REDES DE COMPUTADORES
RELATÓRIO DO TRABALHO PRÁTICO - SERVIDOR WEB - TÉCNICAS DISTINTAS

João Antônio Lucas Costa Lucas Emanuel

Janeiro de 2025

1 Servidor iterativo

1.1 Introdução

A primeira técnica utilizada para a implementação do servidor web foi o servidor iterativo. O código implementa um servidor TCP utilizando a API Winsock em C, que é uma interface para comunicação em redes no sistema operacional Windows. O servidor escuta conexões na porta 8080 e, ao aceitar uma conexão, responde com uma mensagem HTML básica. A estrutura do programa é composta por etapas como a inicialização do Winsock, a criação e configuração do socket, e a escuta e aceitação de conexões de clientes. Após estabelecer uma conexão, o servidor recebe uma solicitação do cliente e envia uma resposta de sucesso, indicando que o servidor está operando corretamente. Este relatório detalha a implementação, funcionamento e considerações do código, fornecendo uma visão abrangente do processo de comunicação em rede.

1.2 Listagem das Rotinas

O código gerencia a comunicação em redes no sistema operacional Windows. A função `initialize_ winsock` é responsável por iniciar a API Winsock, preenchendo a estrutura `WSADATA` com informações sobre a versão utilizada. Caso a inicialização seja bem-sucedida, a função retorna 1; caso contrário, retorna 0. Em seguida, a função `create_socket` cria um socket TCP para o servidor. Se a criação do socket for bem-sucedida, a função retorna o socket criado; se houver um erro, retorna `INVALID_SOCKET`.

A função `configure_server_address` configura a estrutura de endereço do servidor, definindo o tipo de família do endereço como IPv4, o endereço IP como `INADDR_ANY`, que permite aceitar conexões de qualquer interface de rede, e a porta de escuta, definida pela constante `PORT`. Essa função sempre retorna 1, indicando que a configuração foi realizada com sucesso. Posteriormente, a função `bind_socket` realiza o *binding* do socket à estrutura de endereço do servidor, permitindo que o servidor escute conexões na porta especificada. Essa função retorna 1 em caso de sucesso ou 0 se ocorrer um erro durante o *binding*.

Após o *binding*, a função `start_listening` coloca o socket em modo de escuta, permitindo que ele aceite conexões de clientes. Se a operação for bem-sucedida, a função retorna 1; em caso de erro, retorna 0. Uma vez que o servidor está escutando, a função

`send_response` é utilizada para enviar uma resposta HTTP simples ao cliente, informando que a requisição foi bem-sucedida. Esta função não retorna nenhum valor, mas envia a resposta diretamente pelo socket do cliente.

A função principal `main` coordena todo o processo. Ela inicia a API Winsock, cria o socket TCP, configura o endereço do servidor, realiza o *binding* e inicia a escuta de conexões. Em um loop contínuo, a função aceita conexões de clientes, recebe solicitações e envia respostas. Após o tratamento de todas as conexões, os recursos são limpos e o programa é encerrado. Essa implementação serve como um exemplo básico de comunicação em rede, ilustrando os passos essenciais para a configuração de um servidor TCP.

1.3 Descrição dos Algoritmos e Estruturas de Dados Utilizadas

A estrutura de dados primária utilizada é a `struct sockaddr_in`, que faz parte da biblioteca de sockets e serve para definir o endereço do servidor e do cliente. Esta estrutura contém informações sobre a família do endereço (neste caso, IPv4), o endereço IP e o número da porta. Ela permite que o servidor identifique a origem das conexões e as configure adequadamente.

O algoritmo principal do servidor segue um padrão iterativo para gerenciar conexões de clientes. Após a inicialização do Winsock, o código cria um socket utilizando a função `socket`, que retorna um descritor de socket. O algoritmo configura o endereço do servidor e realiza o *binding* do socket à porta especificada. Uma vez que o servidor está em modo de escuta, ele entra em um loop onde utiliza a função `accept` para esperar e aceitar conexões de clientes. Este é um ponto crítico, onde o servidor aguarda ativamente por novas conexões.

Quando uma conexão é aceita, o servidor utiliza um buffer de caracteres (`char buffer[BUFFER_SIZE]`) para armazenar os dados recebidos da solicitação do cliente. O tamanho do buffer é definido por `BUFFER_SIZE`, que é uma constante, garantindo que o servidor possa lidar com mensagens de um tamanho máximo específico. A função `recv` é utilizada para ler a solicitação do cliente e armazena os dados nesse buffer.

