



Universidade Federal
de São João del-Rei

Análise de diferentes técnicas em uma implementação de servidor WEB

Redes

Professor: Rafael Sachetto Oliveira

Matheus Tavares Elias

Vítor Augusto Niess Soares Fonseca

1. Introdução

Este trabalho aborda a implementação de um servidor *web* simples em linguagem C, utilizando diferentes abordagens para gerenciar múltiplos clientes simultâneos. Foram desenvolvidas quatro versões distintas do servidor, cada uma explorando técnicas diferentes para lidar com solicitações concorrentes.

A primeira implementação emprega um modelo sequencial, onde o servidor processa as requisições dos clientes de maneira linear, aguardando cada solicitação individualmente. Em seguida, foi desenvolvida uma versão utilizando forks, permitindo que o servidor crie processos Filho para manipular cada requisição de forma independente.

A terceira abordagem adotada utiliza threads, permitindo a execução concorrente de várias tarefas dentro do mesmo processo. Para garantir a sincronização adequada entre as threads, foram empregados semáforos e mutex.

A última implementação utiliza o mecanismo de I/O multiplexado com a função *select*, permitindo que o servidor monitore múltiplos sockets para entrada ou saída disponível. Essa abordagem eficiente é especialmente útil para lidar com um grande número de clientes simultâneos.

Além disso, cada implementação foi adaptada para aceitar solicitações específicas, como a requisição de arquivos de texto (*file.txt*) e imagens PNG (*image.png*). Além disso, na implementação que utiliza o *select* foi introduzido um simples sistema de chat para permitir a comunicação entre os clientes conectados, pois foi seguido um exemplo do código fornecido pelo professor que continha essa utilização além de ser a recomendação para quando se utiliza *select*.

Para avaliar o desempenho e a escalabilidade de cada implementação, o *software* “SIEGE” foi utilizado para realizar testes de carga. O “SIEGE” simula múltiplos usuários acessando o servidor simultaneamente, fornecendo métricas valiosas sobre a capacidade de resposta e a estabilidade de cada versão do servidor *web*.

2. Descrição dos servidores

As diferentes implementações de servidores *web* têm suas próprias características e são recomendadas para cenários específicos, dependendo dos requisitos de desempenho e escalabilidade. Veremos uma descrição de cada um dos servidores e uma análise de recomendação para seus diferentes cenários de uso.

2.1. Servidor Iterativo

O servidor atende uma única conexão por vez. Ele aguarda o término do processamento da conexão antes de aceitar a próxima.

É útil para servidores com carga leve a moderada e quando a simplicidade na implementação é prioritária. É fácil de entender e pode ser uma escolha adequada quando a escalabilidade não é uma prioridade.

2.2. Servidor usando *Fork*

Após aceitar uma conexão, um processo filho (usando *fork*) é criado para atender a conexão. O processo pai continua a aceitar novas conexões.

É uma opção intermediária que oferece escalabilidade moderada. Recomendado quando você deseja atender a várias conexões concorrentemente, mas a carga não é muito alta. É mais complexo de implementar do que o servidor iterativo, mas oferece uma melhor capacidade de resposta.

2.3. Servidor usando *Threads* e Fila de Tarefas

Após aceitar uma conexão, o *socket* é inserido em uma fila de tarefas. Um número fixo de threads é responsável por retirar os sockets da fila e processar as solicitações (modelo produtor-consumidor).

É apropriado para cenários em que a concorrência é uma preocupação significativa, e você deseja uma maneira eficiente de atender a várias conexões com um controle rigoroso sobre a concorrência. É especialmente útil quando a carga de trabalho é alta.

2.4. Servidor Concorrente com *Select*

Nesse modelo, você usa a função *select* para aguardar simultaneamente em todos os sockets abertos e acorda o processo somente quando novos dados chegam.

É adequado para situações em que a eficiência de recursos é uma prioridade. É útil para servidores que precisam lidar com um grande número de conexões simultâneas, como servidores de bate-papo ou jogos online. Ele permite que o servidor gerencie várias conexões com recursos mínimos.

