Faculdade de Design, Tecnologia e Comunicação
Universidade Europeia

Relatório final trabalho prático de Sistemas Operativos

Tema: Multi-threading Application

Repositório: https://github.com/Kev1n18/Sistemas-Operativos

Docente: Pedro Rosa

Grupo:
Wesley Augusto-20200344
Kevin Dos Santos-20210448
Eularinani Ecombe-20210329

Desenvolvimento de Servidor Web Multi-Threading em Java

1. Descrição do Problema e Motivação:

Com o advento da internet e o crescimento exponencial de utilizadores online, a demanda por aplicações web eficientes e escaláveis tornou-se uma necessidade premente. Os servidores web desempenham um papel fundamental nesse contexto, sendo responsáveis por receber, processar e responder às solicitações dos clientes de forma rápida e confiável. No entanto, conforme o número de usuários e solicitações aumenta, os servidores enfrentam desafios de desempenho e escalabilidade.

Um dos principais desafios é a capacidade de lidar com múltiplas solicitações de forma simultânea, garantindo uma experiência de usuário fluida e sem interrupções. Tradicionalmente, os servidores web utilizam uma abordagem baseada em threads para lidar com múltiplas solicitações concorrentes. No entanto, o gerenciamento eficiente dessas threads e a prevenção de condições de corrida e outros problemas de concorrência podem ser complexos e desafiadores.

Diante desse cenário, surge a necessidade de explorar técnicas avançadas de programação, como multi-threading, para desenvolver servidores web capazes de lidar com alta concorrência de forma eficiente e escalável. Este projeto visa abordar esse desafio, propondo o desenvolvimento de um **servidor web** simples, porém robusto, utilizando a linguagem de programação Java e explorando os conceitos de multi-threading para lidar com múltiplos clientes simultaneamente.

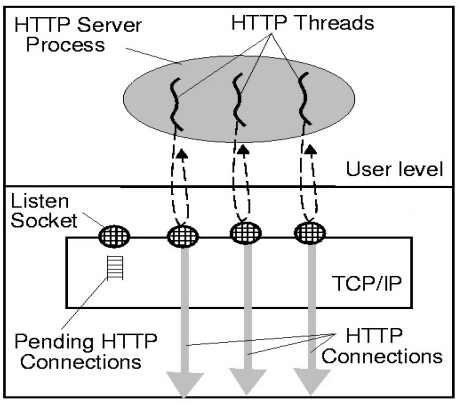
Identificação de Casos de Uso:

1. **Hospedagem de Sites**: O servidor desenvolvido pode ser utilizado para hospedar uma variedade de sites, desde páginas estáticas até aplicações web dinâmicas, garantindo uma resposta rápida e confiável às solicitações dos utilizadores.

2. **Aplicações Web com Alta Concorrência**: Serviços online que requerem alta concorrência, como salas de bate-papo, fóruns ou jogos multiplayer, podem se beneficiar significativamente de um servidor capaz de lidar eficientemente com múltiplas conexões simultâneas.

3. **Implementação de APIs e Serviços RESTful**: O servidor desenvolvido pode ser utilizado para implementar serviços RESTful e APIs, permitindo a comunicação eficiente entre clientes e servidores em ambientes distribuídos.

   Mais abaixo temos um diagrama de como funciona uma thread.



2. Descrição da Solução a Implementar:

Solução Genérica:

A solução desenvolvida consiste em um servidor web multithreading robusto programado em Java, projetado para atender às crescentes demandas por serviços web de alto desempenho. O servidor foi implementado para lidar com múltiplas conexões HTTP de forma simultânea, utilizando um modelo multithreaded para processar cada solicitação de maneira independente e eficiente. Essa abordagem garantiu uma distribuição eficaz dos recursos do servidor, proporcionando uma resposta rápida e confiável mesmo em situações de pico de tráfego.
Durante o desenvolvimento, a solução foi aprimorada com as seguintes **alterações que diferem da proposta inicial**:

- **Containerização com Docker:** O servidor web multithreading em Java foi encapsulado em containers Docker, facilitando a portabilidade, o isolamento e a gestão dos ambientes de execução.
- **Implementação na AWS:** Utilizamos a infraestrutura da AWS para hospedar os containers Docker, beneficiando-nos da escalabilidade e robustez da plataforma.
- **Load Balancer:** Um load balancer da AWS foi configurado para distribuir o tráfego de rede entre dois containers Docker, garantindo alta disponibilidade e balanceamento de carga eficiente.
- **Ambiente de Execução:** Os containers Docker são executados em máquinas virtuais Linux na AWS, proporcionando um ambiente escalável e robusto para o servidor web.
- **Arquivos aceites:** Na primeira proposta, foi mencionado que o servidor trabalharia apenas com arquivos estáticos do formato HTML, mas após algumas discussões com o docente houve a necessidade de adicionar mais alguns formatos aceites. Como **CSS, JAVASCRIPT, imagem(.JPG)**
- **Validação de requests:** A junção com a cadeira de **compiladores**, fez com que implementássemos esta funcionalidade. O servidor aceita apenas pedidos **GET e POST.** O que fez com implementássemos a validação dos pedidos através da expressão regular **"^(GET|POST)\\s(\\/\\S*)\\s(HTTP\\/\\d\\.\\d)$"**. **Caso pedido não seja um GET ou POST a página retorna um Bad request (400 error).**

Com essas melhorias, a solução desenvolvida se tornou mais robusta, escalável e eficiente, atendendo melhor às necessidades dos utilizadores e garantindo um desempenho excecional em diferentes condições de carga.

Enquadramento nas Áreas da Unidade Curricular:

Este projeto está alinhado com os objetivos da disciplina ao explorar conceitos avançados de programação em Java e sua aplicação na construção de sistemas distribuídos e servidores web. Ao desenvolver um servidor web multi-threaded em Java, os alunos terão a oportunidade de aprofundar seus conhecimentos em programação orientada a objetos, manipulação de threads e comunicação de rede.

Além disso, o projeto permite explorar conceitos teóricos, como protocolos de rede e protocolo HTTP, na prática, contribuindo para uma compreensão mais abrangente dos fundamentos da computação distribuída e desenvolvimento de sistemas web.

Ao final do projeto, espera-se que os alunos estejam aptos a aplicar os conceitos e técnicas aprendidos em cenários do mundo real, enfrentando desafios comuns na implementação de servidores web e sistemas distribuídos.

Requisitos Técnicos:

1. Proficiência avançada em programação Java e sólido entendimento dos conceitos de programação orientada a objetos, incluindo herança, polimorfismo e encapsulamento.

2. Conhecimento sólido dos princípios fundamentais de redes, incluindo protocolos **TCP/IP e HTTP**, bem como familiaridade com conceitos como requisição, resposta e cabeçalhos HTTP.

3. Capacidade de utilizar eficazmente ambientes de desenvolvimento integrado (IDEs) para Java, como Eclipse, IntelliJ IDEA, para desenvolver, depurar e testar o servidor web.

4. Experiência prévia e compreensão profunda de programação multithreaded em Java, incluindo sincronização, monitoramento de condições e práticas recomendadas para evitar condições de corrida e **deadlocks**.

5. Capacidade de projetar e implementar estruturas de dados e algoritmos eficientes para manipular solicitações HTTP concorrentes, garantindo o desempenho, a escalabilidade e a segurança do servidor web.

6. Familiaridade com técnicas de otimização de código e depuração de problemas de desempenho em ambientes multithreading, para garantir uma execução suave e eficiente do servidor sob diferentes condições de carga.

7. Habilidade para documentar adequadamente o código-fonte, incluindo comentários claros e concisos, e seguir boas práticas de desenvolvimento de software, como modularidade, coesão e baixo acoplamento.

8. Conhecimento e experiência em utilizar ferramentas de teste de carga, como o Apache JMeter, para avaliar o desempenho, a escalabilidade e a confiabilidade do servidor web em condições simuladas de tráfego intenso.

