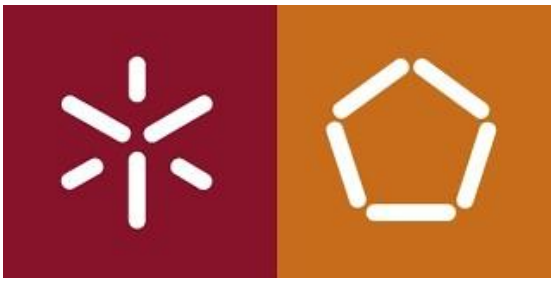
Universidade do Minho



Sistemas Distribuídos
Trabalho Prático

Armazenamento de dados em memória com acesso remoto

Ano letivo 2024/2025

Discentes:
a104526, Olavo Fernandes Malainho Santos Carreira
a104270, João Pedro Loureiro Pinto
a104272, Miguel Nogueira Barrocas
a104183, Diogo Alexandre Silva

Índice

Introdução

Este trabalho tem como objetivo a implementação de um serviço de armazenamento de dados partilhado, baseado no modelo cliente-servidor, onde a comunicação entre as partes é realizada através de sockets TCP. O sistema é projetado para armazenar e recuperar dados de forma eficiente, utilizando uma estrutura de armazenamento chave-valor em memória. O serviço deve ser capaz de atender múltiplos clientes simultaneamente, respeitando limitações de concorrência e garantindo operações atómicas.

As funcionalidades básicas do sistema são a autenticação e registo de utilizadores, as operações de leitura/escrita simples e compostas e também a capacidade de limitar o número de utilizadores concorrentes. Adicionalmente, o sistema oferece ainda operações avançadas, como a leitura condicional, que permite recuperar dados de forma bloqueante até que uma condição específica seja atendida. O serviço deve ainda suportar a operação de múltiplos pedidos concorrentes por parte dos clientes, sem que um pedido bloqueie os demais.

O desenvolvimento do projeto envolve a criação de um servidor capaz de gerir as conexões de vários clientes, tratando de forma eficiente as operações de armazenamento e consulta de dados. Para garantir a consistência dos dados e a integridade das operações, utilizam-se técnicas de sincronização adequadas para as várias operações de leitura e escrita.

Este relatório descreve a arquitetura e os principais componentes do sistema, as decisões de design adotadas e também os testes de desempenho realizados para avaliar a eficácia das soluções implementadas. O sistema foi desenvolvido utilizando a linguagem de programação Java, empregando sockets TCP e streams binários para a comunicação entre o cliente e o servidor.

Arquitetura do Sistema

Servidor

O servidor inicia um socket TCP na porta 11111, conhecida por todos os clientes. Quando um cliente tenta estabelecer uma conexão com o servidor, este aceita a ligação e cria uma nova thread para tratar dos pedidos do cliente de forma concorrente, sem bloquear o servidor para novos clientes. Esta abordagem garante que o servidor consegue atender múltiplos clientes simultaneamente.

O servidor utiliza várias estruturas de dados e serviços para gerir os pedidos e o estado do sistema. O **StorageService** é um dos principais componentes, sendo responsável por armazenar e recuperar os dados. Para garantir que as operações de leitura e escrita nos dados sejam realizadas de forma eficiente e sem conflitos, é utilizada a estrutura **ConcurrentHashMap**, que permite a manipulação concorrente das informações sem o risco de inconsistências. Além disso, o **ReentrantLock** é usado para controlar o acesso a recursos críticos que requerem sincronização mais fina, garantindo que, em determinadas situações, apenas uma thread possa aceder a uma parte específica da informação ao mesmo tempo.

Um dos principais desafios no servidor é a **gestão de concorrência** entre as threads que lidam com os pedidos. Para isso, o servidor utiliza semáforos (**Semaphore**) para limitar o número de clientes que podem ser atendidos simultaneamente, evitando sobrecarga no sistema. O uso da classe **Semaphore** é essencial para garantir que, mesmo com múltiplos clientes, o sistema mantenha um bom desempenho e não perca informações devido à competição por recursos.