Em resumo, a escolha da implementação depende das demandas de desempenho, escalabilidade e eficiência do servidor *web*. Em cenários de baixa carga, a simplicidade de um servidor iterativo pode ser suficiente. Para cargas moderadas a altas, as implementações com *fork*, *threads* e fila de tarefas ou concorrente com *select* podem ser mais apropriadas, dependendo dos requisitos

específicos do projeto. Porém, em cenários reais, o teste de desempenho é fundamental para determinar a melhor implementação para um caso de uso específico.

3. Metodologia

Nesta parte, detalhamos a metodologia utilizada na implementação e teste das quatro versões distintas do servidor *web* em linguagem C, com foco em aspectos técnicos. Cada seção explana sobre as bibliotecas utilizadas, a manipulação de solicitações específicas, e aborda aspectos técnicos relevantes de cada implementação.

3.1. Explicação Detalhada

A seguir, fornecemos uma visão técnica de cada abordagem implementada, destacando aspectos específicos relacionados ao código.

3.1.1. Iterativo

A implementação iterativa utiliza *sockets* da biblioteca *socket.h* e a função *accept* para aguardar a conexão de clientes. O servidor sequencialmente lida com cada solicitação, utilizando funções como *fopen*, *fseek*, e *fread* para processar e responder a requisições de arquivos de texto (*file.txt*) e imagens PNG (*image.png*). O código aguarda a conclusão do processamento de uma conexão antes de aceitar a próxima.

3.1.2. Forks

A versão que utiliza forks emprega a biblioteca *unistd.h* para criar processos filhos independentes para manipular cada conexão. O servidor aceita a solicitação de um cliente, cria um processo filho utilizando *fork*, e delega o processamento ao filho. A comunicação entre processos é realizada por meio de pipes. A implementação suporta requisições de arquivos de texto e imagens PNG, semelhante à abordagem iterativa.

3.1.3. Threads

A implementação com threads utiliza a biblioteca *pthread.h* para criar threads concorrentes. Após aceitar uma conexão, uma nova thread é criada para processar a solicitação. Semáforos (*sem_init*, *sem_wait*, *sem_post*) e mutex (*pthread_mutex_lock*, *pthread_mutex_unlock*) são utilizados para garantir a sincronização adequada entre threads. A manipulação de requisições de arquivos e imagens é semelhante às versões anteriores.

3.1.4. Select

A implementação com select adota a função select para monitorar simultaneamente múltiplos sockets. A abordagem utiliza a biblioteca *sys/select.h*. Além de aceitar solicitações de arquivos e imagens como as abordagens anteriores, ele introduz um sistema de chat simples. A identificação de mensagens de chat é feita por meio da detecção da string "CHAT:" na requisição, seguida da mensagem. O servidor utiliza *send* para transmitir mensagens entre clientes, permitindo a comunicação entre eles.

3.2. Listagem de Rotinas

3.2.1. Iterativo

- *handle_error(const char *message)*: Lida com erros e exibe mensagens.
- *serve_file(int client_socket, const char *file_path, const char *content_type)*: Serve um arquivo específico ao cliente.
- *main()*: Função principal que cria e gerencia o socket do servidor, aceita conexões e processa solicitações.

3.2.2. Forks

- *handle_error(const char *message)*: Lida com erros e exibe mensagens.
- *serve_file(int client_socket, const char *file_path, const char *content_type)*: Serve um arquivo específico ao cliente.
- *handle_client(int client_socket)*: Lida com a solicitação de um cliente.
- *main()*: Função principal que cria e gerencia o socket do servidor, aceita conexões e cria processos filhos para lidar com as solicitações.