9. Conhecimento em Docker para containerização de aplicações, incluindo criação de Dockerfiles, gerenciamento de containers e compreensão dos conceitos de isolamento e portabilidade.

10. Experiência em configurar e gerenciar infraestrutura na AWS, incluindo a criação e configuração de **máquinas virtuais (EC2), gerenciamento de redes (VPCs)**, e uso de serviços de **balanceamento de carga (ELB)**.

11. Capacidade de configurar e gerenciar load balancers na AWS para distribuir o tráfego de rede entre múltiplos containers Docker, garantindo alta disponibilidade e balanceamento eficiente de carga.

12. Familiaridade com **sistemas operacionais Linux**, incluindo comandos básicos de administração e configuração, para gerenciar máquinas virtuais que executam containers Docker na AWS.

    Observação: A utilização de containers e de um load balancer de cloud, não nos livra do trabalho de configurar Docker e containers no terminal com os comandos da tecnologia Docker, ou seja, é necessário conhecimento profundo de Docker para poder

configurar Docker files, fazer expose de portas e etc. A única diferença é que a cloud poupa trabalho ao desenvolver de instalá-los e consumir recursos da sua própria máquina.

Arquitetura da Solução:

A arquitetura da solução foi cuidadosamente projetada para garantir a eficiência e a confiabilidade do servidor web multithreading. No núcleo da arquitetura está um servidor principal altamente escalável, capaz de gerenciar e coordenar conexões de clientes de forma eficiente.

Componentes da Arquitetura:

1. **Servidor Principal Multithreading:**
    - Cada vez que uma conexão é estabelecida, o servidor principal dinamicamente cria uma nova thread dedicada para lidar com as solicitações desse cliente específico. Essa abordagem permite que o servidor atenda a múltiplas solicitações simultaneamente, sem sobrecarregar o sistema ou comprometer sua estabilidade.

    - Cada thread é responsável por gerenciar integralmente uma conexão individual, manipulando a entrada e saída de dados associados à solicitação HTTP correspondente. Isso significa que o servidor pode processar solicitações de forma independente, distribuindo eficientemente a carga de trabalho entre as threads disponíveis.

2. **Containerização com Docker:**
    - O servidor web multithreading em Java foi encapsulado em containers Docker. Essa containerização proporciona portabilidade, isolamento e facilita a gestão dos ambientes de execução.
    - Dois containers Docker são utilizados para executar instâncias do servidor, aumentando a redundância e a capacidade de lidar com um maior volume de tráfego.

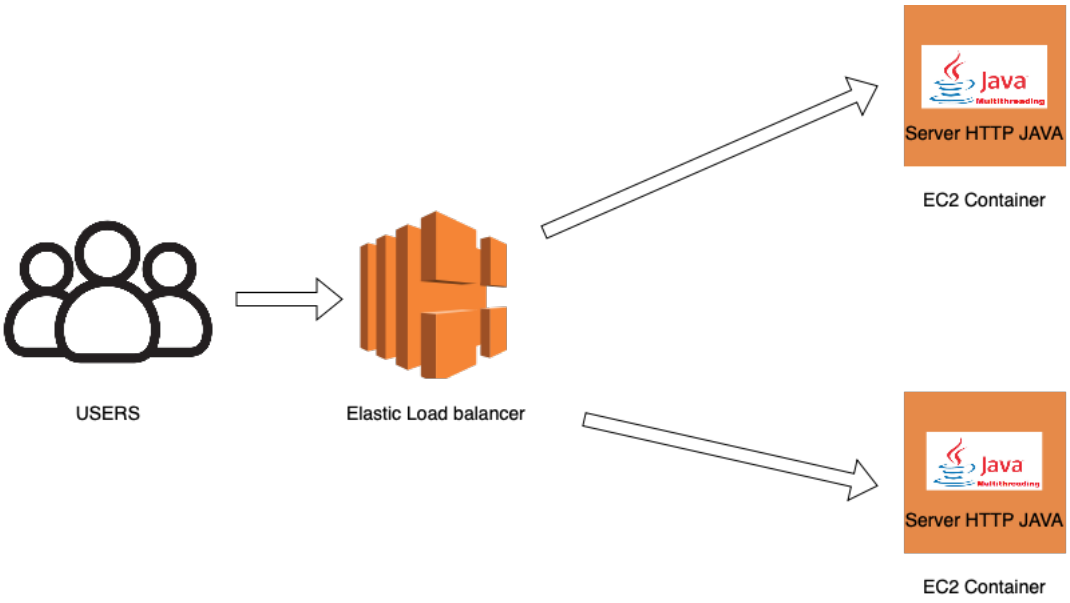
3. **Infraestrutura na AWS:**
    - Os containers Docker são implantados em máquinas virtuais (ECS) na AWS, oferecendo uma plataforma escalável e robusta para a execução do servidor web.
    - O ambiente de execução baseado em Linux nas instâncias ECS garante uma infraestrutura confiável e eficiente.