Além disso, o **GetWhenService** é responsável por processar os commandos dos clientes que envolvem a leitura condicional. Para gerir esta comunicação de forma eficaz e segura, foi utilizado o **BlockingQueue**. Esta estrutura de dados permite que as threads bloqueiem e esperem por eventos específicos (neste caso esperar que o value da keycondition seja igual a keyvalue), facilitando o processamento assíncrono dos comandos. Também são usados locks adicionais para garantir que operações críticas, como a leitura e escrita de dados na fila, sejam realizadas de forma exclusiva quando necessário, evitando problemas de concorrência.

Ao longo de todo o processo, a sincronização e o controlo do acesso concorrente são fatores cruciais para garantir que o servidor funcione de forma estável e eficiente, atendendo a múltiplos clientes de forma confiável e sem falhas de integridade nos dados.

Cliente

O cliente conecta-se ao servidor através do socket aberto na porta “11111”. Após a conexão, é apresentado ao utilizador um menu inicial que permite escolher entre “Register” ou “Login”.
Depois de autenticado com sucesso, o utilizador acede ao programa em si, onde pode realizar todas as operações disponíveis. Estas funcionalidades consistem em enviar pedidos ao servidor, receber respostas e apresentar a informação ao utilizador.

Para suportar a comunicação com o servidor, o cliente utiliza threads que permitem a gestão simultânea de diferentes fluxos de informação. Uma thread é responsável por enviar pedidos ao servidor e processar as respetivas respostas. Outra thread fica constantemente à escuta de notificações enviadas pelo servidor, garantindo que o utilizador é informado em tempo real sobre eventos relevantes.

Comunicação entre Cliente e Servidor

No sistema desenvolvido, a comunicação entre o cliente e o servidor é realizada através de **sockets TCP**, que garantem a troca de mensagens entre as duas entidades. O cliente estabelece uma conexão com o servidor através de um **socket**, e a comunicação é feita de forma bidirecional utilizando **streams** de entrada e saída para garantir a troca de informações de maneira eficiente.
A comunicação entre o cliente e o servidor é implementada utilizando os fluxos de **entrada e saída** (DataInputStream e DataOutputStream). Estes fluxos são usados para enviar e receber dados através do socket. No lado do cliente, o **socket** é criado para conectar-se ao servidor em uma porta específica e garantir a comunicação via TCP/IP. Os dados são enviados de forma estruturada (por exemplo, como **strings** ou objetos serializados), dependendo da operação que o cliente deseja realizar.

->Comunicação do Cliente para o Servidor

Quando o cliente deseja realizar uma operação, como autenticação, registo ou interações com o serviço de chave-valor, ele envia dados ao servidor através do método output.writeUTF() da classe DataOutputStream. O cliente envia uma mensagem contendo as informações necessárias para a operação. Após isso, o cliente aguarda uma resposta do servidor.

O fluxo de comunicação é síncrono, ou seja, o cliente envia uma mensagem ao servidor e espera pela resposta antes de seguir com a execução. Por exemplo, ao tentar autenticar-se ou registar-se, o cliente envia os dados de login e aguarda a confirmação do servidor de que a operação foi realizada com sucesso. Dependendo da operação, o cliente poderá enviar pedidos subsequentes para realizar leituras ou escritas, através de outras interações com o servidor.

A comunicação entre o cliente e o servidor é **sincrónica** no lado do cliente. O cliente envia um pedido e aguarda a resposta do servidor antes de executar outras operações. Para gerenciar essa comunicação, o cliente utiliza a classe CommandLineInterface, que permite ao utilizador interagir com o sistema. Essa interface lê os comandos do utilizador e envia os pedidos ao servidor. O cliente só prossegue com a execução quando recebe a resposta do servidor, garantindo que a interação entre as duas partes ocorre de forma ordenada.

O cliente é projetado para lidar com operações de forma sequencial, esperando que cada resposta seja recebida antes de realizar novas solicitações.

->Comunicação do Servidor para o Cliente

Quando o servidor recebe uma solicitação do cliente, ele processa a informação e gera uma resposta, que é então enviada de volta ao cliente. Para isso, o servidor utiliza o fluxo de **saída** (DataOutputStream), onde escreve a resposta que deve ser enviada de volta ao cliente.

O servidor pode enviar vários tipos de respostas, dependendo da operação que foi realizada. Por exemplo, após o cliente enviar uma solicitação de leitura, o servidor envia de volta os dados solicitados. O cliente recebe a resposta através do método input.readUTF() da classe DataInputStream, que lê a informação vinda do servidor.