3.2.3. Threads

- *serve_file(int client_socket, const char *file_path, const char *content_type)*: Serve um arquivo específico ao cliente.
- *enqueue_task(int client_socket)*: Adiciona uma tarefa à fila de tarefas.
- *dequeue_task()*: Remove uma tarefa da fila de tarefas.
- *handle_client(int client_socket)*: Lida com a solicitação de um cliente.
- *worker_thread(void* arg)*: Função executada por cada thread para processar tarefas.
- *main()*: Função principal que cria e gerencia o socket do servidor, aceita conexões e utiliza threads para lidar com as solicitações.

3.2.4. Select

- *-handle_client(int client_socket, const char *request)*: Lida com a solicitação de um cliente.
- *main()*: Função principal que cria e gerencia o socket do servidor, aceita conexões e utiliza *select* para lidar com as solicitações.
-

4. Resultados e Análise

Após a implementação e descrição detalhada de cada versão do servidor *web*, é crucial analisar os resultados obtidos durante os testes de carga realizados com o *software* “SIEGE”. Os dados coletados são informações valiosas sobre o desempenho, a capacidade de resposta e a estabilidade de cada implementação.

4.1. Testes Práticos

Os testes de carga realizados com o “SIEGE” permitiram avaliar o desempenho geral de cada versão do servidor. As métricas, como tempo médio de resposta, taxa de transferência e número de solicitações atendidas simultaneamente, serão apresentadas e comparadas entre as diferentes implementações. Essas informações são essenciais para entender como cada servidor se comporta sob diferentes cargas de trabalho.

Para realizar os testes de cada um dos algoritmos utilizados, como mencionado, foi usado o *software* “SIEGE”, de forma que cada algoritmo foi testado diversas vezes em quatro etapas. Cada uma dessas etapas consistiu em variar o número de clientes que acessam o servidor concorrentemente. A primeira etapa foi feita com quinze clientes, a segunda com cinquenta, a terceira com cem e a quarta com duzentos e cinquenta e cinco. Todos os resultados serão mostrados em mais detalhes nas seções subsequentes.

Vale ressaltar, que para os testes, foram considerados os tempos gastos pelas requisições mais demoradas, já que elas representam o pior caso possível de cada teste.

4.1.1. Iterativo

Como esperado durante a implementação, o algoritmo iterativo se demonstrou eficiente apenas com um número pequeno de conexões concorrentes. A seguir será mostrado uma tabela com a relação entre o número de clientes acessando e o tempo da requisição mais demorada de cada etapa de testes.

| Clientes/Tempo | Teste 1 | Teste 2 | Teste 3 | Teste 4 |
|----------------|---------|---------|---------|---------|
| 15 | 1,04 | 1,04 | 1,04 | 1,03 |

| | | | | |
|-----|------|------|------|------|
| 50 | 3,46 | 3,05 | 3,25 | 3,05 |
| 100 | 7,84 | 8,09 | 7,76 | 7,53 |
| 255 | 7,73 | 8,82 | 7,83 | 7,80 |

Tabela 1. Teste do Algoritmo Iterativo

De acordo com os dados mostrados na tabela, é possível verificar que o algoritmo em questão não possui uma boa escalabilidade, ou seja, em casos em que o número de clientes cresce muito, ele pode acabar tornando as requisições dos clientes mais lentas do que o esperado. Porém, para aplicações que terão um baixo número de requisições simultâneas, o algoritmo iterativo pode ser uma boa opção, já que ele, apesar do problema anterior, demonstrou manter uma boa estabilidade, mantendo seus tempos de acesso constantes.

4.1.2. Fork

Diferentemente do algoritmo anterior, o algoritmo baseado em *fork*, teoricamente teria um tempo menor para cada requisição, o que se demonstrou verdade após feitos os testes. Novamente, a seguir está mostrado uma tabela com os tempos das requisições mais demoradas em cada etapa de testes.

| Clientes/Tempo | Teste 1 | Teste 2 | Teste 3 | Teste 4 |
|----------------|---------|---------|---------|---------|
| 15 | 1,04 | 1,01 | 0,02 | 0,20 |
| 50 | 3,24 | 3,25 | 3,24 | 3,06 |
| 100 | 4,73 | 7,27 | 3,87 | 4,36 |
| 255 | 7,69 | 7,51 | 7,75 | 8,17 |

Tabela 2. Teste do Algoritmo de Fork

Como dito, foi possível perceber uma melhoria no tempo individual de cada requisição, principalmente ao considerar o aumento no número de clientes. Contudo, em detrimento da escalabilidade, a estabilidade desse algoritmo não se mostrou muito confiável, como evidenciado nos testes com quinze e cem clientes, onde ocorreu uma grande variação entre os tempos de cada requisição.

