Universidade do Minho
Escola de Engenharia

Engenharia Informática

2024/2025

Sistemas Distribuídos – Trabalho Prático

Armazenamento de Dados em Memória com Acesso Remoto

Grupo 21:

Afonso Gregório de Sousa - A104262

Filipa Oliveira da Silva - A104167

Maria Cleto Rocha - A104441

Mário Rafael Figueiredo da Silva - A104182

1. Introdução

Este projeto, desenvolvido no âmbito de Sistemas Distribuídos na Universidade do Minho, foca na implementação de um serviço de armazenamento de dados partilhado. A informação é mantida num servidor central e acedida remotamente por clientes através de sockets TCP, permitindo operações de inserção e consulta em um formato chave-valor. O sistema foi desenhado para atender múltiplos clientes concorrentemente, armazenando dados em memória e empregando estratégias que reduzem a contenção e minimizam por sua vez o uso de threads ativas. Este relatório descreve a arquitetura do serviço, os protocolos utilizados e as decisões de desenho chave que orientaram a construção do projeto.

A arquitetura proposta busca de certa forma otimizar o desempenho e a eficiência do sistema através do uso eficaz de mecanismos de sincronização e gestão de concorrência, assegurando assim que o acesso e a modificação dos dados ocorrem de maneira segura e sem conflitos. As interações entre o cliente e o servidor são meticulosamente geridas para proporcionar uma experiência de utilização ágil, crucial para sistemas que dependem da integridade e da velocidade na recuperação de dados. Através da análise de cenários de uso comuns e potenciais desafios operacionais, este relatório também detalha as justificativas para escolhas técnicas específicas, contribuindo para a compreensão da aplicação prática dos conceitos de sistemas distribuídos.

2. Arquitetura da Solução

A arquitetura do nosso sistema de armazenamento de dados distribuído foi concebida para assegurar eficiência e segurança. Segue-se uma descrição das classes que considerámos fundamentais e as suas funções no sistema, perante o enunciado proposto.

2.1. Cliente

A classe ‘Cliente’ é responsável por estabelecer e manter a comunicação com o servidor através de Sockets TCP. A mesma encapsula a lógica de conexão, envio e receção de dados, permitindo dessa forma aos clientes interagir com o servidor para realizar operações de armazenamento e consulta de dados.

Variáveis de instância:

- *Socket socket*: Socket para a comunicação com o servidor.
- *DataInputStream input*: Para ler dados do servidor.
- *DataOutputStream output*: Para enviar dados ao servidor.

2.2. BibliotecaCliente

Esta classe serve como fachada para as operações que os clientes podem realizar, abstraindo a complexidade das comunicações de rede. Para além disso, permite a simplicidade da InterfaceCliente, pois evita que esta precise de lidar diretamente com os detalhes da comunicação.

Variáveis de instância:

- *Cliente cliente:* utilizada para enviar mensagens ao servidor.

2.3. InterfaceCliente

É a interface de linha de comando que permite aos utilizadores interagir com o sistema através da 'BibliotecaCliente'.

Funcionalidades:

- *Menu de Utilizador:* Oferece um menu interativo para realizar operações como registo, autenticação, e manipulação de dados.
- *Processamento de Comandos:* Encaminha comandos do utilizador para a 'BibliotecaCliente' para processamento.

2.4. Servidor

Suporta as funcionalidades essenciais para interagir com múltiplos clientes simultaneamente, permitindo o armazenamento de dados em memória e a manipulação dos pares chave-valor de forma eficiente e concorrente.

Funcionalidades:

- Gere conexões de clientes;
- Gere o registo e autenticação de utilizadores;
- Fornece um conjunto de operações no armazenamento partilhado(*put*, *get*, *multiput*,*multiget*, *getwhen*);
- Garante a integridade dos dados.

3. Comunicação Cliente-Servidor

Como mencionado no enunciado, Clientes interagem com o servidor através de sockets TCP de forma a inserir e consultar informação. Esta relação é fundamental nos sistemas distribuídos, pois é um dos pilares para a comunicação entre processos que estão a executar em máquinas distintas. Nos sistemas distribuídos, esta arquitetura de comunicação não só permite a divisão de tarefas e a partilha de recursos, mas também oferece flexibilidade e escalabilidade ao sistema.

