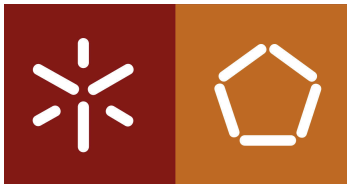

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Sistemas Distribuídos
Trabalho Prático

Grupo 26

Fernando Pires	**Edgar Ferreira**	**Pedro Teixeira**	**Pedro Gomes**
a77399	a99890	a103998	a104540

Data da Receção	
Responsável	
Avaliação	
Observações	

Sistemas Distribuídos

Fernando Pires a77399

Edgar Ferreira a99890

Pedro Teixeira a103998

Pedro Gomes a104540

dezembro, 2024

Índice

Lista de Figuras

1. Introdução

Este relatório apresenta o trabalho prático desenvolvido no âmbito da unidade curricular de **Sistemas Distribuídos**, do curso de Engenharia Informática na Universidade do Minho. O objetivo principal deste projeto foi a implementação de um **sistema distribuído** para armazenamento de dados em memória com acesso remoto, explorando conceitos fundamentais como concorrência, comunicação entre processos e atomicidade.

O sistema desenvolvido permite a interação de clientes com um servidor central através de **sockets TCP**, suportando operações de leitura e escrita em formato chave-valor. Além disso, o projeto visa atender a requisitos adicionais como a gestão de sessões concorrentes e a escalabilidade. O trabalho inclui um servidor, uma interface de cliente e cenários de teste para avaliação de desempenho.

Este documento descreve detalhadamente a **arquitetura do sistema**, a **implementação técnica** e as decisões tomadas ao longo do desenvolvimento. Serão também apresentados os resultados dos testes realizados, acompanhados de uma reflexão sobre o impacto das escolhas implementadas no desempenho global do sistema.

2. Arquitetura

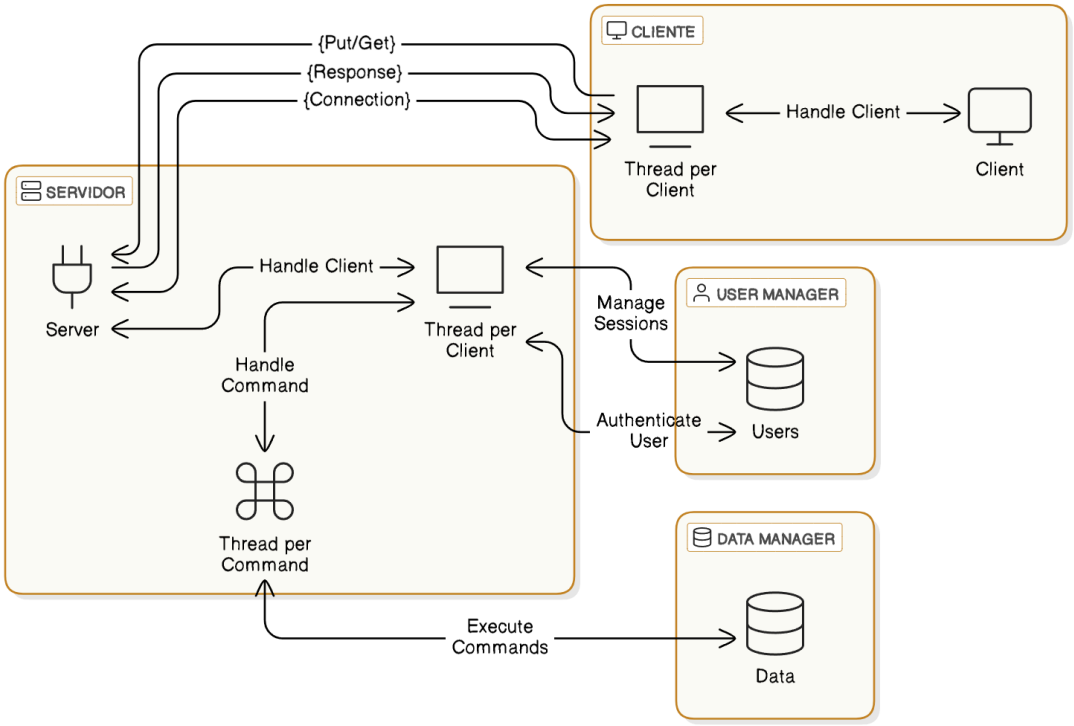

Figura 1: Arquitetura do Sistema

A arquitetura do sistema está organizada em quatro componentes principais: **Cliente**, **Servidor**, ***User Manager*** e ***Data Manager***. A figura representa a interação entre estes componentes, detalhando os fluxos de comunicação e a divisão de responsabilidades.