4. **Load Balancer da AWS:**
    - Um load balancer da AWS é configurado para distribuir o tráfego de rede entre os dois containers Docker. Isso garante alta disponibilidade e balanceamento eficiente de carga, proporcionando uma experiência de usuário fluida e responsiva.
    - O load balancer monitora as instâncias do servidor e distribui o tráfego de acordo com a carga atual de cada container, maximizando a utilização dos recursos e evitando sobrecargas.

Benefícios da Arquitetura:

- **Escalabilidade:** A utilização de containers Docker e a infraestrutura da AWS permitem escalar horizontalmente a solução conforme necessário, adicionando mais containers e ajustando os recursos da infraestrutura de maneira flexível.
- **Resiliência:** A configuração com múltiplos containers e um load balancer garante alta disponibilidade e resiliência a falhas, mantendo o serviço disponível mesmo em caso de problemas em uma das instâncias.
- **Desempenho:** A arquitetura multithreading permite que o servidor processe múltiplas solicitações simultaneamente, proporcionando uma resposta rápida e consistente às solicitações dos clientes, mesmo sob cargas de tráfego intenso.

Ao adotar essa arquitetura, garantimos que a solução seja altamente responsiva, escalável e confiável, atendendo a uma ampla gama de cenários de uso e proporcionando uma experiência de alta qualidade para os usuários finais.



Arquitetura

Tecnologias a Utilizar:

- **Linguagem de programação:** Java para desenvolver o servidor web, utilizando bibliotecas como java.net para comunicação de rede.
- **IDE para desenvolvimento Java:** Como IntelliJ IDEA, escolhida por sua eficiência e recursos avançados de desenvolvimento.
- **Ferramentas de teste: JUnit** para testes unitários e **Apache JMeter** para testes de carga, permitindo uma avaliação abrangente do desempenho e escalabilidade do servidor web em diferentes cenários de uso.
- **Containerização:** Docker: Utilizado para containerizar o servidor web, proporcionando portabilidade, isolamento e facilidade na gestão dos ambientes de execução. Ferramentas como Docker Compose podem ser utilizadas para orquestração dos containers.
- **Plataformas de cloud(AWS):**

    1. **EC2 (Elastic Compute Cloud):** Para a criação e configuração de máquinas virtuais (instâncias) que executam os containers Docker.
    2. **ELB (Elastic Load Balancing):** Para distribuir o tráfego de rede entre os containers Docker, garantindo alta disponibilidade e balanceamento eficiente de carga.

3. Planeamento e Calendarização:

Calendarização do Projeto

Semana	Data	Atividade
1	14 - 20 de fevereiro	Planejamento e Configuração Inicial
2	21 - 27 de fevereiro	Planejamento e Configuração Inicial
3	28 de fevereiro - 5 de março	Desenvolvimento do Servidor
4	6 - 12 de março	Desenvolvimento do Servidor
5	13 - 19 de março	Desenvolvimento do Servidor
6	20 - 26 de março	Desenvolvimento do Servidor
7	27 de março - 2 de abril	Containerização e Implementação na AWS
8	3 - 9 de abril	Containerização e Implementação na AWS
9	10 - 16 de abril	Containerização e Implementação na AWS
10	17 - 23 de abril	Otimizações e Testes de Carga
11	24 - 30 de abril	Otimizações e Testes de Carga
12	1 - 7 de maio	Configuração do Load Balancer
13	8 - 14 de maio	Testes e Correção de Erros
14	15 - 21 de maio	Testes e Correção de Erros

4. Resultados

Durante o desenvolvimento e teste do servidor web multithreading, uma série de resultados significativos foram obtidos, demonstrando a eficácia e a robustez da solução implementada.