Em resposta, o servidor utiliza a função `send` para enviar uma mensagem HTTP básica ao cliente, confirmando que a requisição foi bem-sucedida. O algoritmo não utiliza estruturas de dados complexas, mas emprega essas operações básicas de sockets e manipulação de strings para comunicação em rede. A abordagem iterativa é simples, mas eficaz, permitindo que o servidor aceite uma conexão, processe a solicitação e responda, tudo dentro de um loop contínuo.

1.4 Análise dos Resultados Obtidos

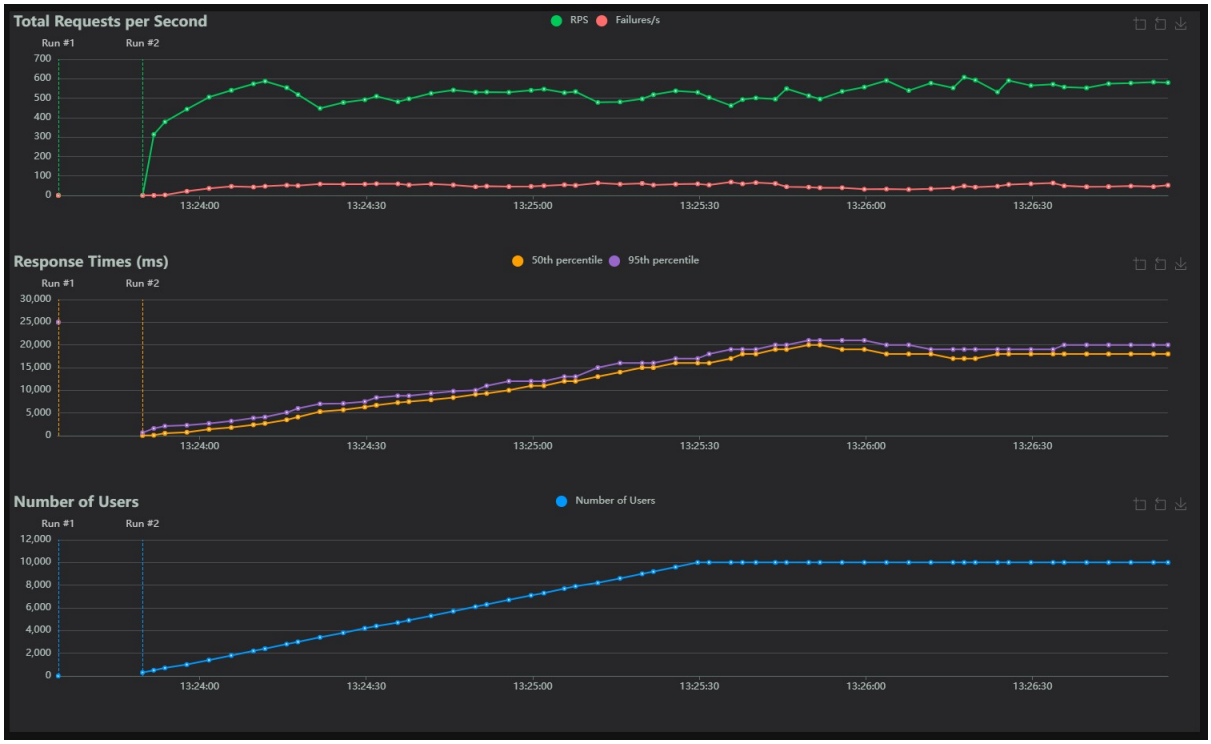

Figura 1: Resultados dos testes - Servidor iterativo

Gráfico 1: Total Requests per Second

- **RPS (verde)**: Representa o número de requisições por segundo que o servidor conseguiu processar. No início, há um aumento no RPS, indicando que o servidor está respondendo bem ao aumento de carga. O RPS estabiliza em torno de 600 requisições por segundo, mostrando a capacidade máxima do servidor.
- **Failures/s (vermelho)**: Número de requisições que falharam por segundo. Este valor é baixo e constante, indicando que a maioria das requisições foi processada com sucesso.

Gráfico 2: Response Times (ms)

- **50th Percentile (laranja)**: Tempo de resposta mediano (50% das requisições são atendidas nesse tempo ou menos).
- **95th Percentile (roxo)**: Tempo de resposta para 95% das requisições.
- Ambos os tempos aumentam conforme o número de usuários cresce, mostrando que o servidor demora mais para responder sob alta carga.