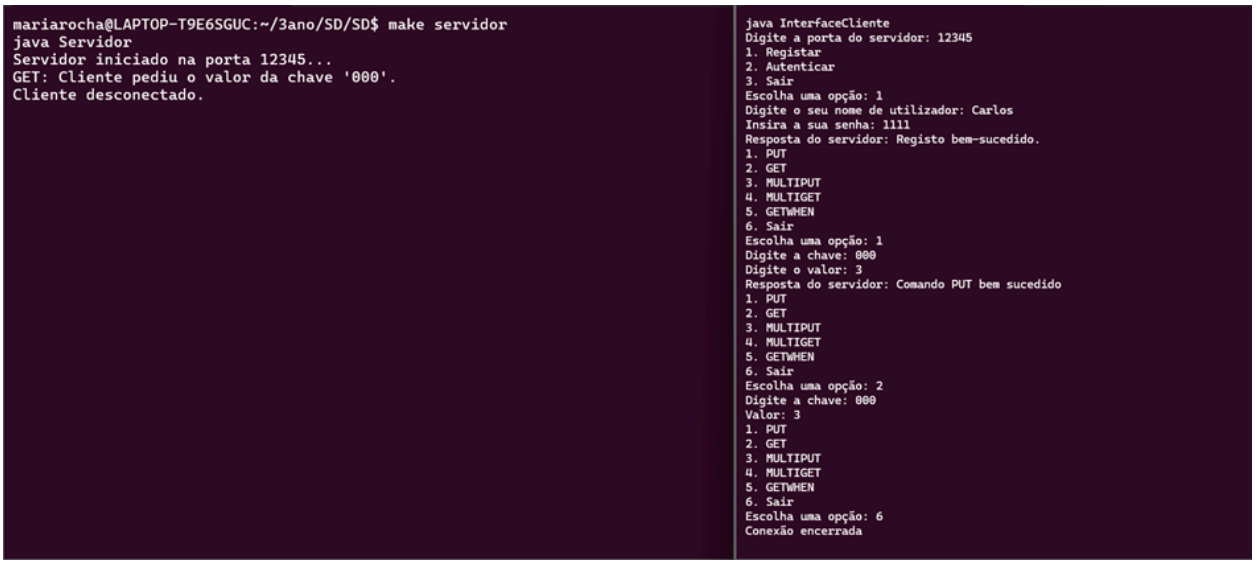

Figura 1 - Fluxo de Interação Cliente-Servidor

A Figura 1 demonstra um exemplo específico de um processo de interação entre o cliente e o servidor num sistema de armazenamento de dados distribuídos. A sequência de operações inclui:

1. Inicializar o Servidor: O servidor é iniciado e fica à espera de conexões na porta especificada.
2. Conexão do Cliente: O cliente estabelece uma conexão e inicia o diálogo com o servidor.
3. Registo e Autenticação: O cliente realiza operações de registo e autenticação com o servidor, que valida as credenciais.
4. Operações de Dados: O cliente executa várias operações de dados, como PUT e GET. Estas operações envolvem o envio de uma chave e a receção de um valor correspondente do servidor.
5. Terminar Sessão: Após as operações, o cliente encerra a sessão, desconectando-se do servidor.

3.1. Implementação dos Sockets

- *Cliente*

A classe Cliente configura um socket para se conectar ao servidor utilizando a porta especificada. Isto permite que o cliente envie comandos de operações e receba respostas do servidor. Esta classe é também responsável por enviar requisições e processar respostas do servidor, utilizando 'DataInputStream' e 'DataOutputStream' para manipular os fluxos de dados.

- *Servidor*

No servidor, um 'ServerSocket' recebe e aceita conexões dos clientes. Cada conexão é tratada em uma thread separada, permitindo que múltiplas requisições sejam processadas simultaneamente sem bloqueio, o que é crucial para a escalabilidade e desempenho do sistema.

O uso de threads separadas para cada cliente permite que o servidor gira várias sessões de utilizador ao mesmo tempo, aumentando a sua capacidade de resposta do sistema.
As operações via sockets garantem também que os dados transmitidos entre o cliente e o servidor sejam encapsulados e transmitidos de forma segura, minimizando a possibilidade de erros de transmissão e ataques externos.

3.2. Funções e Fluxos de Trabalho

- *Registo e Autenticação:*

Os clientes podem se registar e autenticar no sistema através de comandos enviados ao servidor, que valida as credenciais e mantém um registo dos utilizadores numa estrutura de dados concorrente.

