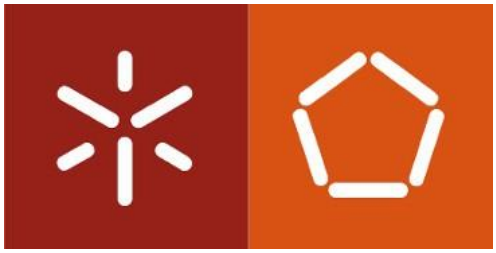

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Sistemas Distribuídos

Ano Letivo de 2024/2025

Armazenamento de dados em memória com acesso remoto

Grupo 42
A104532 - Tomás Sousa Barbosa
A104274 - João Miguel Freitas Rodrigues
A100756 - Mariana Miguel Leão Barros Oliveira Pinto
A100551 - Diogo Miguel Torres Moreira de Oliveira Pinto

28 de dezembro de 2024

Índice

Conteúdo

1. Contextualização

O projeto consiste na implementação de um sistema de armazenamento de dados distribuído, onde um servidor mantém a informação em memória e múltiplos clientes acedem à mesma através de conexões TCP. O sistema implementa diversas funcionalidades como a autenticação de utilizadores, operações de leitura e escrita (simples, compostas e condicional), e gestão de concorrência com limites de utilização.

O sistema foi projetado com foco em:

- Concorrência e paralelismo
- Escalabilidade
- Consistência de dados
- Controlo de acesso
- Operações atómicas
- Gestão eficiente dos recursos

2. Arquitetura do Sistema

2.1. Visão geral

O sistema utiliza uma arquitetura cliente-servidor onde múltiplos clientes podem se conectar simultaneamente no servidor através de *sockets* TCP. O servidor é responsável por armazenar os dados em memória e processar pedidos de clientes que se ligam ao mesmo. O cliente é responsável por enviar *frames* ao servidor dependendo do pedido que querem fazer, obtendo depois uma resposta do mesmo com o resultado obtido. É utilizado um protocolo baseado em uma frame para serializar e desserializar os dados.

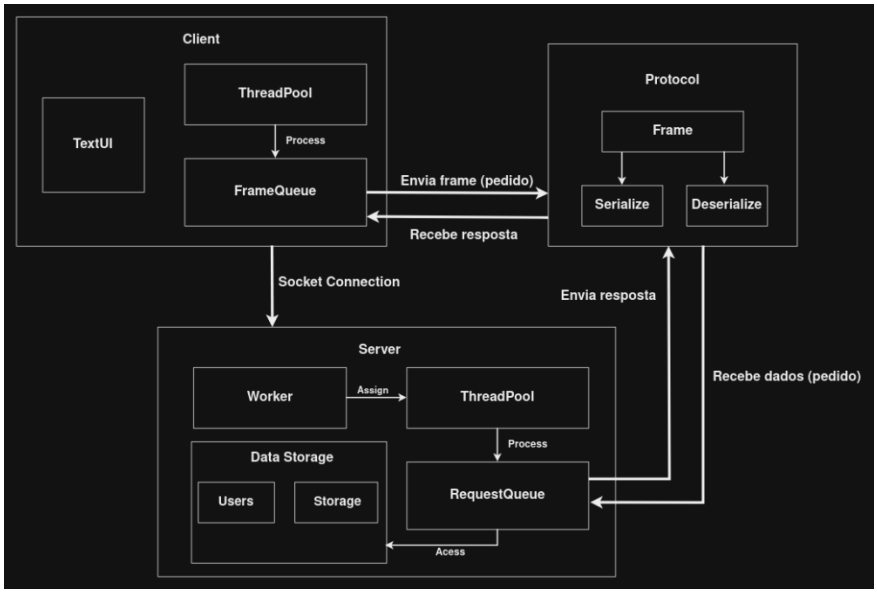

Figura 1 – Arquitetura Geral do Sistema

2.2. Componentes Principais

Cliente (Client.java)

- Implementa uma *thread pool* para o processamento paralelo de pedidos.
- Mantém uma *queue* para ordenação de *frames.*
- Processa respostas assíncronas.
- Gere sessões de um utilizador.
- Implementa *callbacks* para operações que ficam bloqueadas.

