aumento no número de utilizadores, será necessário aumentar o tamanho da fila, ou o servidor poderá sofrer uma grande perda de performance e apresentar problemas.

Neste trabalho, no geral, foi possível verificar um bom desempenho em todos os ser-

vidores implementados. No entanto, é importante compreender que os resultados podem variar com diferentes execuções, sendo necessário um teste mais rigoroso para uma aplicação real.

6 Conclusão

Ao observar os diferentes modos de programação em *socket*, é possível notar a grande importância de cada uma das técnicas, cada qual com as suas devidas aplicações. Saber qual técnica utilizar depende do tipo de problema em mãos, e testes muito mais rigorosos do que os realizados neste trabalho deverão ser efetuados para determinar a melhor opção.

Além disso, é possível compreender a necessidade de ajustar parâmetros, como o número de *threads* e o tamanho da fila, numa aplicação real, onde o número de requisições não é controlado, para que o funcionamento do servidor não escape ao controle.

Por fim, este trabalho foi útil para entender as dificuldades do desenvolvimento de aplicações de rede, compreendendo como o servidor lida com requisições e como isso deve ser considerado aquando da implementação de um sistema.

Referências

- [Cook, 2019] Cook, J. (2019). c send and receive file. <https://stackoverflow.com/questions/11952898/c-send-and-receive-file>.
- [J. KUROSE, 2006] J. KUROSE, K. R. (2006). *Redes de Computadores e a Internet - Uma Nova Abordagem*. Addison-Wesley.
- [Kernighan and Ritchie, 1988] Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall, 2nd edition edition.
- [Lopes, 2011] Lopes, R. (2011). Simples servidor http com concorrência feito em c. <https://www.vivaolinux.com.br/script/Simples-servidor-http-com-concorrencia-feito-em-C/>.
- [Sorber, 2019] Sorber, J. (2019). Sending and handling signals in c (kill, signal, sigaction). <https://www.youtube.com/watch?v=83M5-NPDeWs>.