Gráfico 3: Número de Usuários

- Mostra o número de usuários simultâneos durante o teste. O número cresce linearmente, simulando um aumento gradual na carga.

Type	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)
GET	99122	9375	16000	12645.84	8	21924	17.2
Aggregated	99122	9375	16000	12645.84	8	21924	17.2

Tabela 1: Tabela de Desempenho

Os resultados do servidor indicam que, de um total de 99.122 requisições, houve 9.375 falhas, resultando em uma taxa de erro de aproximadamente 9,5%. A mediana do tempo de resposta foi de 16.000 milissegundos, com 95% das requisições atendidas em até 20 segundos e uma média de 12.645,84 milissegundos. O servidor atingiu uma taxa média de 17,2 requisições por segundo, mas apresentou uma alta taxa de falhas de 52,2 por segundo. Esses resultados sugerem que, apesar da capacidade de processar um grande volume de requisições, o servidor enfrenta problemas de latência e robustez, indicando a necessidade de otimizações para melhorar a performance e reduzir as falhas.

2 Servidor com Threads

2.1 Introdução

O objetivo do servidor é que ele possa gerenciar várias conexões de clientes simultaneamente, proporcionando um ambiente eficiente e responsivo para atender a solicitações HTTP. O servidor escuta na porta 8080 e responde com uma mensagem HTML simples para cada solicitação recebida.

A arquitetura dele é baseada em threads, onde cada conexão de cliente é tratada em uma nova thread. Isso permite que o servidor processe múltiplas requisições simultaneamente, melhorando a sua escalabilidade e capacidade de atendimento. O código inclui funções para inicializar o Winsock, criar e configurar um socket, realizar o *binding* do socket a um endereço IP e porta, e aceitar conexões de clientes.

Ao longo deste relatório, será apresentada uma análise detalhada das principais funcionalidades do código, bem como a descrição dos algoritmos e estruturas de dados utilizadas, além da avaliação dos resultados obtidos durante os testes de desempenho do servidor.

2.2 Listagem das Rotinas

A função `handle_client` é executada em uma nova *thread* para lidar com a conexão de um cliente. Ela recebe a solicitação do cliente através do *socket* fornecido, envia uma resposta HTTP simples com a mensagem `"Hello World"` e, em seguida, fecha a conexão com o cliente. Para garantir que o *socket* do cliente permaneça acessível dentro da *thread*, ele é passado como um argumento alocado dinamicamente.

A função `initialize_winsock` inicializa a API Winsock, preenchendo a estrutura `WSADATA` com informações sobre a versão do Winsock utilizada. Ela retorna 1 se a inicialização for bem-sucedida e 0 em caso de erro. A rotina `create_socket` é responsável por criar um *socket* do tipo TCP. Se a criação do *socket* falhar, a função exibe uma mensagem de erro e retorna `INVALID_SOCKET`; caso contrário, retorna o descritor do *socket* criado.

A função `configure_server_address` configura os parâmetros da estrutura `sockaddr_in` para o servidor, definindo a família do endereço como IPv4, o endereço IP como `INADDR_ANY` (aceitando conexões em qualquer interface) e a porta como 8080. A rotina

`bind_socket` realiza o *binding* do *socket* a um endereço e porta específicos, retornando 1 em caso de sucesso e 0 em caso de erro durante a operação de *binding*.

A função `start_listening` coloca o *socket* em modo de escuta, permitindo que ele aceite conexões de clientes. Ela retorna 1 se a operação for bem-sucedida e 0 se ocorrer um erro. Por fim, a função `main` coordena a execução do servidor, inicializando o Winsock, criando e configurando o *socket*, realizando o *binding* e começando a escutar por conexões. Em um *loop* contínuo, aceita conexões de clientes e cria uma nova *thread* para cada conexão, delegando o tratamento à função `handle_client`.

2.3 Descrição dos Algoritmos e Estruturas de Dados Utilizadas

A principal estrutura de dados utilizada é a `struct sockaddr_in`, que é uma estrutura da API de *sockets* que armazena informações sobre o endereço do servidor, incluindo a família do endereço (`AF_INET` para IPv4), o endereço IP e a porta de escuta. Essa estrutura é crucial para a configuração do *socket* e o *binding* a um endereço específico.

Os algoritmos implementados no código incluem a inicialização do Winsock, a criação do *socket* e a configuração do endereço do servidor, que são realizados sequencialmente no início da execução do programa. A função `listen` é empregada para habilitar o *socket* a aceitar conexões, enquanto a função `accept` é utilizada para aguardar e aceitar uma nova conexão de cliente. Para cada conexão aceita, o algoritmo cria uma nova *thread* usando a função `CreateThread`, passando o *socket* do cliente como argumento. Isso permite que cada conexão seja tratada de forma independente, garantindo que o servidor possa atender a múltiplos clientes simultaneamente.