Funcionalidades

Autenticação e registo de utilizadores

O código implementa um serviço de autenticação que gerencia a comunicação entre cliente e servidor através de DataInputStream e DataOutputStream. Quando iniciado, o sistema apresenta ao utilizador duas opções: (1) Login ou (2) Registo. Para o login, o sistema solicita *username* e *password*, enviando estas credenciais ao servidor para validação. Se as credenciais estiverem incorretas, o utilizador pode tentar novamente ou, caso o utilizador não exista, tem a opção de se registar. Para o registo de novo utilizador, o sistema recolhe um novo username e password, que são armazenados

num ficheiro *CSV* denominado "users", permitindo a persistência dos dados de autenticação. Em ambos os casos, o servidor responde com mensagens indicando o resultado da operação, seja sucesso ou falha, permitindo ao utilizador prosseguir com a aplicação ou tentar novamente conforme necessário.

Put

Esta função implementa um comando "put" que permite armazenar dados numa estrutura chave-valor. A função aceita uma chave (String) e um valor (array de bytes) como parâmetros. Ao ser executada, envia ao servidor: o comando "put", a chave, o tamanho do valor em bytes e o próprio valor. O servidor processa o pedido e retorna uma mensagem de confirmação, que é apresentada ao utilizador. Em caso de perda de conexão durante a operação, a função lança uma IOException com uma mensagem descritiva do erro.

Get

Esta função implementa um comando "get" para recuperar dados armazenados. O cliente envia o comando "get" e a chave desejada ao servidor. Se a chave existir (resposta "Get_Ok"), a função lê o tamanho do valor e os bytes correspondentes, retornando-os. Se a chave não for encontrada (resposta "Get_Not_Found"), retorna null com uma mensagem apropriada. Para respostas inesperadas do servidor, lança uma IOException.

MultiPut

Esta função implementa um comando "multiput" que permite armazenar múltiplos pares chave-valor simultaneamente. Envia inicialmente o comando "multiput" e o número total de pares a serem inseridos. Em seguida, para cada par do Map, transmite a chave, o tamanho do valor em bytes e o próprio valor. Após processar todos os pares, o servidor retorna uma mensagem de confirmação que é apresentada ao utilizador.

MultiGet

Esta função implementa um comando "multiget" para recuperar múltiplos valores simultaneamente. Envia o comando e o número de chaves a recuperar, seguido das chaves individuais. O servidor responde com o número de resultados e, para cada chave, envia: a chave, o tamanho do valor (-1 se não encontrado) e o valor (se existir). Os resultados são armazenados num HashMap, onde chaves não encontradas são mapeadas para null. O HashMap final com todos os pares chave-valor é retornado.

GetWhen

Esta função implementa um comando "getwhen" que permite uma recuperação condicional de dados.

O cliente envia o comando com três parâmetros: a chave a recuperar, uma chave de condição e um valor de condição. O servidor aguarda até que a condição seja satisfeita (isto é, até que a chave de condição tenha o valor especificado) ou até expirar o tempo limite de 60 segundos. A resposta do servidor pode ser: "GetWhen_Ok" (retornando o valor pedido), "GetWhen_Not_Found" (chave não encontrada) ou "GetWhen_Timeout" (tempo limite excedido). Em caso de resposta desconhecida, lança uma IOException.

Testes e desempenho

Para testarmos o desempenho e tempo de resposta do programa fase as diferentes operações que executa usamos o profiller “VisualVM”. Vimos que o commando que mais tempo demora é o getWhen (media de 10 000 ms, enquanto que as outras operações demoram por volta de 1.5 ms). A significativa diferença no tempo de execução entre a operação getWhen e as demais operações (put, get, multiput e multiget) pode ser explicada pela natureza fundamentalmente distinta de seu funcionamento. Enquanto as operações put e get são operações síncronas que executam imediatamente, a operação getWhen implementa um padrão de espera condicional, necessitando aguardar até que uma condição específica seja satisfeita no sistema.