- *Operações de Dados:*
    - *PUT* e *GET*: Os clientes podem inserir (*PUT*) e recuperar (*GET*) dados baseados em chaves. O servidor processa essas solicitações e responde apropriadamente, garantindo dessa forma a integridade dos dados com bloqueios e condições.
    - Operações Compostas: *'MULTIPUT'* e *'MULTIGET'* permitem a manipulação de múltiplas chaves simultaneamente, otimizando a interação com o servidor e facilitando o desenvolvimento de aplicações mais complexas.
    - Operações Condicionais: *'GETWHEN'* permite a obtenção de dados com base em condições específicas, suspendendo a operação até que uma chave condicional (keyCond) assuma um valor específico (valueCond). Ao utilizar bloqueios condicionais, o método minimiza o consumo de recursos, evitando assim verificações frequentes e otimizando a gestão de condições de corrida.

- *Gestão Concorrente do Estado:*
    - *'handlePut'* e *'handleGet'*: Estas funções no servidor gerem o estado dos dados de maneira thread-safe usando um 'ConcurrentHashMap' e bloqueios explícitos, para assegurar a consistência dos dados mesmo sob condições de alta concorrência.

3.3. Otimização de Threads e Redução de Latência

A eficiência no uso de threads é crucial para a escalabilidade e a performance de sistemas distribuídos. No nosso sistema, adotamos práticas de design que não só gerem as sessões de utilizador de forma eficaz mas também minimizam o número de threads acordadas, reduzindo dessa forma o overhead e melhorando por sua vez a resposta do sistema.

Técnicas de Minimização de Threads Acordadas:

- *Bloqueios e Sincronização:* Empregamos *'ReentrantLocks'* e condições (*Condition*) para controlar o acesso e a modificação dos estados compartilhados. Estes mecanismos asseguram que as threads não fiquem acordadas desnecessariamente à espera de recursos já ocupados. As threads que não podem prosseguir são colocadas em espera de forma eficaz até que a condição necessária seja atendida.
- *Monitoramento de Condições de Concorrência:* Implementamos também uma lógica para monitorar condições específicas que devem ser satisfeitas antes que uma operação prossiga (como em *'handleGetWhen'*). Isto evita a ativação constante de threads que não têm trabalho útil a realizar, diminuindo de tal forma o consumo de CPU e melhorando, por sua vez, o desempenho geral do sistema.

Impacto na Performance e Eficiência:

- *Redução de Overhead de Context Switching:* Ao minimizar as threads ativas desnecessariamente, o sistema reduz o overhead associado ao switching de contextos entre threads, o que é crítico em ambientes de alta carga.
- *Melhoria na Escalabilidade:* A capacidade de atender a um número maior de requisições com menos recursos ativos demonstra uma melhoria significativa na escalabilidade e eficiência do sistema.

4. Testes e Resultados

De forma a avaliar a eficácia e o desempenho do nosso sistema de armazenamento de dados, conduzimos uma série de testes com um número variável de operações, cujos resultados são ilustrados no gráfico da Figura 2. Estes testes foram projetados com o intuito de medir a latência média de diferentes operações ('*GET', 'PUT', 'MULTIGET', 'MULTIPUT', 'GETWHE*N') sob cargas de trabalho crescentes, representadas por 10, 100 e 1000 operações.

Os testes foram realizados utilizando o '*ClienteTeste'*, que simula múltiplos clientes que realizam operações simultâneas no servidor, permitindo assim observar o comportamento do sistema sob condições de alta concorrência. Esta abordagem é crucial para identificar potenciais obstáculos e para garantir que o sistema mantenha, de certa forma, a integridade e a consistência dos dados em cenários de uso real.