4.1.3. Thread

O esperado para esse algoritmo é semelhante ao esperado para o algoritmo anterior. Porém, assim como nos casos anteriores, será mostrado uma tabela relacionando o tempo da requisição

mais demorada com o número de clientes acessando o servidor antes de uma discussão sobre o desempenho do algoritmo.

| Clientes/Tempo | Teste 1 | Teste 2 | Teste 3 | Teste 4 |
|----------------|---------|---------|---------|---------|
| 15 | 0,01 | 1,02 | 1,03 | 0,01 |
| 50 | 1,03 | 1,04 | 1,04 | 1,23 |
| 100 | 3,25 | 1,87 | 3,03 | 3,05 |
| 255 | 7,92 | 2,70 | 6,41 | 4,74 |

Tabela 3. Teste do Algoritmo de Threads

Assim como o esperado, o algoritmo baseado em *threads* se demonstrou eficiente ao se considerar a escalabilidade do sistema, mantendo um tempo de resposta razoável à medida que o número de clientes aumenta. Entretanto, da mesma forma que o algoritmo de *fork*, a estabilidade desse algoritmo deixou a desejar, sofrendo grandes variações em alguns dos testes feitos.

4.1.4. Select

O Select é adequado para situações em que a eficiência de recursos é uma prioridade. Ele permite que o servidor gerencie várias conexões com recursos mínimos. No caso do teste não foi possível retirar o máximo desempenho que o algoritmo oferece, pois foi avaliado assim como os anteriores, mas ainda assim é possível perceber uma eficiência maior que os demais.

| Clientes/Tempo | Teste 1 | Teste 2 | Teste 3 | Teste 4 |
|----------------|---------|---------|---------|---------|
| 15 | 0,02 | 0,02 | 0,02 | 0,02 |
| 50 | 1,28 | 1,30 | 1,22 | 1,05 |
| 100 | 1,45 | 1,46 | 2,22 | 1,27 |
| 255 | 3,09 | 3,28 | 3,27 | 3,06 |

Tabela 4. Teste do Algoritmo Select

O algoritmo baseado em *select* apresentou um desempenho mais equilibrado em comparação com os anteriores. Manteve tempos de resposta aceitáveis em diferentes cargas de trabalho e demonstrou boa estabilidade. Embora não tenha alcançado o tempo mais baixo em todos os casos, sua consistência o torna uma opção atraente.

4.1.5. Desempenho Geral

Como uma forma de representação visual dos resultados obtidos, foi desenvolvido um gráfico indicando o contraste entre o tempo de requisição mais lenta entre todos os algoritmos. Tal gráfico é mostrado a seguir.