Uma parte fundamental dos testes realizados foi a avaliação do desempenho e escalabilidade do servidor em diferentes cenários de uso. Utilizando a ferramenta Apache JMeter, realizamos testes de carga simulando um alto volume de tráfego de usuários acessando o servidor simultaneamente. Os resultados desses testes foram analisados para avaliar a capacidade do servidor de lidar com uma carga de trabalho substancial e garantir uma resposta rápida e consistente às solicitações dos clientes.

O gráfico gerado pelo Apache JMeter revelou que aproximadamente 80% das conexões foram bem-sucedidas durante os períodos de pico de tráfego. Isso indica que o servidor conseguiu manter um desempenho estável e uma boa taxa de resposta mesmo sob carga intensa.

No entanto, os restantes 20% das conexões resultaram em erros, incluindo uma quantidade significativa de requisições resultando em **"Bad Request" (400) e "Not Found" (404).**

A presença desses erros sugere áreas de melhoria no servidor, como o aprimoramento da gestão de requisições inválidas e a implementação de mecanismos de recuperação de falhas mais robustos. Embora a maioria das conexões tenha sido bem-sucedida, é crucial abordar esses casos de erro para garantir uma experiência de usuário consistente e confiável.

Em resumo, os resultados obtidos durante os testes de desempenho e escalabilidade demonstram que o servidor web multithreading é capaz de lidar eficientemente com uma carga substancial de tráfego, proporcionando uma resposta rápida e confiável à maioria das solicitações. No entanto, a presença de erros em uma parte significativa das conexões destaca a importância contínua da otimização e aprimoramento do servidor para garantir um desempenho consistente em todos os cenários de uso.

Sendo assim concluímos que o ponto fraco da nossa implementação consiste nas **conexões que não passam**. Vale enfatizar também que implementamos tudo que foi proposto e na secção anexos, anexamos capturas de ecrã sobre tudo que foi mencionado na secção resultados.

5. Bibliografia:

- https://leon-wtf.github.io/doc/java-concurrency-in-practice.pdf
- https://theswissbay.ch/pdf/Gentoomen Library/Programming/O%27Reilly Desining Series/O%27Reilly Head First Servlets and JSP.pdf
- Documentação oficial do Java SE.
- Documentação oficial do protocolo HTTP
- https://docs.aws.amazon.com/elasticloadbalancing/
- https://docs.aws.amazon.com/ecs/
- https://www.rfc-editor.org/rfc/rfc2616
- https://lp.jetbrains.com/intellij-idea-features-promo/?source=google&medium=cpc&campaign=EMEA_en_WEST_IDEA_Branded&term=intellij&content=693349187751&gad_source=1&gclid=Cj0KCQjwu8uyBhC6ARIsAKwBGpTdNM-lTUMD_d1MGPjpjfobyyEjuPg_27lRPtCYPQEz-cGfrDIoAAoaAkwYEALw_wcB

6.Anexos

Pagina Principal servida pelo servidor:

Identificação de threads **(0 é a thread de conexão ao servidor)** pela consola e extração do Body do request por parte do Regex:

```
20:01:01.114 [Thread-0] INFO com.example.core.ServerListenerThread - Connection accepted: /0:0:0:0:0:0:0:1
20:01:01.137 [Thread-2] INFO com.example.core.HttpConnectionWorkerThread - Request Line: GET / HTTP/1.1
20:01:01.137 [Thread-2] INFO com.example.core.HttpConnectionWorkerThread - Parsed Request: HttpRequest{method='GET', resource='/', httpVersion='HTTP/1.1'}
```