2.1. Conexão

O protocolo implementado foca-se no uso das classes `Demultiplexer`, `TaggedConnection` e `Frame`, que permitem a gestão da transmissão e receção de mensagens entre cliente e servidor de maneira **estruturada** e **concorrente**, permitindo que múltiplas *threads* enviem e recebam os dados de forma independente através dos *sockets* TCP.

A serialização é realizada pela classe `TaggedConnection`, que escreve os dados da mensagem num fluxo binário (`DataOutputStream`).

A desserialização, por outro lado, lê os dados a partir de um fluxo binário (`DataInputStream`), reconstruindo a estrutura original da mensagem com base no formato acima.

O método `receive()` da classe `TaggedConnection` lê uma mensagem completa do fluxo de entrada e cria uma nova instância de `Frame`. No caso de receção de mensagens do lado do cliente, o

`Demultiplexer` processa essa mensagem, encaminhando-a para a *thread* responsável, com base na *tag* associado.

2.1.1. *Frame*

Cada mensagem trocada segue uma estrutura com uma *tag* e os dados (em *bytes*), derivada da classe `Frame`. Este design modular de encapsulamento torna a comunicação mais previsível e simplifica a serialização/desserialização. Com esta abordagem, diferentes tipos de comandos podem ser representados e tratados uniformemente.

2.1.2. *Demultiplexer*

A classe `Demultiplexer` atua como um controlador do lado do cliente que permite que múltiplas *threads* recebam mensagens simultaneamente. Internamente, utiliza *locks* e variáveis de condição para garantir que cada mensagem seja associada à *tag* correta, permitindo notificações eficientes e acesso controlado aos dados.

2.2. Fluxo geral do sistema

1. O cliente inicia uma conexão com o servidor.

2. O servidor cria uma *thread* dedicada para processar os comandos do cliente.

3. O cliente autentica-se enviando credenciais para o servidor.

4. Após autenticação bem-sucedida os comandos do cliente são processados e as operações são delegadas ao *Data Manager* para execução.

5. O *Data Manager* recupera ou atualiza os dados conforme os comandos recebidos e responde ao servidor, que retransmite os resultados ao cliente.

6. Quando o cliente encerra a sessão, a *thread* é finalizada.

2.3. Principais características

A arquitetura destaca-se pela sua **modularidade**, com separação clara entre Cliente, Servidor, *User Manager* e *Data Manager*, facilitando a manutenção e **escalabilidade**. O sistema suporta **concorrência** eficiente através de *threads* independentes, garantindo operações simultâneas por múltiplos clientes, enquanto *locks* e sincronização asseguram a **consistência dos dados**. Além disso, o uso de *conditions* e *signals* permite **coordenação** eficiente entre *threads* em operações bloqueantes. A comunicação é estruturada, utilizando *frames* etiquetados para padronização e eficiência. A **robustez** é assegurada com encerramento limpo de sessões e tratamento de erros, como comandos inválidos ou desconexões inesperadas.

3. Cliente

O programa do cliente foi feito baseado numa *CLI* (*command-line interface*) que, enquanto está a ser executado, lê comandos a partir do input do utilizador e executa-os.

```
[INFO] List of commands:
[INFO] - help: List all commands.
[INFO] - put <key> <value>: Adds or updates a single key-value pair in the server.
[INFO] - get <key>: Retrieves the value associated with the given key, or returns null
if the key does not exist.
[INFO] - multiput <n> <key> <value>...: Adds or updates n key-value pairs in the
server.
[INFO] - multiget <n> <key>...: Retrieves n values for the specified keys and returns
them as a map.
[INFO] - getWhen <key> <keyCond> <valueCond>: Blocks until the value of keyCond matches
valueCond, then retrieves the value of key.
[INFO] - end: End program
```

3.1. Cliente *multi-threaded*

O cliente utiliza *threads* dedicadas para processar cada comando enviado ao servidor de forma independente. Cada comando é gerido por uma instância de `CommandHandler`, que envia o pedido ao servidor e aguarda a resposta sem bloquear a interação do utilizador. Além disso, o uso de `TaggedConnection` e `Demultiplexer` garante que as respostas sejam tratadas corretamente para cada comando, permitindo comunicação simultânea e eficiente com o servidor.