Além disso, o uso da alocação dinâmica de memória (com `malloc`) para armazenar o *socket* do cliente é um aspecto importante que permite que cada *thread* tenha sua própria cópia do *socket*. Essa abordagem é fundamental para evitar conflitos entre *threads* e garantir que cada uma possa operar de maneira autônoma.

2.4 Análise dos Resultados Obtidos

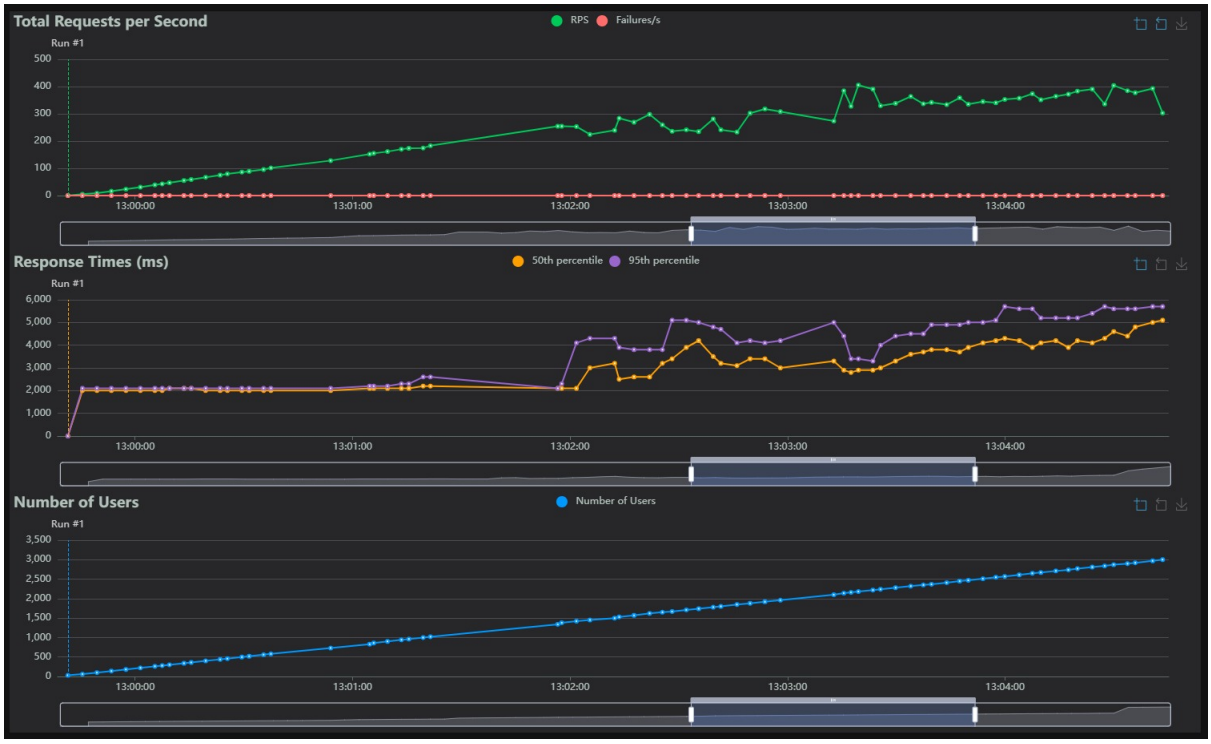

Figura 2: Resultados dos testes - Servidor com threads

- **Gráfico 1: Total Requests per Second**
  - O RPS (linha verde) aumenta gradualmente conforme mais usuários (*threads*) são adicionados ao teste. A linha vermelha indica a taxa de falhas, que permanece praticamente zero, sugerindo que o servidor está lidando bem com a carga durante o teste.
- **Gráfico 2: Response Times (ms)**
  - O tempo de resposta no percentil 50 (tempo médio) é relativamente estável, mas apresenta aumentos graduais ao longo do teste. O tempo de resposta no percentil 95 é mais irregular e aumenta consistentemente à medida que o número de usuários cresce, sugerindo maior latência sob alta carga.
- **Gráfico 3: Number of Users**
  - O número de usuários (linha azul) aumenta linearmente ao longo do tempo, conforme o esperado em um teste de carga controlado.

