Escola de Engenharia

Universidade do Minho

Sistemas Distribuídos 2024/25

Trabalho Prático

Armazenamento de dados em memória com acesso remoto

Grupo 7

Pedro Seabra Vieira
a104352

pedrovieira712

Pedro Filipe Maneta Pinto
a104176

pedropinto27

Marco António Fernandes Brito
a104187

Marco9165

Tomás Henrique Alves Melo
a104529

hhtomaswt11

1 Índice

2 Introdução

A crescente demanda por sistemas de armazenamento e processamento de dados eficientes impulsiona a adoção de arquiteturas distribuídas para atender às necessidades de desempenho e escalabilidade. Neste contexto, este trabalho prático visa o desenvolvimento de um **serviço de armazenamento de dados partilhado**, que utiliza uma interface chave-valor acessível remotamente através de sockets TCP.

O principal objetivo do projeto é implementar um sistema que permita a interação entre clientes e um servidor central, garantindo a inserção e consulta de dados de forma concorrente e eficiente. Para tal, será utilizada uma abordagem que prioriza a minimização de contenção de recursos e o uso otimizado de threads, visando reduzir a latência e melhorar o desempenho do sistema.

Este relatório descreve a arquitetura desenvolvida, os protocolos adotados para comunicação entre os componentes, e as decisões de projeto tomadas ao longo do processo de implementação.

3 Arquitetura do sistema

3.1 Servidor

O servidor inicializa um socket TCP na porta número 8080 conhecido pelos clientes. Para além disso, é criado uma instância de ServerModel que cria o gestor de utilizadores e o mapa de acesso. Ainda, é criado uma threadPool, projetada para gerenciar threads de forma dinâmica e eficiente.

O servidor espera conexões de clientes e aceita as conexões destes, sendo criado um socket que gere a conexão com o cliente. Uma nova tarefa é enviada à threadpool para tratar a conexão do cliente. Desta forma, fica assegurado que cada cliente seja atendido numa thread separada, permitindo o processamento simultâneo de várias conexões.
Para cada conexão de cliente um objeto CommonIdent é criado, sendo inicializado com o socket criado anteriormente.

Consulta do lado do servidor:

```
Server started on port 8080
New client connected: /127.0.0.1:52756
ACTION [REGISTER] - USER [pedro] - STATUS [SUCCESS] - DETAILS [User registered]
```

Figura 1- Server Output

3.2 Cliente

O Cliente estabelece uma conexão com um servidor que está à escuta na porta 8080 no mesmo host (localhost). Cada cliente possui uma implementação multi-threaded refletida pela classe Demux, permitindo realizar múltiplos pedidos e receber respostas assíncronas. Utiliza a classe Demux para gerenciar a comunicação com o servidor (envio e recebimento de mensagens).

O cliente pode executar, de seguida, funcionalidades permitidas pela aplicação como operações de registo, login, logout, operações de escrita e leitura simples (put, get), operações de escrita e leitura compostas (multiPut, multiGet) e operações de leitura condicional (getWhen).

Consulta do lado do servidor:

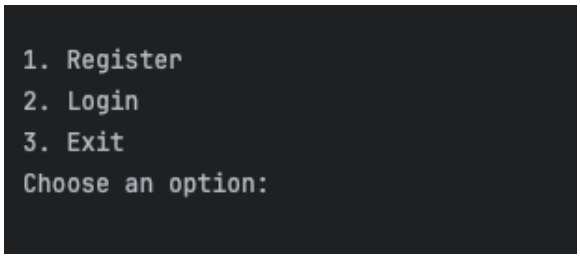


Figura 2 - Cliente OutPut Menu #1

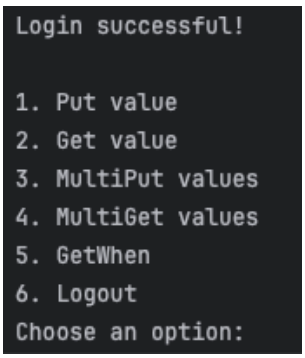


Figura 3 - Cliente OutPut Menu #2

3.3 Message

A classe Message é uma implementação de uma estrutura de mensagem que serve como meio de comunicação entre cliente e servidor num sistema distribuído. Esta classe encapsula as informações necessárias para realizar diferentes tipos de operações e utiliza a interface Serializable para permitir que as mensagens sejam enviadas e recebidas por meio de sockets em formato binário. Nesta classe, definimos os diferentes tipos de mensagem suportados (enumeração): REGISTER, LOGIN, LOGOUT para operações de autenticação; PUT, GET, MULTIPUT, MULTIGET para operações de armazenamento de dados; GETWHEN para leitura condicional e RESPONSE para respostas do servidor.

3.4 Comunicação cliente-servidor

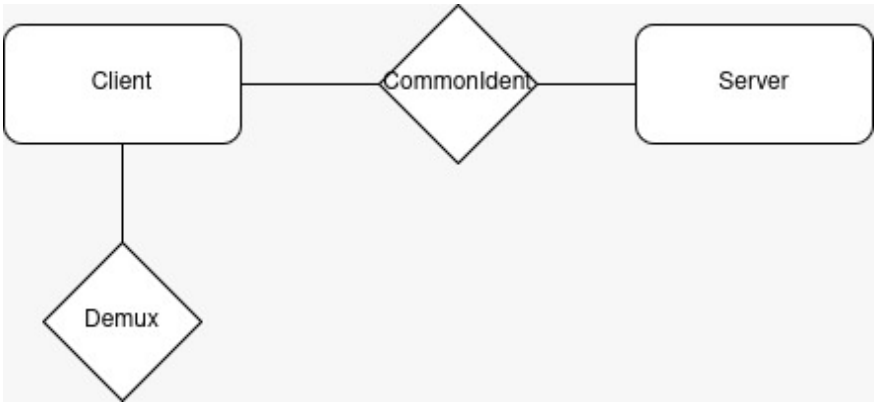
Para implementar a comunicação entre cliente-servidor foram criadas duas classes, Demux e CommonIdent.

3.5 Demux

A classe Demux encapsula um CommonIdent e, para além de delegar o envio de mensagens para esta, disponibiliza a operação receive, que bloqueia a thread invocadora até chegar uma mensagem com o tipo especificado como argumento do método, retornando o conteúdo desta. Esta classe é especialmente útil para o cliente, porque permite diferenciar entre mensagens de notificação enviadas pelo servidor e respostas específicas a pedidos feitos pelo cliente.

3.6 CommonIdent

A classe CommonIdent encapsula e gerencia a identificação e comunicação do socket. Tem como função a gestão da troca de mensagens entre cliente e servidor e vice-versa recorrendo a recursos de etiquetamento de mensagens com base no tipo da mensagem. Assim, foram implementados métodos para enviar e receber mensagens através do socket, utilizando serialização e escrita em formato binário, por meio das classes DataInputStream e DataOutputStream. Como esta classe é utilizada tanto no cliente quanto no servidor, os métodos de envio e receção foram adaptados para atender às necessidades específicas de cada um. Esta classe faz a "ponte" de comunicação entre mensagens cliente-servidor, garantindo eficácia, uma vez que é associada ao socket de cada cliente. O método sendMessage permite o envio dos parâmetros da mensagem do cliente para o servidor (como username, password, key, value, entre outros, dependendo do tipo de mensagem enviada pelo cliente) e envio dos parâmetros da mensagem de resposta ao método do servidor para o cliente. O método receiveMessage permite ao cliente receber a mensagem de resposta do servidor e ainda permite ao servidor ler os parâmetros da mensagem enviada pelo cliente e constrói-a, de modo a poder executar operações futuras com base no tipo da mensagem recebido.