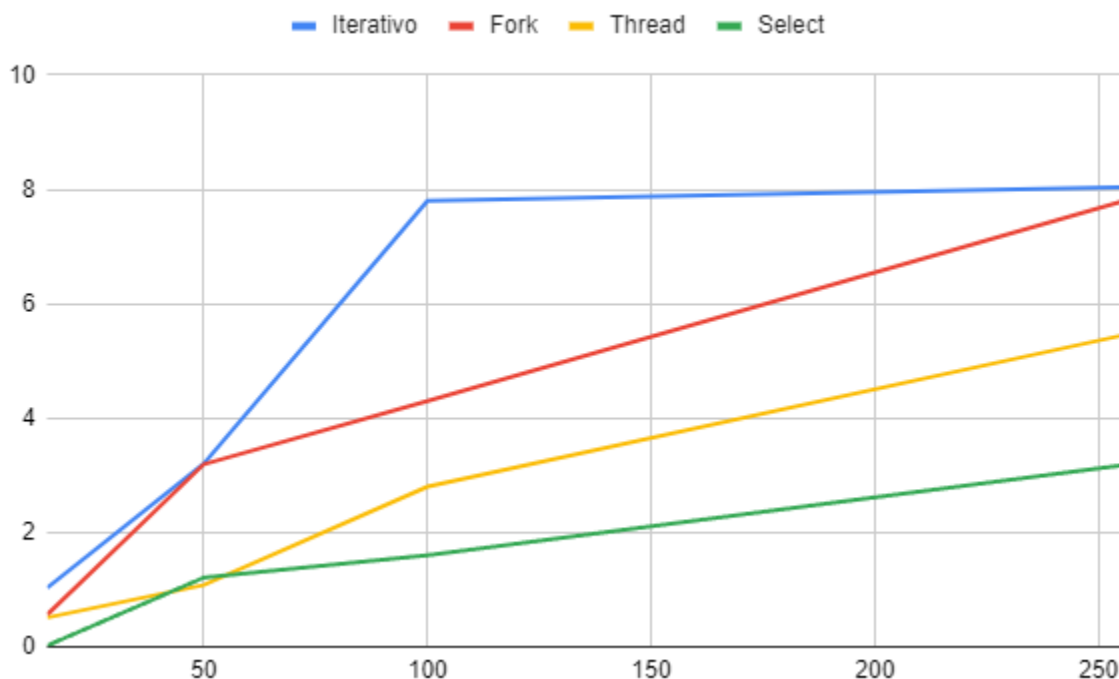


Figura 1. Gráfico da Média de Tempo dos Algoritmos

5. Conclusão

Com base nos resultados dos testes de carga e na análise de desempenho foi possível identificar que cada algoritmo possui seus pontos fortes e fracos, e a escolha entre eles dependerá das necessidades específicas do sistema.

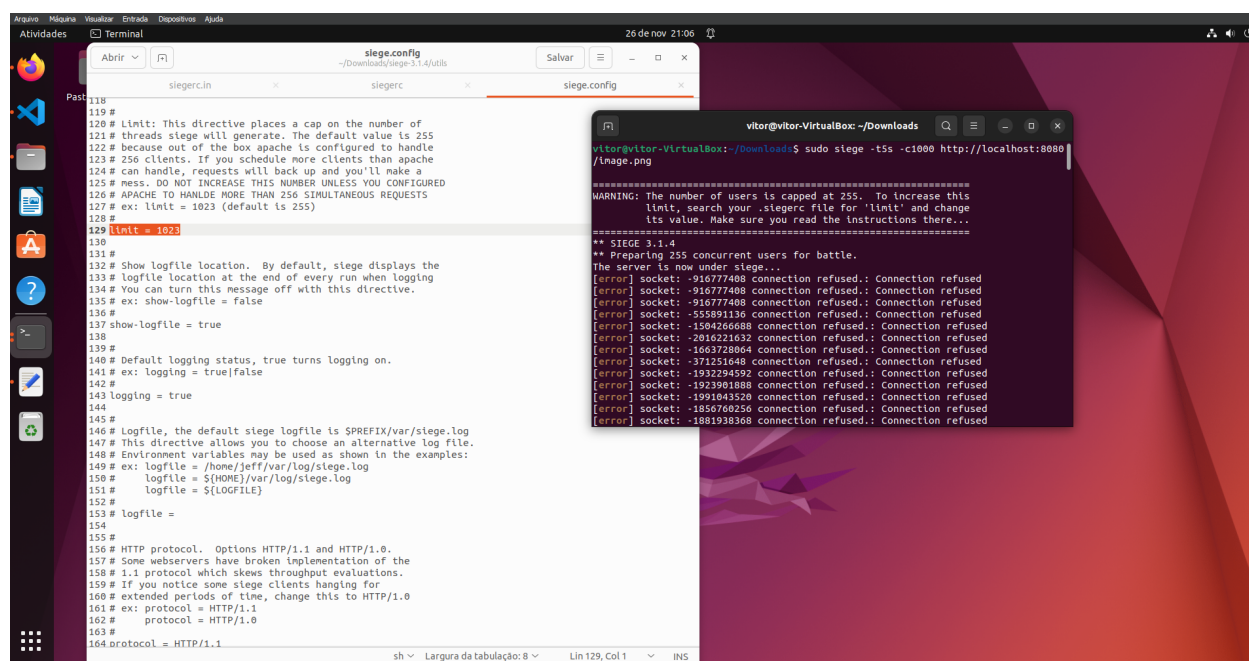
O algoritmo iterativo mostrou-se eficiente apenas para um número pequeno de conexões concorrentes. Sua escalabilidade é limitada, e o tempo de resposta tende a aumentar significativamente com um número crescente de clientes. No entanto, ele demonstrou estabilidade em situações de carga mais leve, mantendo tempos de acesso relativamente constantes.