Type	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)
GET	157027	1728	4900	5145.07	2011	14888	18.79
Aggregated	157027	1728	4900	5145.07	2011	14888	18.79

Tabela 2: Tabela de Desempenho

O servidor processou um total de 157.027 requisições com 1.728 falhas (taxa de erro de aproximadamente 1,1%). O tempo de resposta mediano foi de 4.900 milissegundos,

com percentis de 95 e 99 apresentando tempos de 9.200 e 12.000 milissegundos, respectivamente. O servidor apresentou uma taxa de requisições por segundo de 279,1 e uma latência variando de 2.011 a 14.888 milissegundos. A análise destaca a necessidade de otimizações para reduzir a latência e aumentar a confiabilidade, considerando a quantidade de falhas observadas.

3 Servidor utilizando threads e fila de tarefas

A terceira técnica implementada neste servidor web é o modelo produtor-consumidor utilizando `threads e uma fila de tarefas`. Nesse modelo, o processo principal do servidor é responsável por aceitar as conexões dos clientes e, em seguida, enfileirar os descritores de socket em uma fila compartilhada. Um número fixo de threads é criado para consumir as tarefas enfileiradas e processá-las de maneira assíncrona. Cada thread retira uma tarefa da fila, processa a requisição HTTP correspondente e envia a resposta ao cliente. Esse modelo é eficaz para balancear a carga de trabalho e melhorar o desempenho do servidor, já que múltiplas threads podem processar as requisições simultaneamente, evitando que o servidor se sobrecarregue ao lidar com muitas conexões de forma sequencial.

3.1 Listagem de Rotinas

A função `enqueue()` é responsável por inserir uma tarefa na fila de tarefas compartilhada. O acesso à fila é protegido por um `mutex` (`queue_mutex`) para garantir que somente uma thread possa modificá-la de cada vez, prevenindo condições de corrida. Quando uma tarefa é enfileirada, a função primeiro verifica se a fila está cheia. Isso é feito verificando se a posição seguinte de `queue_end` é igual a `queue_start`, o que indicaria que não há espaço disponível. Caso a fila esteja cheia, a função imprime uma mensagem de erro. Se a fila não estiver cheia, a tarefa é inserida na posição indicada por `queue_end`, e a variável `queue_end` é atualizada para apontar para a próxima posição da fila, seguindo uma abordagem circular. A função, então, sinaliza a variável de condição `queue_cond`, indicando que há uma nova tarefa na fila e que uma thread pode ser desbloqueada para processá-la. Após a inserção, o `mutex` é liberado para permitir que outras threads possam acessar a fila.

A função `dequeue()` retira uma tarefa da fila compartilhada. Ela também é protegida pelo `mutex` (`queue_mutex`), garantindo que apenas uma thread possa acessar a fila de cada vez. Se a fila estiver vazia (quando `queue_start` é igual a `queue_end`), a função entra em um estado de espera, aguardando que uma tarefa seja inserida na fila. Isso é feito com a utilização da variável de condição `queue_cond`, que é sinalizada quando uma nova tarefa é enfileirada. Assim que uma tarefa está disponível, ela é retirada da posição indicada por `queue_start`, e o índice `queue_start` é atualizado de forma circular, garantindo que a fila opere de forma eficiente. Após a remoção da tarefa, o `mutex` é liberado, permitindo que outras threads possam acessar a fila. A tarefa retirada da fila é então retornada para ser processada.

A função `process_request()` é responsável por processar uma requisição HTTP recebida de um cliente. A função recebe o socket do cliente como parâmetro, o qual é utilizado para ler e escrever dados. Primeiro, a função lê os dados da requisição utilizando o comando `read()`, que armazena os dados no buffer fornecido. Caso a leitura seja bem-sucedida (i.e., `bytes_read > 0`), a função envia uma resposta HTTP simples, com um código de status 200 OK e um corpo contendo um HTML básico com a mensagem

"Olá, Mundo!". A resposta é enviada ao cliente através do comando `write()`, que escreve no socket do cliente. Após o envio da resposta, a conexão com o cliente é fechada com o comando `close()`, liberando os recursos associados ao socket.

A função `worker_thread()` é a rotina executada pelas threads de trabalho do servidor. Cada thread, em um loop contínuo, chama a função `dequeue()` para retirar uma tarefa da fila. Uma vez que uma tarefa é retirada, a função `process_request()` é chamada, passando o socket do cliente como parâmetro, para processar a requisição HTTP associada àquela tarefa. Como as threads ficam em um loop infinito, elas continuam a processar tarefas enquanto o servidor estiver em funcionamento. Isso permite que múltiplas threads operem de maneira concorrente, processando as requisições dos clientes de forma paralela. O uso de múltiplas threads melhora a escalabilidade e a eficiência do servidor, permitindo que ele atenda várias requisições simultaneamente.