3.2. Autenticação

Ao iniciar a aplicação no modo cliente, o utilizador é recebido com um menu para se autenticar, podendo escolher entre registar-se ou iniciar sessão:

```
[INFO] Choose an option, write 1 or 2.
[INFO] 1. Register
[INFO] 2. Login
[INFO] Your Option:
>
```

Para registar-se, o utilizador insere um nome único e uma palavra-passe. Caso a conta seja criada com sucesso, é apresentada uma mensagem de confirmação, ou, em caso de conflito, é informado que o nome já existe:

```
[INFO] Username:
>
[INFO] Password:
>
[INFO] New account created with username: <username>
[INFO] <username> already exists
```

Se optar por iniciar sessão, o utilizador insere as credenciais. O servidor valida os dados e, em caso de sucesso, autoriza o acesso:

```
[INFO] Username:
>
[INFO] Password:
>
[INFO] Logged in successfully!
```

Caso o número máximo de sessões seja atingido, o utilizador aguarda na fila até poder aceder. Este processo garante que apenas utilizadores registados acedam à aplicação e organiza o acesso de forma segura e eficiente.

3.3. Pedidos ao servidor

Após a autenticação, o cliente pode enviar comandos ao servidor para realizar operações específicas:

- ***Put***: Associa um valor a uma chave no servidor. Por exemplo, ao executar `put Sistemas Distribuidos`, a chave `"Sistemas"` é criada ou atualizada com o valor `"Distribuidos"`.
- ***Get***: Recupera o valor associado a uma chave. Por exemplo, `get Sistemas` retorna o valor `"Distribuidos"` para a chave `"Sistemas"`.
- ***MultiPut***: Realiza a escrita atómica de múltiplos pares chave-valor. Por exemplo, `multiput 2 Inteligencia Artificial Comunicacoes Computador` atualiza as chaves `"Inteligencia"` e `"Comunicacoes"` com os valores `"Artificial"` e `"Computador"`, respetivamente.
- ***MultiGet***: Recupera de forma atómica valores associados a múltiplas chaves. Por exemplo, `multiget 2 Inteligencia Comunicacoes` devolve os pares `"Inteligencia: Artificial"` e `"Comunicacoes: Computador"`.
- ***GetWhen***: Retorna o valor de uma chave após validar que outra chave atingiu um valor específico. Por exemplo, `getwhen Sistemas Desenvolvimento Software` espera que a chave `"Desenvolvimento"` tenha o valor `"Software"`. Assim que a condição é satisfeita, retorna o valor associado à chave `"Sistemas"`.

O servidor utiliza sincronização para garantir consistência e processamento eficiente destas operações em ambientes concorrentes.

4. Servidor

4.1. Servidor *multi-threaded*

O servidor é o componente central do sistema, projetado para suportar múltiplas conexões de clientes simultaneamente. Utiliza um modelo *multi-threaded*, onde cada cliente é tratado de forma independente, garantindo escalabilidade e eficiência em ambientes concorrentes.

4.1.1. *Thread Pool*

O servidor utiliza dois *thread pools* principais:

- *Pool* de Clientes: Gere conexões de clientes através de instâncias de `ClientHandler`. Cada cliente conectado é associado a uma *thread* do *pool*, garantindo comunicação sem bloquear novas conexões.
- *Pool* de Comandos: Processa os comandos enviados pelos clientes. Cada comando é delegado a uma instância de `CommandExecutor`, que interage com o `DataManager` para realizar as operações necessárias.

Estes *thread pools* permitem reutilização eficiente de recursos, enquanto mantêm limites configuráveis para o número de *threads* ativas e tarefas na fila.

4.1.2. *Thread per Request*

Para comandos recebidos pelos clientes, o servidor segue o modelo ***thread per request***, criando uma *thread* para cada operação a partir do *pool* de comandos. Este modelo assegura que os pedidos sejam tratados de forma paralela e independente, otimizando o desempenho do sistema.