Servidor (Server.java)

- Aceita múltiplas conexões concorrentes.
- Utiliza uma *thread pool* para processamento de pedidos.
- Implementa o controlo de concorrência para acesso aos dados.
- Gere a autenticação e o registo de utilizadores.
- Processa operações de leitura/escrita no *dataStorage.*

Armazenamento (DataStorage.java)

- Mantém os dados em memória usando estruturas *thread-safe.*
- Implementa controlo de concorrência via *ReadWriteLock.*
- Suporta operações atómicas e compostas.
- Gere condições para operações que ficam bloqueadas.

3. Protocolo de Comunicação

3.1. Frames

O sistema utiliza *frames* como unidade básica de comunicação entre cliente e servidor. Cada *frame* contém:

- **tag**: Identificador único do pedido.
- **type**: Tipo da operação (Login,Create_Account, Put, Get, Multiput, Multiget GetWhen).
- **data**: Dados da operação
- i**sResponse**: *Flag* que indica se a *frame* é de resposta

3.2 Tipos de Operações

Autenticação

Login

Propósito: Autenticação de utilizadores existentes
Dados: **User object (username, password)**
Resposta: **Boolean** indicando sucesso/falha

Create Account

Propósito: Criação de novos utilizadores
Dados: **User object (username, password)**
Resposta: **Boolean** indicando sucesso/falha

Operações Básicas

Put

Propósito: Escrita simples de par chave-valor
Dados: **PutOne object (key, value)**
Resposta: **Boolean** indicando sucesso/falha

Get

Propósito: Leitura simples por chave
Dados: **String (key)**
Resposta: **byte[ ]** com o valor ou *array* vazio se não encontrado

Operações Compostas

Multiput

Propósito: Escrita atómica de múltiplos pares
Dados: **Map<String, byte[ ]>**
Resposta: **Boolean** indicando sucesso/falha
Atomicidade: Todas as escritas são feitas ou nenhuma é realizada

Multiget

Propósito: Leitura de múltiplas chaves
Dados: **Set<String>** com as chaves
Resposta: **Map<String, byte[ ]>** com os resultados

Getwhen

Propósito: Leitura condicional
Dados: **GetWhen object (key, conditionKey, conditionValue)**
Resposta: **byte[ ]** com o valor quando a condição é satisfeita
Comportamento: Bloqueia até a condição ser satisfeita

3.3 Serialização

Características do protocolo de serialização:

- Formato binário eficiente
- Serialização diferente para cada tipo de operação

- Mantém formato consistente entre cliente e servidor
- Suporte de dados de tamanho variável
- Verificação da integridade

4. Controlo de Concorrência

O controlo de concorrência é uma das partes essenciais do sistema, de modo a garantir eficiência e integridade durante o processamento de dados e operações. Para a gestão do mesmo, decidimos organizar em três níveis:

4.1 No Cliente

Para o controlo de concorrência no cliente, decidimos utilizar várias estratégias como uma *queue* para enviar *frames* de forma ordenada através de uma *CustomQueue* implementada que garante que a *queue* seja *thread-safe*. Utilizamos uma *threadPool* (implementada por nós) de modo a garantir que os clientes sejam *multi-threaded*, ou seja que consigam mandar pedidos paralelamente. Nas operações críticas decidimos utilizar *locks* de forma a garantir a consistência de dados durante os acessos simultâneos. Por fim decidimos utilizar um mecanismo de *callbacks* para respostas assíncronas, através da criação de uma ***resultThread*** na função ***getWhen***, de modo a garantir o fluxo principal do programa sem o interromper.

4.2 No Servidor

Para garantir o controlo de concorrência no Servidor as estratégias utilizadas foram o controlo sobre o número de conexões simultâneas, onde o servidor contém um número máximo de conexões simultâneas, utilizando um *lock* de forma a garantir que o número de clientes conectados não tivesse leituras e escritas incorretas. Quando se chega ao número máximo de conexões ao mesmo tempo, o próximo cliente que tente realizar uma operação, vai ficar em espera e só é atendido quando algum outro cliente decide terminar a sua conexão, permitindo assim que o número máximo seja respeitado. Foi utilizada uma *threadPool* de forma a processar os pedidos de forma paralela, e utilizada uma *queue* que garante que os pedidos que cheguem ficam colocados em espera de forma ordenada, utilizando outra vez a *CustomQueue* implementada por nós que contém um *lock*, garantindo que seja *thread-safe*.