A função `setup_server()` é responsável pela configuração inicial do servidor. Ela realiza a criação do socket de rede através da chamada `socket()`, especificando que o socket será do tipo TCP/IP (`SOCK_STREAM`). Em seguida, a função configura o endereço do servidor, criando uma estrutura `sockaddr_in` e inicializando seus campos, incluindo o endereço IP (`INADDR_ANY`, que permite que o servidor aceite conexões de qualquer endereço) e a porta de escuta (`PORT`, definida como 8080). O servidor então associa o socket ao endereço configurado com a chamada `bind()`, o que permite que o servidor ouça conexões na porta especificada. Após isso, a função configura o socket para escutar por conexões de clientes utilizando a chamada `listen()`, com uma fila de espera de tamanho 5 para conexões pendentes. A função retorna o descritor de arquivo do servidor, que será usado posteriormente para aceitar as conexões.

3.2 Análise dos Resultados Obtidos

Os testes foram realizados utilizando o `Locust` [Loc], simulando um total de **10.000 usuários** com uma taxa de incremento de **100 usuários por segundo**, durante **5 minutos** de execução. O servidor foi configurado com **4 threads** para processamento paralelo e uma fila limitada a **10 tarefas** (modelo produtor-consumidor). O teste consistiu em uma requisição e resposta simples, com o servidor retornando uma página `HTML` básica, o que pode influenciar os resultados, uma vez que cenários mais complexos poderiam gerar maior latência. Os resultados mostraram uma taxa estável de **1.190 requisições por segundo (RPS)** ao final do teste, sem falhas. O tempo de resposta mediano foi baixo, **33 ms**, mas os tempos nos percentis superiores ($95^{\underline{o}}$ e $99^{\underline{o}}$) atingiram **17 e 20 segundos**, indicando que algumas requisições enfrentaram alta latência devido a gargalos na fila ou no processamento. Conclui-se que o servidor é robusto para cargas moderadas em operações simples, mas melhorias no número de `threads` e no tamanho da fila podem aumentar sua eficiência para lidar com picos de carga e cenários mais complexos.