4.2. *User Manager*

O `UserManager` gere a autenticação e as sessões dos utilizadores.

- **Autenticação**: Valida credenciais de *login* ou cria novas contas durante o registo.
- **Gestão de Sessões**: Limita o número de utilizadores ativos, utilizando uma fila de espera (`queue`) para controlar acessos concorrentes de forma justa.
- **Encerramento de Sessões**: Ao desconectar-se, o utilizador é removido da lista de sessões ativas, e um sinal é enviado para libertar espaço para novos clientes.

4.3. *Data Manager*

O `DataManager` é responsável pela gestão de dados do sistema, armazenados em um modelo de chave-valor.

- **Operações Atómicas**: Implementa métodos como `put` , `get` , `multiput` e `multiget` com garantias de consistência e integridade dos dados.

- **Condições de Bloqueio**: Suporta operações como `getwhen`, permitindo que *threads* esperem por condições específicas antes de continuar, utilizando *locks* e variáveis de condição.

5. Testes de Desempenho

Este capítulo apresenta os testes realizados para avaliar o **desempenho**, **escalabilidade** e **robustez** do sistema desenvolvido. Os testes foram conduzidos em diferentes cenários, simulando cargas variadas de clientes e operações concorrentes. O servidor foi configurado com um limite de **100 sessões ativas simultâneas** (*max_clients*), de modo a refletir restrições reais de um ambiente de produção.

Os testes têm como objetivo medir a capacidade do sistema em lidar com múltiplos clientes de forma eficiente, analisando métricas como o tempo total de execução, tempo médio de resposta e comportamento sob *workloads* intensos. Adicionalmente, avaliam-se as limitações do sistema em cenários de alta concorrência e possíveis melhorias para futuras otimizações.

Abaixo seguem dois exemplos de testes realizados.

5.1. *testWorkload*

O teste de *workload* avalia o desempenho do sistema ao processar operações intensas com diferentes números de clientes (10, 50, 100, 500 e 1.000). Cada cliente executa 100 operações alternadas de *put* e *get*, após registo e *login*. Foram medidos o tempo total de execução e o tempo médio de resposta por cliente.

Os resultados mostram que o tempo total cresce significativamente com o aumento do número de clientes, especialmente para 500 e 1.000 clientes. O tempo médio de resposta permanece estável para até 100 clientes (22.000 ms), mas dobra para cerca de 46.000 ms com 1.000 clientes.

Estes dados indicam que o sistema é eficiente para cargas moderadas, mas enfrenta limitações com *workloads* intensos devido à concorrência por recursos e ao limite de clientes do servidor.