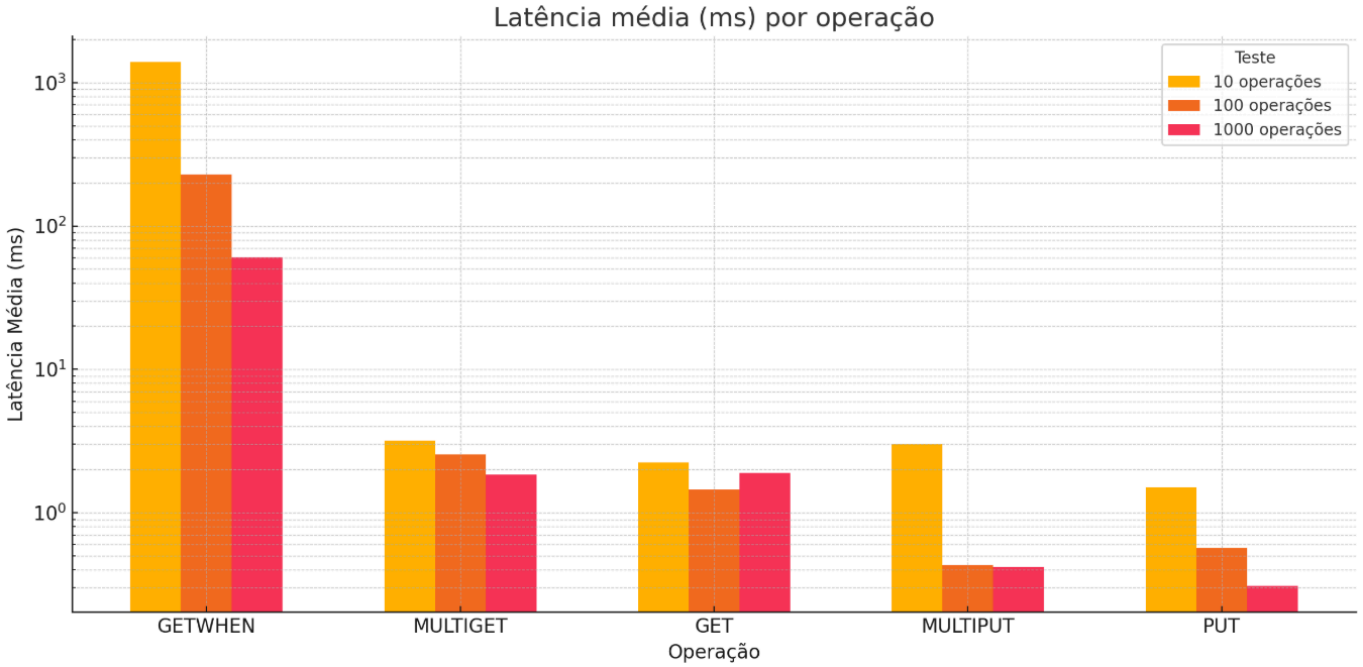

Figura 2 - Gráfico de latência média para diferentes operações e volumes de teste

Avaliação dos Resultados:

- *'GETWHEN'* mostrou a maior latência, especialmente com o aumento do número de operações. Este resultado reflete o custo associado ao bloqueio condicional e à espera de que certas condições sejam atendidas antes que a operação possa ser completa.
- *'MULTIGET'* e '*MULTIPUT'* apresentaram uma redução significativa na latência com o aumento das operações, sugerindo assim que o sistema lida eficientemente com operações em lote, aproveitando melhor as conexões de rede e os recursos do servidor.
- *'PUT'* e '*GET'* mantiveram as latências consistentes, destacando a eficácia do sistema em gerir operações simples de leitura e de escrita.

Estes resultados não só confirmam a robustez do nosso sistema ao lidar com diferentes tipos de operações sob variadas cargas, mas também destacam áreas para melhoria contínua, como a otimização do '*GETWHEN'* para reduzir a latência em cenários de espera condicional.

5. Conclusões

Concluindo, este projeto implementou um serviço de armazenamento de dados partilhado utilizando uma arquitetura cliente-servidor com comunicação via sockets TCP. Métodos como *'put'*, *'get'*, *'multiPut'*, *'multiGet'* e *'getWhen'* foram desenvolvidos para manipular operações de dados, demonstrando a aplicação prática de conceitos de concorrência e gestão de estado compartilhado. Esta abordagem reforçou a nossa compreensão teórica e destacou desafios práticos, como a necessidade de sincronização eficaz entre threads para evitar condições de corrida e garantir a integridade dos dados.

Embora tenhamos conseguido gerir múltiplas requisições simultâneas com sucesso, percebemos que a otimização do gerenciamento de conexões e a segurança dos dados são áreas que poderiam ser melhoradas. Estratégias como a implementação de técnicas de hashing para segurança de senhas ou o uso de conexões encriptadas poderiam ser formas de melhorar e prevenir o sistema contra ataques externos. Adicionalmente, a implementação de uma estratégia de caching eficiente poderia minimizar a latência e maximizar a eficiência do acesso a dados frequentemente requisitados.

Este trabalho não só ilustrou a complexidade envolvida na construção de sistemas distribuídos robustos, mas também forneceu uma base sólida para exploração e melhorias futuras nesta área relevante da computação.