Retornos do servidor, caso o pedido não seja **GET ou POST**

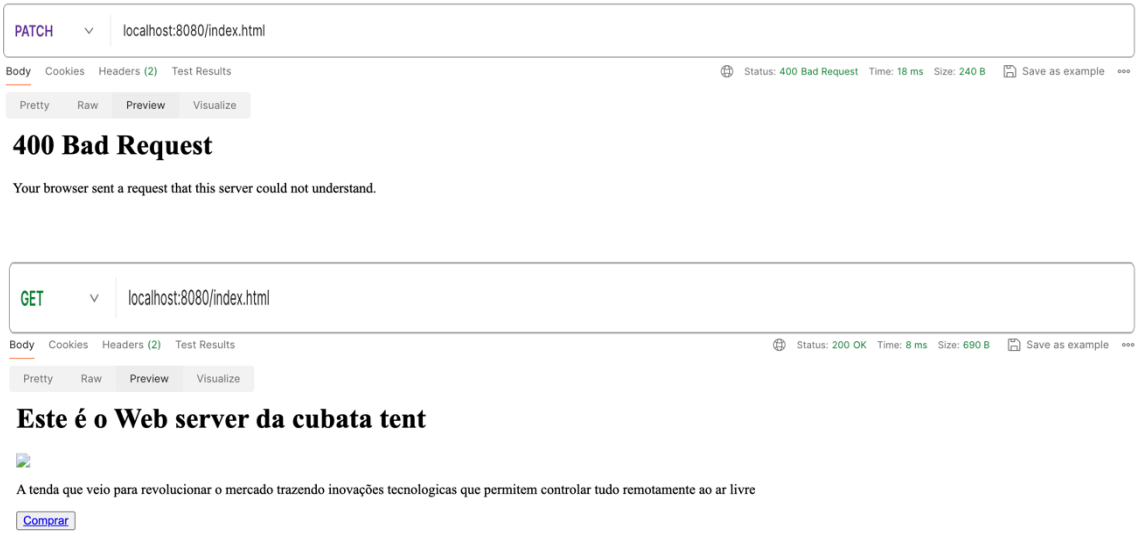


Tabela de conexões aceites e não aceites:

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(...
182	20:30:02.453	Thread Group 1...	HTTP Request	5		690	126	4	0
183	20:30:02.459	Thread Group 1...	HTTP Request	6		690	126	6	3
184	20:30:02.468	Thread Group 1...	HTTP Request	7		690	126	7	3
185	20:30:02.477	Thread Group 1...	HTTP Request	4		690	126	4	0
186	20:30:02.482	Thread Group 1...	HTTP Request	2		690	126	2	1
187	20:30:02.486	Thread Group 1...	HTTP Request	2		690	126	2	0
188	20:30:02.489	Thread Group 1...	HTTP Request	2		690	126	2	0
189	20:30:02.492	Thread Group 1...	HTTP Request	3		690	126	2	1
190	20:30:02.496	Thread Group 1...	HTTP Request	2		690	126	2	0
191	20:30:02.499	Thread Group 1...	HTTP Request	3		690	126	3	1
192	20:30:02.503	Thread Group 1...	HTTP Request	2		690	126	2	1
193	20:30:02.506	Thread Group 1...	HTTP Request	3		690	126	3	1
194	20:30:02.510	Thread Group 1...	HTTP Request	2		690	126	2	1
195	20:30:02.513	Thread Group 1...	HTTP Request	2		690	126	2	1
196	20:30:04.425	Thread Group 1...	HTTP Request	3		690	126	3	1
197	20:30:04.430	Thread Group 1...	HTTP Request	5		690	126	5	2
198	20:30:04.436	Thread Group 1...	HTTP Request	3		690	126	3	1
199	20:30:04.443	Thread Group 1...	HTTP Request	3		690	126	3	1
200	20:30:04.447	Thread Group 1...	HTTP Request	3		690	126	3	1
201	20:30:04.452	Thread Group 1...	HTTP Request	3		690	126	3	1
202	20:30:04.457	Thread Group 1...	HTTP Request	3		690	126	3	1
203	20:30:04.462	Thread Group 1...	HTTP Request	4		690	126	4	1
204	20:30:04.468	Thread Group 1...	HTTP Request	5		690	126	5	1
205	20:30:04.474	Thread Group 1...	HTTP Request	3		690	126	3	1
206	20:30:04.478	Thread Group 1...	HTTP Request	4		690	126	4	1
207	20:30:04.483	Thread Group 1...	HTTP Request	4		690	126	4	1
208	20:30:04.490	Thread Group 1...	HTTP Request	3		690	126	2	1
209	20:30:04.494	Thread Group 1...	HTTP Request	3		690	126	3	0
210	20:30:04.498	Thread Group 1...	HTTP Request	4		690	126	3	1
211	20:30:06.420	Thread Group 1...	HTTP Request	9		690	126	8	2
212	20:30:06.430	Thread Group 1...	HTTP Request	6		690	126	6	1