Já o algoritmo baseado em *fork* apresentou uma melhoria no tempo individual de cada requisição, especialmente quando o número de clientes aumentou. No entanto, a escalabilidade não é ideal, e a estabilidade do sistema foi comprometida em alguns testes, evidenciando variações nos tempos de acesso.

Semelhante ao algoritmo de *fork*, o algoritmo baseado em *threads* mostrou eficiência em termos de escalabilidade. Manteve um tempo de resposta razoável conforme o número de clientes aumentou. Todavia, também enfrentou problemas de estabilidade, com variações significativas nos tempos de acesso em alguns cenários.

O último algoritmo testado, o *select*, mostrou ter um equilíbrio entre desempenho e estabilidade, destacando-se como uma opção viável em diversos cenários, manteve tempos de resposta aceitáveis em diferentes cargas de trabalho e demonstrou boa estabilidade. No entanto, a seleção final deve considerar fatores como a natureza da aplicação, a carga prevista e os recursos disponíveis.

Por fim, é importante dizer que a limitação de testes com um número maior de clientes se deu devido às restrições do software “SIEGE” e “LOCUST”, sendo que não foi possível alterar o limite superior de clientes do “SIEGE” (255 simultâneos) e o uso do “LOCUST” se demonstrou inviável por apenas aceitar 10000 ou mais clientes, o que requer uma alteração no próprio sistema operacional Ubuntu, que limita essa quantidade em 1024. Ambas as alterações se demonstraram complexas ou, como no caso do “SIEGE”, apenas não funcionavam. A seguir seguem alguns *prints* evidenciando os problemas mencionados.