O tempo médio de 10000ms para a operação getWhen reflete seu comportamento bloqueante, onde a thread do cliente permanece em espera até que a condição especificada seja atendida ou até que ocorra um timeout. Este comportamento é intencional e necessário para implementar a funcionalidade de monitoramento em tempo real de mudanças no sistema. Em contraste, as operações put e get, com tempos médios de execução de 1ms ou menos, são operações diretas que acessam e modificam o armazenamento imediatamente.

Este comportamento é particularmente útil em casos onde o cliente precisa sincronizar suas operações com eventos específicos no sistema, mesmo que isso implique em maiores tempos de espera. A implementação atual balanceia efetivamente a necessidade de monitoramento em tempo real com um mecanismo de timeout que evita bloqueios indefinidos.

Outra parte do código cujo o funcionamento também se demonstrou lento foi a fase de login e registo. No caso do registro de novos utilizadores, o sistema necessita realizar operações de I/O em disco para persistir as credenciais no arquivo CSV (users.csv). A escrita em disco é uma operação significativamente mais lenta que operações em memória, contribuindo para o maior tempo de processamento. Adicionalmente, o sistema implementa mecanismos de sincronização para garantir a integridade dos dados durante o registro de múltiplos utilizadores em simultâneo.
Para o processo de login, o sistema precisa verificar as credenciais fornecidas contra o banco de dados de users. Embora o UserService mantenha um ConcurrentHashMap em memória para otimizar o acesso aos dados dos users, o processo ainda envolve verificações de segurança e operações de sincronização que aumentam o tempo de processamento.

A latência adicional nestes processos é um trade-off aceitável, considerando que a segurança e integridade dos dados de autenticação são prioridades críticas do sistema. Além disso, como estas operações são realizadas com menor frequência comparadas às operações regulares de armazenamento, seu impacto no desempenho geral do sistema é limitado.

Esta característica do sistema reflete uma decisão de design que prioriza a segurança e consistência dos dados de autenticação sobre a velocidade de execução, uma abordagem comum em sistemas que lidam com credenciais de usuário e controle de acesso.

No contexto do sistema de conexão cliente-servidor desenvolvido, a arquitetura de threads apresenta uma configuração específica e dinâmica. Do lado do cliente, o sistema opera com um total de 13 threads, sendo 12 delas configuradas como daemon e uma thread principal. No servidor, a estrutura é ligeiramente diferente, com 15 threads no total, das quais 13 são daemon.

Um aspecto fundamental desta arquitetura é o comportamento dinâmico de criação de threads: a cada nova conexão de cliente, uma thread daemon adicional é automaticamente gerada no servidor. Esta característica permite uma escalabilidade significativa e um tratamento individualizado para cada conexão de cliente.

A predominância de threads daemon na arquitetura indica uma estratégia de otimização de recursos, proporcionando flexibilidade para encerramento automático de threads secundárias e capacidade de gerenciar múltiplas conexões simultâneas de forma eficiente. Este design demonstra uma abordagem robusta para o gerenciamento de comunicações cliente-servidor, priorizando concorrência e eficiência computacional.

A distribuição não uniforme de threads (13 no cliente, 15 no servidor) e a criação dinâmica de threads daemon para novos clientes sugerem um sistema adaptável e dimensionável, capaz de responder rapidamente a variações na carga de conexões.

Conclusão

Em conclusão, o desenvolvimento deste sistema cliente-servidor de armazenamento chave-valor demonstrou ser um projeto bem-sucedido, implementando funcionalidades essenciais de forma robusta e segura. A arquitetura multithreaded, combinada com mecanismos de controle de concorrência e autenticação de utilizadores, permitiu criar uma solução escalável e confiável para armazenamento distribuído de dados.

Como trabalho futuro, o nosso grupo acredita que o desenvolvimento de uma interface gráfica (GUI), poderia ter sido desenvolvida, para facilitar os testes realizados e ser de mais fácil compreensão o uso do programa.

Apesar do ponto de melhoria identificado, o grupo está satisfeito com o trabalho realizado. O sistema atual atende aos requisitos iniciais propostos, demonstrando estabilidade e eficiência nas operações de armazenamento e recuperação de dados, enquanto mantém um bom nível de segurança através do sistema de autenticação implementado. A experiência adquirida durante o desenvolvimento proporcionou um valioso aprendizado sobre sistemas distribuídos, concorrência e desenvolvimento de aplicações cliente-servidor em Java.