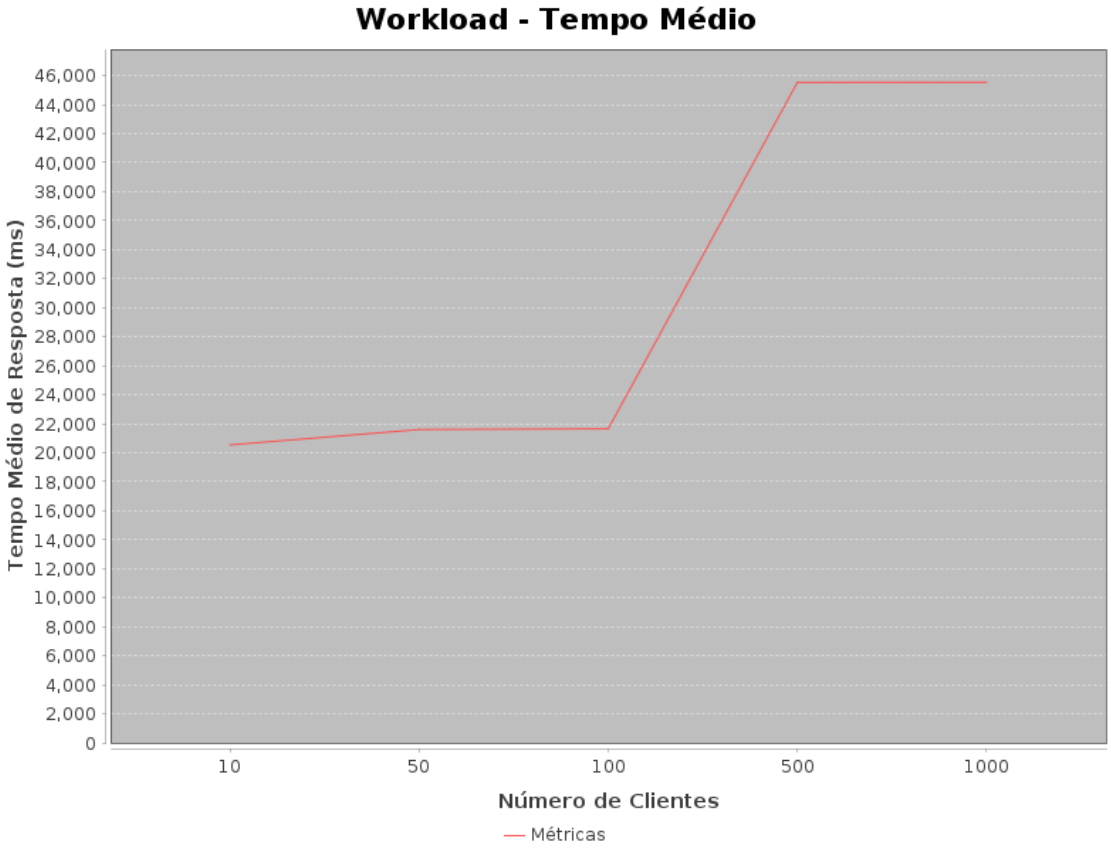

Figura 2: Teste Workload - Tempo Médio

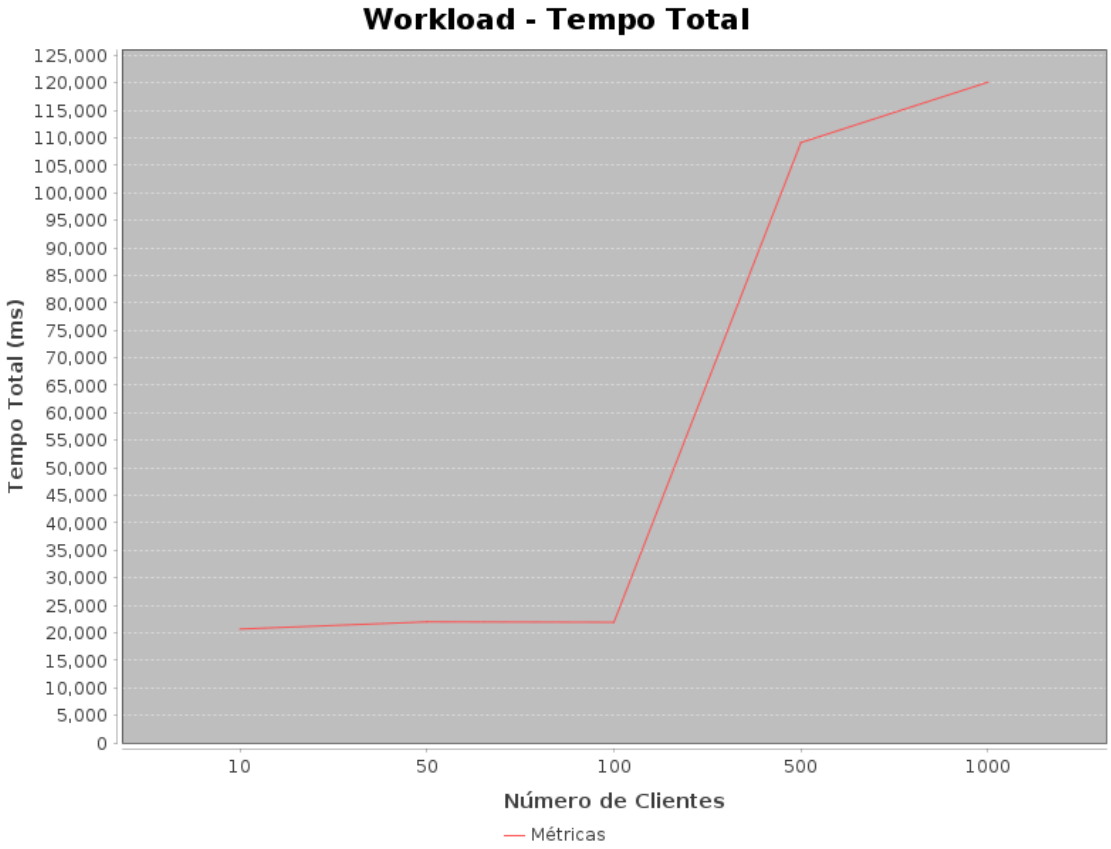

Figura 3: Teste Workload - Tempo Médio

5.2. *testScalability*

O teste de escalabilidade avalia o desempenho do sistema com diferentes cargas de clientes, medindo o tempo total de execução e o tempo médio de resposta. Os clientes realizam as operações de registo, *login*, *put*, *get* e encerramento da sessão, e esses tempos são registados.

Os resultados mostram que o tempo médio de resposta é baixo (~300 ms) para cargas leves (10 a 100 clientes), mas aumenta gradualmente, atingindo cerca de 750 ms para 10.000 clientes. O tempo total cresce de forma exponencial, sendo significativamente maior para cargas altas (32.000 ms para 10.000 clientes).

Esses resultados indicam que o sistema é eficiente para cargas moderadas, mas enfrenta limitações de escalabilidade devido à concorrência por recursos e ao limite de clientes do servidor, especialmente em cenários com muitos clientes simultâneos.