4.3 No DataStorage

Por fim, no *DataStorage*, de modo a garantir a concorrência, decidimos utilizar um *storageLock*, relativo às operações de escrita e leitura. Assim, utilizando o ReentrantReadWriteLock(), de modo a utilizar o *writeLock* em operações de escrita e o *readLock* em operações de leitura, é possível obter atomicidade nas operações compostas. Para além disso, decidimos utilizar um usersLock relativo apenas à autenticação dos clientes no sistema, utilizando na mesma o ReentrantReadWriteLock(). Por fim, decidimos utilizar *conditions,* de forma a gerir operações que ficavam em espera, permitindo uma sincronização eficiente entre as *threads.*

5. Decisões sobre os Desafios Enfrentados

5.1 ThreadPool

Para a comunicação entre o cliente e o servidor decidimos implementar as nossas próprias *threadPools*, tanto no cliente como no servidor, de modo a gerir as múltiplas operações concorrentes de forma eficiente. No Servidor, a threadPool serve para processar as tarefas da fila de pedidos, de modo a limitar o número máximo de *threads* a processar simultaneamente, reutilizar as *threads* existentes, sem ser preciso criar novas para cada tarefa e permitir processar várias tarefas paralelamente. No cliente, a *threadPool* é utilizada para gerir o envio de *frames* para o servidor, de forma a permitir enviar vários *frames* sem bloquear a *thread* principal, gerir a concorrência e manter a ordem de operações através da sendQueue.

5.2 Controle de Concorrência

Ao nível dos *Locks* decidimos utilizar o *ReadWriteLock* em vez de um *lock* simples na gestão dos dados, de forma a aumentar o desempenho do sistema, permitindo que duas *threads* com objetivos diferentes (uma de leitura e outra de escrita) consigam ambas aceder aos dados sem que uma fique bloqueada.

5.3 Protocolo Customizado

Inicialmente decidimos implementar o sistema onde a comunicação era direta e as respostas eram obtidas no momento em que se executava, ou seja, de forma sequencial, mas, ao implementar o cliente *multi-thread,* reparamos que esta estrutura já não ia funcionar, portanto decidimos utilizar o uso de *frames*, identificadas por uma *tag* e um tipo, de forma a garantir que os pedidos sejam executados de forma ordenada e o controlo seja mais eficiente e escalável.

5.4 Interface do Utilizador

Para a Interface do Utilizador decidimos utilizar a classe **Menu** e a classe **TextUI**. Esta implementação foi pensada de forma a criar uma interface de utilizador clara e modular, de forma a separar claramente a lógica de apresentação do menu e das ações que ele executa.

A classe Menu serve para gerir a exibição das opções, a leitura de input do utilizador e o direcionamento das ações através de *handlers.* Esta abordagem permitiu-nos ter uma melhor organização de código e facilitar a *manutenção*.

A classe *TextUI* utiliza dois menus distintos, um para a autenticação e outro para as operações do sistema. Esta separação garante que apenas os utilizadores autenticados podem aceder às operações do sistema.

Assim, através destas duas classes conseguimos garantir flexibilidade para adicionar novas funcionalidades ao menu e facilidade de *manutenção*.

6. Cenários de Teste

Nesta etapa começamos por fazer testes funcionais, verificando se os métodos de autenticação garantiam que um utilizador já registado não se conseguia registar outra vez, ou um utilizador não registado conseguia fazer login. Para além disso, testamos as operações de escrita simples, composta e condicional, e verificamos, através de exemplos simples, que iam de acordo com o expectável.

Para testar o número máximo de clientes em concorrência, verificamos que, quando esse número chegava ao seu limite, é escrito no servidor que o limite foi atingido e o cliente vai ficar em espera. Assim, quando o novo cliente deseja realizar uma operação, esta vai ficar à espera de que algum cliente se desconecte do servidor, continuando assim a operação. Desta forma, garantimos que o limite de conexões concorrentes é respeitado.

7. Conclusão

O projeto desenvolvido destaca-se pela sua arquitetura que promove um equilíbrio entre os diferentes aspetos essenciais como a concorrência e persistência, a simplicidade e escalabilidade, bem como o desempenho e a confiabilidade. Esta abordagem não só atende aos requisitos técnicos e funcionais, como também assegura uma solução robusta, adaptável e eficiente. Ao longo da execução foram explorados conceitos fundamentais de concorrência, discutidos e trabalhados durante as aulas da Unidade Curricular. Esses conhecimentos foram aplicados de forma prática no projeto, criando uma solução que incorpore boas práticas de desenvolvimento.

Além disso, este projeto serviu como uma oportunidade de consolidação de conceitos teóricos, de forma a promover o trabalho futuro.