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(...
94	20:28:58.590	Thread Group 1...	HTTP Request	0		2599	0	0	0
95	20:28:58.591	Thread Group 1...	HTTP Request	7		690	126	7	2
96	20:28:58.598	Thread Group 1...	HTTP Request	2		2067	0	0	0
97	20:28:58.601	Thread Group 1...	HTTP Request	5		690	126	5	2
98	20:28:58.606	Thread Group 1...	HTTP Request	1		2067	0	0	0
99	20:28:58.608	Thread Group 1...	HTTP Request	7		690	126	7	1
100	20:28:58.615	Thread Group 1...	HTTP Request	1		2067	0	0	0
101	20:28:58.617	Thread Group 1...	HTTP Request	5		690	126	5	2
102	20:28:58.622	Thread Group 1...	HTTP Request	1		2067	0	0	0
103	20:28:58.623	Thread Group 1...	HTTP Request	5		690	126	4	1
104	20:28:58.628	Thread Group 1...	HTTP Request	1		2599	0	0	0
105	20:28:58.629	Thread Group 1...	HTTP Request	3		690	126	3	1
106	20:28:58.633	Thread Group 1...	HTTP Request	0		2067	0	0	0
107	20:28:58.635	Thread Group 1...	HTTP Request	5		690	126	5	2
108	20:28:58.640	Thread Group 1...	HTTP Request	1		2067	0	0	0
109	20:28:58.641	Thread Group 1...	HTTP Request	4		690	126	4	1
110	20:28:58.646	Thread Group 1...	HTTP Request	1		2599	0	0	0
111	20:28:58.648	Thread Group 1...	HTTP Request	6		690	126	6	1
112	20:28:58.655	Thread Group 1...	HTTP Request	0		2599	0	0	0
113	20:28:58.656	Thread Group 1...	HTTP Request	3		690	126	3	1
114	20:28:58.660	Thread Group 1...	HTTP Request	1		2067	0	0	0
115	20:28:58.661	Thread Group 1...	HTTP Request	3		690	126	3	1
116	20:28:58.665	Thread Group 1...	HTTP Request	0		2599	0	0	0
117	20:28:58.666	Thread Group 1...	HTTP Request	5		690	126	5	2
118	20:28:58.671	Thread Group 1...	HTTP Request	1		2599	0	0	0
119	20:28:58.672	Thread Group 1...	HTTP Request	4		690	126	4	1
120	20:28:58.676	Thread Group 1...	HTTP Request	1		2067	0	0	0
121	20:29:00.584	Thread Group 1...	HTTP Request	7		690	126	7	1
122	20:29:00.591	Thread Group 1...	HTTP Request	1		2067	0	0	0
123	20:29:00.592	Thread Group 1...	HTTP Request	7		690	126	7	1
124	20:29:00.599	Thread Group 1...	HTTP Request	1		2599	0	0	0

Gráfico de threads(**a linha vermelha representa as conexões que falharam**):