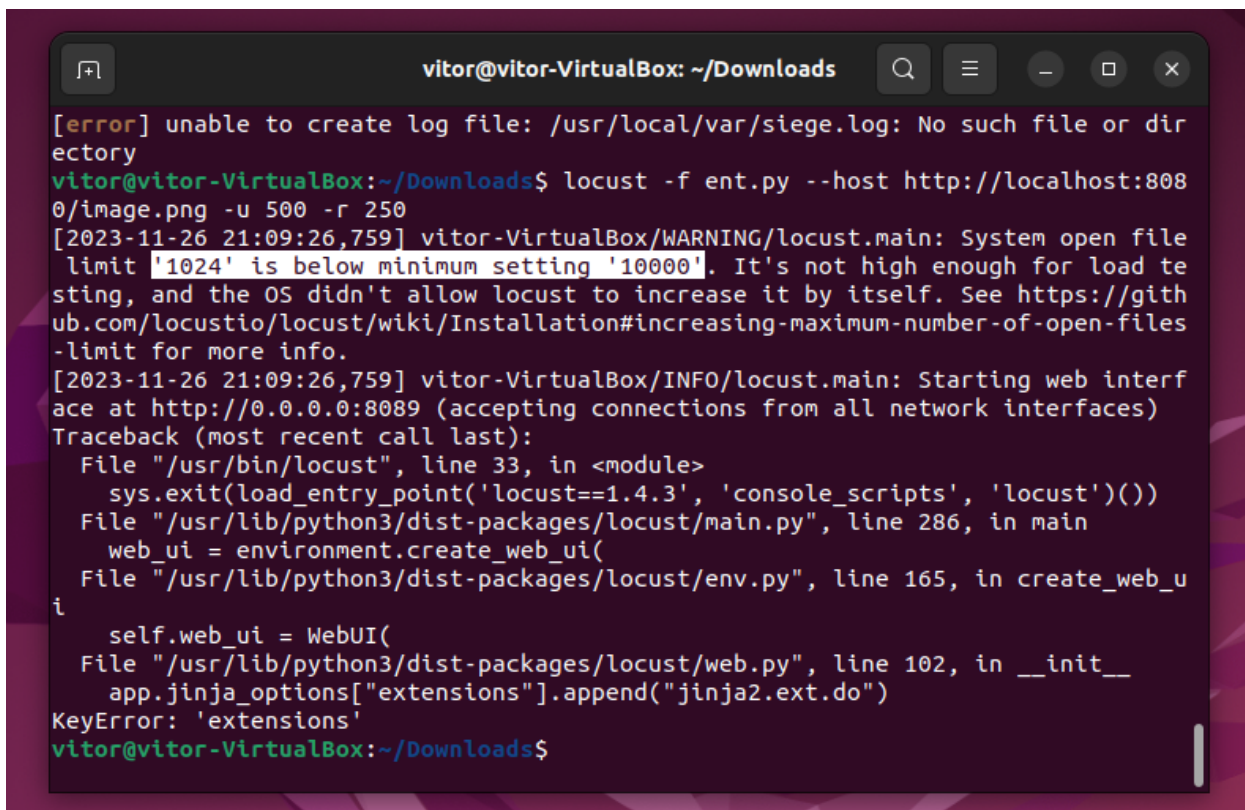


```
118 #
119 #
120 # Limit: This directive places a cap on the number of
121 # threads siege will generate. The default value is 255
122 # because out of the box apache is configured to handle
123 # 256 clients. If you schedule more clients than apache
124 # can handle, requests will back up and you'll make a
125 # mess. DO NOT INCREASE THIS NUMBER UNLESS YOU CONFIGURE
126 # APACHE TO HANDLE MORE THAN 256 SIMULTANEOUS REQUESTS
127 # ex: limit = 1023 (default is 255)
128 #
129 limit = 1024
130 #
131 #
132 # Show logfile location. By default, siege displays the
133 # logfile location at the end of every run when logging
134 # You can turn this message off with this directive.
135 # ex: show-logfile = false
136 #
137 show-logfile = true
138 #
139 #
140 # Default logging status, true turns logging on.
141 # ex: logging = true/false
142 #
143 logging = true
144 #
145 #
146 # Logfile, the default siege logfile is $PREFIX/var/siege.log
147 # This directive allows you to choose an alternative log file.
148 # Environment variables may be used as shown in the examples:
149 # ex: logfile = /home/jeff/var/log/siege.log
150 #      logfile = $(HOME)/var/log/siege.log
151 #      logfile = ${LOGFILE}
152 #
153 logfile =
154 #
155 #
156 # HTTP protocol. Options HTTP/1.1 and HTTP/1.0.
157 # Some webservers have broken implementation of the
158 # 1.1 protocol which skews throughput evaluations.
159 # If you notice some siege clients hanging for
160 # extended periods of time, change this to HTTP/1.0
161 # ex: protocol = HTTP/1.1
162 #      protocol = HTTP/1.0
163 #
164 protocol = HTTP/1.1
```

```
vitor@vitor-VirtualBox: ~/Downloads$ sudo siege -t5s -c1000 http://localhost:8080/image.png
*****
WARNING: The number of users is capped at 255. To increase this
limit, search your .siegerc file for 'limit' and change
its value. Make sure you read the instructions there...
*****
** SIEGE 3.1.4
** Preparing 255 concurrent users for battle.
The server is now under siege...
[error] socket: -916777408 connection refused.: Connection refused
[error] socket: -916777408 connection refused.: Connection refused
[error] socket: -555891136 connection refused.: Connection refused
[error] socket: -1584266688 connection refused.: Connection refused
[error] socket: -2016221632 connection refused.: Connection refused
[error] socket: -1663728064 connection refused.: Connection refused
[error] socket: -371231648 connection refused.: Connection refused
[error] socket: -1932294592 connection refused.: Connection refused
[error] socket: -1923981888 connection refused.: Connection refused
[error] socket: -1991043520 connection refused.: Connection refused
[error] socket: -1856760256 connection refused.: Connection refused
[error] socket: -1881938368 connection refused.: Connection refused
```