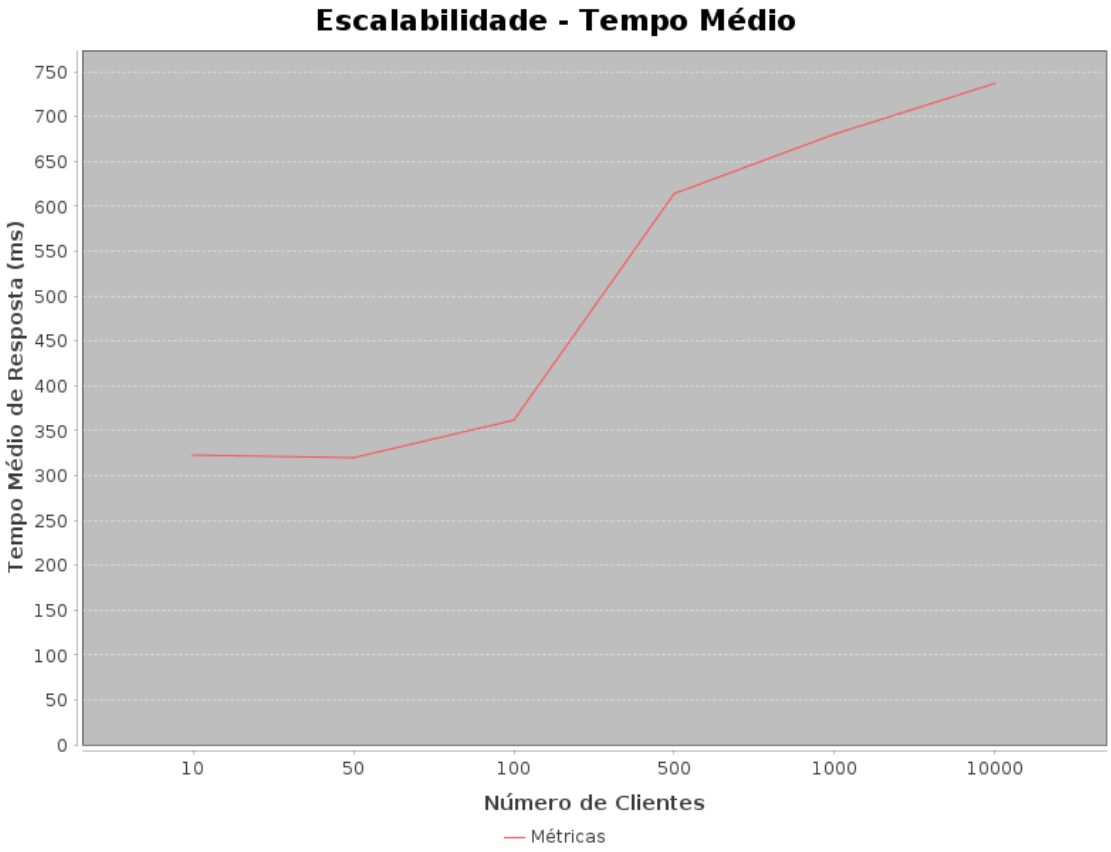

Figura 4: Teste Escalabilidade - Tempo Médio

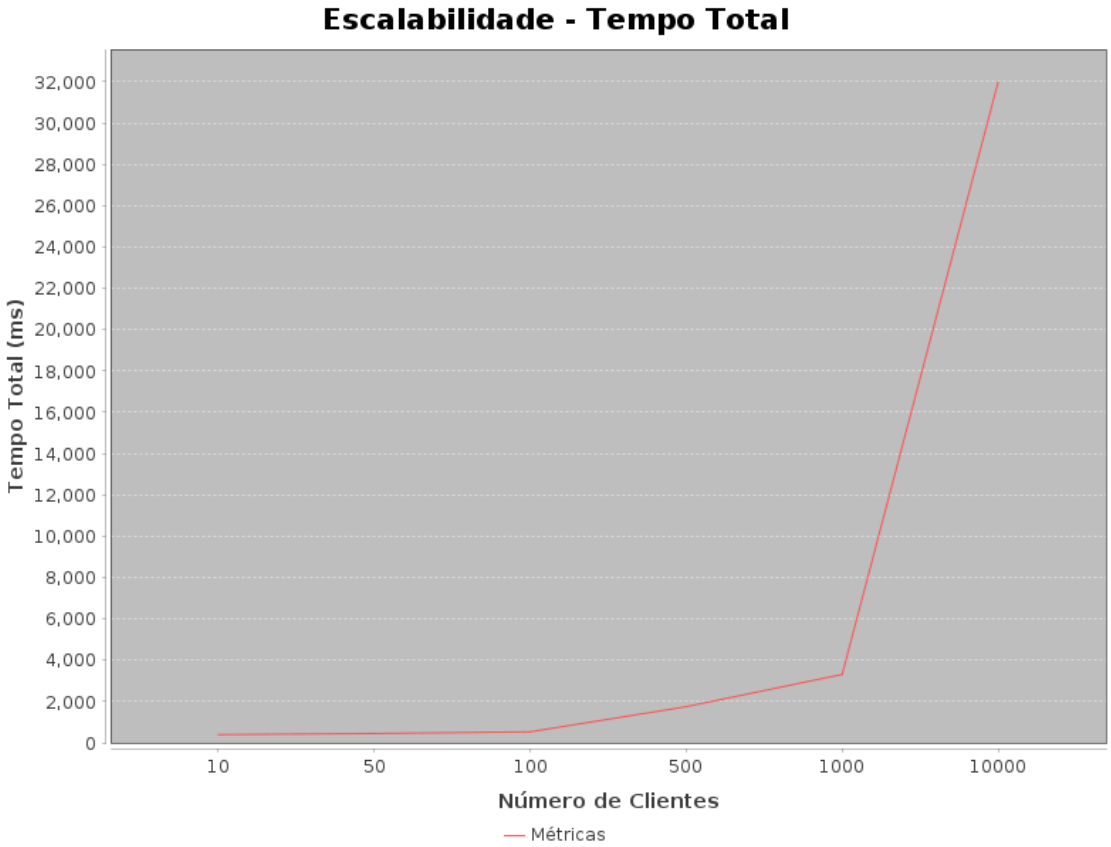

Figura 5: Teste Escalabilidade - Tempo Total

6. Conclusões

Este projeto consistiu no desenvolvimento de um sistema distribuído funcional que permite operações de armazenamento e recuperação de dados com múltiplos clientes simultâneos. Apesar de desafios iniciais na definição da arquitetura e algumas questões na implementação dos testes de desempenho, foi implementado um servidor robusto com autenticação eficiente e gestão de sessões, garantindo consistência nas operações de dados e comunicação confiável com os clientes.

Em suma, o projeto alcançou os objetivos propostos, superou desafios e estabeleceu uma base sólida para futuras melhorias, demonstrando a aplicação prática de conceitos de sistemas distribuídos.