3.7 User

A classe User é uma representação de um utilizador no sistema e encapsula informações relacionadas à autenticação e ao estado de login. Esta classe implementa a interface Serializable, que permite que os objetos dessa classe sejam serializados para transmissão ou armazenamento, uma funcionalidade essencial em sistemas distribuídos. É nesta classe que há o gerenciamento de informações de autenticação de um utilizador (nome de

utilizador e password), armazenamento do estado atual do login do utilizador (logado ou não) e, portanto, facilita a validação de credenciais para autenticação no sistema.

3.8 UserManager

Esta classe permite a gestão das operações do utilizador numa fase inicial, isto é, para registo, login e logout de clientes. O servidor, com base no tipo de mensagem recebida pelo cliente, acessa esta classe para realizar operações de registo, login e logout, suportadas por métodos definidos nesta classe.

3.9 MapAccess

Esta classe teve a necessidade de ser criada de modo a gerir as operações de escrita e leitura quer simples quer compostas e operações de leitura condicionais. Caso o tipo de mensagem recebida pelo servidor por parte do cliente seja put, get, multiPut, multiGet ou getWhen, o servidor acessa esta classe que dará suporte ao pedido feito pelo cliente. No fim da realização de cada uma destas operações, uma mensagem de resposta do lado do servidor é criada, de modo a responder ao pedido que havia sido feito pelo cliente, formada no método sendResponse, chamando o método sendMessage da classe CommonIdent, associado ao socket do cliente que realizou o pedido.

4 Funcionalidades implementadas

4.1 Limite de utilizadores concorrentes

Para garantir que apenas, no máximo, n sessões estejam de forma concorrente no servidor, isto é, garantir que apenas n clientes diferentes estejam a usar o servidor concorrentemente, recorremos à criação de uma variável de condição loginCondition associada ao writeLock. Ao pedido de login por parte de um utilizador, é verificado o número de logins atuais no servidor. Caso esse número seja maior ou igual do que o número de logins permitido no servidor, a thread associada ao pedido, aguarda em loginCondition.await(). Sempre que um cliente se desconecta do servidor, o número de logins atuais decresce e loginCondition.signal() permite que uma thread à espera em loginCondition.await() no método authUser acorde, permitindo o login do utilizador à espera.

4.2 Operação de leitura condicional – Funcionalidade Avançada

Para garantirmos o funcionamento deste método, na classe MapAccess, foi criado um Map<String, Condition> nomeado conditionMap, cuja chave representa a key a inserir no mapKeyValue (mapa global, compartilhado, acessado por todos os clientes), pelos métodos put e multiPut. Para cada vez que um par (key, value) é inserido no mapa global,

isto é, no mapKeyValue, através das operações de escrita simples e compostas, obtemos a condição com base na key inserida e é feito um alerta à condição para a chave específica caso ela exista. Desta forma, garantimos que apenas são acordadas as threads necessárias, isto é, as threads que estejam associadas à keyCond, dado que acordar threads associadas a outras chaves do mapa, tornaria o programa ineficiente, uma vez que esperamos no método getWhen apenas uma atualização nos valores da chave keyCond, não sendo portanto necessário acordar todas as threads sempre que haja uma alteração num valor de uma chave do mapa global.

5 Conclusão

Este projeto permitiu implementar com sucesso um serviço de armazenamento de dados partilhado, acessível remotamente, cumprindo todos os requisitos especificados. Foram desenvolvidas funcionalidades como autenticação, operações de leitura e escrita simples e compostas e ainda leitura condicional, garantindo um sistema funcional e robusto.

A arquitetura baseada em sockets TCP e threads demonstrou eficiência na gestão de múltiplas conexões simultâneas, enquanto o uso de atomicidade em todas as operações assegurou consistência nos dados. A avaliação de desempenho mostrou que o sistema é escalável e apresenta boa resposta sob diferentes cargas de trabalho.

Este trabalho consolidou conhecimentos teóricos e práticos em sistemas distribuídos, destacando a importância de decisões arquiteturais bem fundamentadas na obtenção de um serviço eficiente e escalável.