Figura 2. *Print* mostrando problema no “SIEGE”

Nesse *print* é possível ver a mensagem de alerta do “SIEGE” informando que o limite de clientes é de 255, porém, ao lado, está o arquivo de configuração do mesmo, com o limite em 1024.

A terminal window titled 'vitor@vitor-VirtualBox: ~/Downloads' with standard window controls. The terminal output shows an initial error about a missing log file, followed by the execution of the 'locust' command with specific flags. A warning message indicates that the system's open file limit (1024) is below the required minimum (10000). The application then starts a web interface. Finally, a traceback is shown, leading to a 'KeyError: 'extensions'' in the 'locust/web.py' file.

```
[error] unable to create log file: /usr/local/var/siege.log: No such file or directory
vitor@vitor-VirtualBox:~/Downloads$ locust -f ent.py --host http://localhost:8080/image.png -u 500 -r 250
[2023-11-26 21:09:26,759] vitor-VirtualBox/WARNING/locust.main: System open file limit '1024' is below minimum setting '10000'. It's not high enough for load testing, and the OS didn't allow locust to increase it by itself. See https://github.com/locustio/locust/wiki/Installation#increasing-maximum-number-of-open-files -limit for more info.
[2023-11-26 21:09:26,759] vitor-VirtualBox/INFO/locust.main: Starting web interface at http://0.0.0.0:8089 (accepting connections from all network interfaces)
Traceback (most recent call last):
  File "/usr/bin/locust", line 33, in <module>
    sys.exit(load_entry_point('locust==1.4.3', 'console_scripts', 'locust')())
  File "/usr/lib/python3/dist-packages/locust/main.py", line 286, in main
    web_ui = environment.create_web_ui()
  File "/usr/lib/python3/dist-packages/locust/env.py", line 165, in create_web_ui
    self.web_ui = WebUI(
  File "/usr/lib/python3/dist-packages/locust/web.py", line 102, in __init__
    app.jinja_options["extensions"].append("jinja2.ext.do")
KeyError: 'extensions'
vitor@vitor-VirtualBox:~/Downloads$
```

Figura 3. *Print* mostrando problema no “LOCUST”

Já neste, é mostrado a mensagem de erro do “LOCUST”, informando que o limite de arquivos abertos simultaneamente do sistema operacional (1024) é inferior ao mínimo requerido pelo aplicativo (10000).

Após todos os problemas encontrados, fica evidente a importância de se considerar não apenas o desempenho intrínseco dos algoritmos, mas também as ferramentas disponíveis para a realização dos testes de carga. A busca por soluções para essas limitações pode ser um ponto crucial para futuras melhorias no processo de avaliação de desempenho.

Em resumo, a análise de desempenho proporcionou informações valiosas, destacando a necessidade contínua de refinamento e otimização para atender às demandas variáveis dos ambientes de servidor *web*.

Referências Bibliográficas

1. Kurose, J. F., & Ross, K. W. (2013). Redes de Computadores e a Internet: Uma Abordagem Top-Down. Editora.
2. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.).
Acessado em 19/11/2023
<https://os.ecci.ucr.ac.cr/slides/Abraham-Silberschatz-Operating-System-Concepts-10th-2018.pdf>