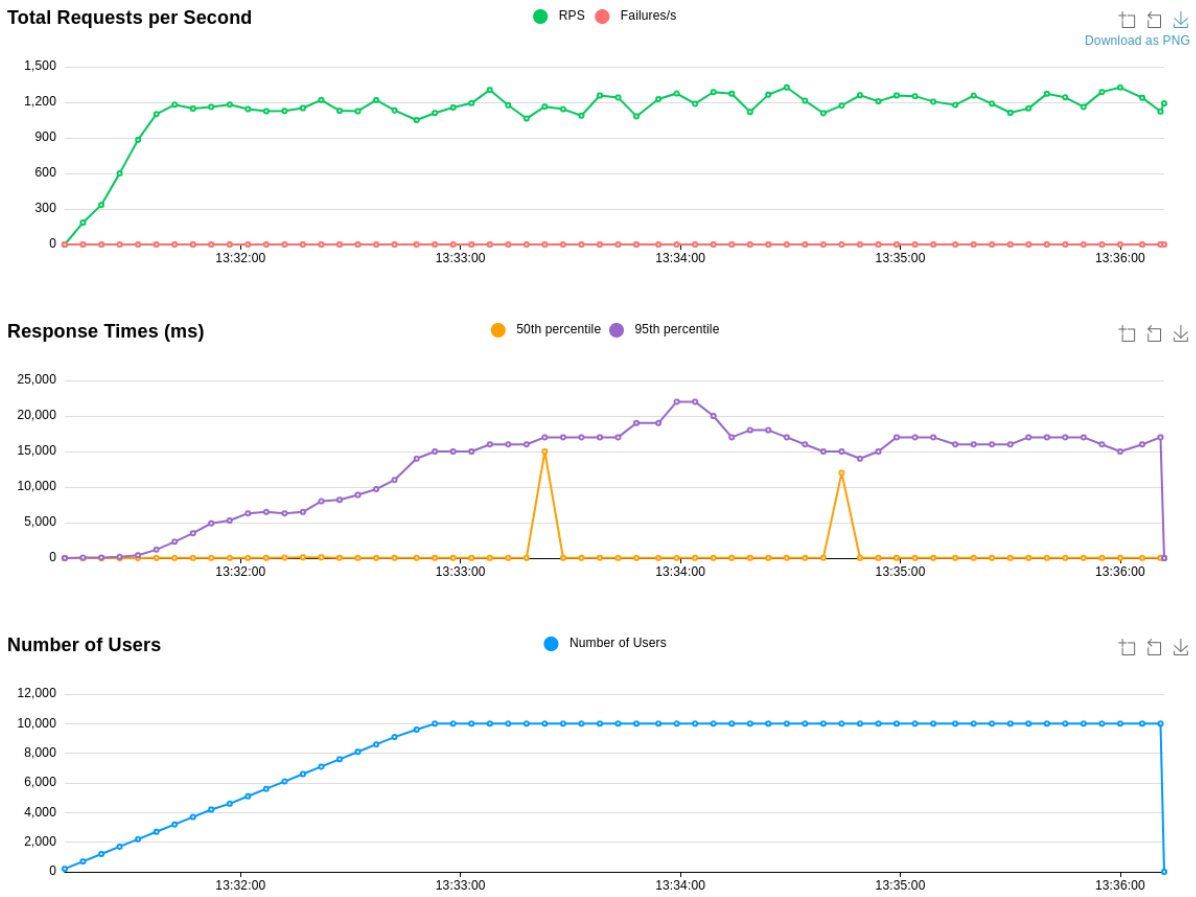

Figura 3: Resultados dos testes com fila de tarefas

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Current RPS
GET	/	347128	0	33	4780.59	1190.2
Aggregated	-	347128	0	33	4780.59	1190.2

Tabela 3: Resultados dos testes de desempenho com Locust.

4 Servidor Concorrente

4.1 Introdução

Na comunicação em rede, os servidores concorrentes têm um papel extremamente importante quando se trata da simultaneidade eficiente de múltiplas conexões. Uma abordagem de implementação muito utilizada nesses servidores é o uso do select (como o código disponibilizado no SIGAA). Essa técnica permite que o servidor monitore múltiplos sockets abertos ao mesmo tempo, aguardando de forma eficiente pela chegada de novos dados, sem bloquear desnecessariamente o processo.

O select funciona como um mecanismo que coloca o processo em estado de espera até que haja atividade em pelo menos um dos sockets monitorados. Quando novos dados chegam ou algum evento relevante ocorre, o processo "acorda"e pode tratar tais requisições de maneira imediata. De tal forma, o desperdício de recursos computacionais é diminuído significavelmente.

4.2 Listagem de rotinas

A seguir, descrevemos as principais rotinas do programa:

A função `handle_client_request` é responsável por tratar as solicitações recebidas de um cliente conectado. Primeiro, ela chama `recv` para receber os dados enviados pelo cliente. Se a conexão for encerrada ou ocorrer um erro, o socket do cliente é fechado. Caso contrário, uma resposta HTTP simples é enviada de volta ao cliente antes do encerramento do socket.

Já dentro da função `main`, que gerencia todo o funcionamento do servidor, é criado um socket TCP de escuta usando a função `socket` e o configura para permitir reutilização de endereço com `setsockopt`. Depois, configura a estrutura `sockaddr_in` para definir a família de endereços (`AF_INET`), o endereço IP (`INADDR_ANY`) e a porta de escuta (`PORT`); associa o socket ao endereço configurado usando `bind` e coloca o servidor em modo de escuta com `listen`; inicializa dois conjuntos de descritores de arquivos (`fd_set`), `master` e `read_fds`, para monitorar os sockets usando `select`; entra em um loop principal, onde monitora atividade nos sockets utilizando `select`. Quando um novo cliente tenta se conectar, a função `accept` é chamada para aceitar a conexão, e o novo socket é adicionado ao conjunto `master`.

Além disso, a função `handle_client_request` (quando há atividade em um socket já conectado) é chamada para tratar a requisição, e o socket é removido do conjunto após o término do atendimento. Este código demonstra como implementar um servidor TCP concorrente de forma eficiente, utilizando a função `select` para gerenciar múltiplas conexões simultâneas em um único processo, reduzindo a sobrecarga de recursos e melhorando o desempenho.

4.3 Análise dos Resultados Obtidos

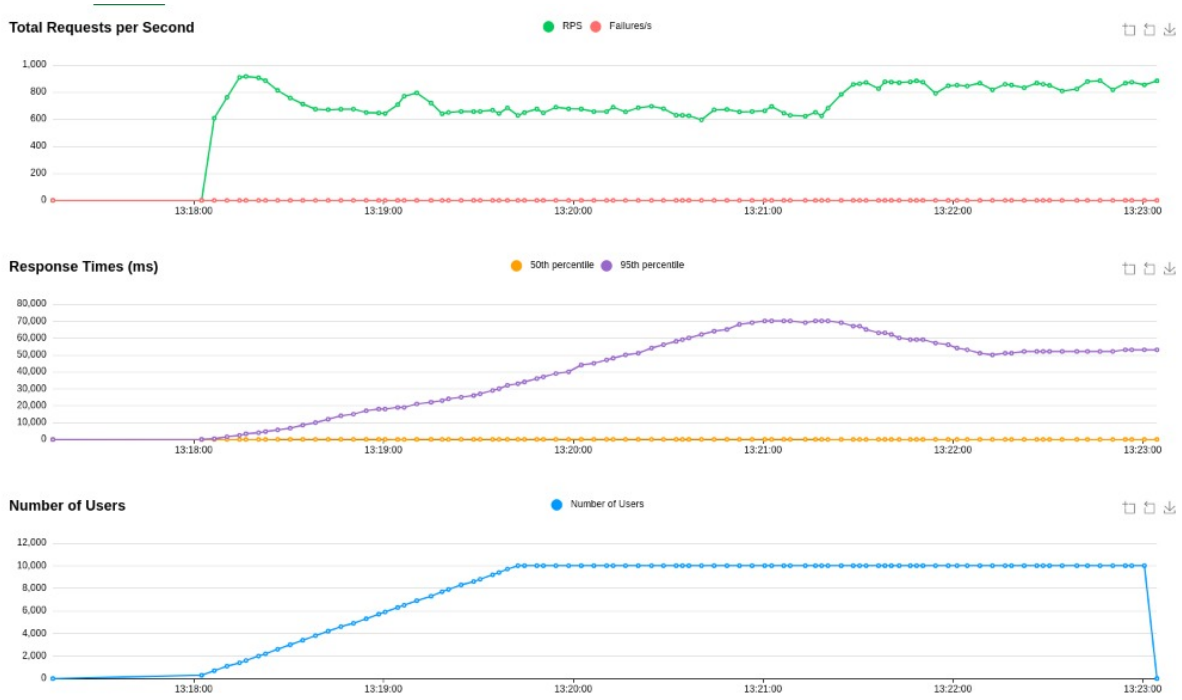

Figura 4: Resultados dos testes do servidor concorrente

Assim como no servidor utilizando threads e fila de tarefas, foi simulado um total de **10.000 usuários** com uma taxa de incremento de **100 usuários por segundo**, durante **5 minutos** de execução; além de uma limitação de no máximo 10 conexões que podem estar aguardando na fila de escuta do socket (antes de serem aceitas).

- **Total Requests per Second (RPS):**
  - O RPS (taxa de requisições por segundo) aumenta rapidamente no início, proporcionalmente ao aumento do número de usuários. Após atingir aproximadamente 900 RPS, o desempenho estabiliza, com pequenas oscilações ao longo do tempo. A quantidade de falhas (representada pela linha vermelha) inexistente, indicando que o servidor foi capaz de atender às requisições com alta disponibilidade.

- **Response Times (ms):**
  - O tempo de resposta aumenta gradualmente à medida que o número de usuários cresce. No percentil 95 (linha roxa), os tempos de resposta chegam a 70.000 ms (70 segundos) no pico de carga. O percentil 50 (linha laranja) permanece constante e baixo durante a maior parte do teste, indicando que a maioria das requisições foi atendida rapidamente, mas um número significativo de requisições teve atrasos muito altos. No cenário prático, tempos de resposta tão altos no percentil 95 poderiam impactar negativamente a experiência de alguns usuários.

- **Number of Users:**
  - O número de usuários aumenta de maneira linear, começando em 0 e chegando a 10.000 em aproximadamente 100 segundos. O servidor manteve o comportamento esperado com o aumento de usuários, sem quedas ou interrupções abruptas durante o teste. Após o pico, o número de usuários cai rapidamente ao final do teste.

O servidor apresentou bom desempenho inicial, conseguindo atender às requisições até atingir o pico de carga. Entretanto, apesar dos bons resultados gerais, os tempos de resposta elevados em cenários de alta concorrência podem comprometer aplicações em tempo real ou de alta demanda.

Referências

[Loc] Locust.io. *Locust*. Acesso em: 19 jan. 2025. URL: `https://locust.io/`.