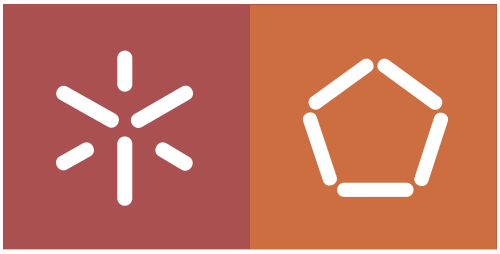


Universidade do Minho
Escola de Engenharia

Sistemas Distribuídos

Trabalho Prático

Armazenamento de dados em memória com acesso remoto

Grupo 28

Gonçalo Alves a104079
João Cunha a104611
João Sá a104612

Indíce

1. Introdução

O armazenamento de dados eficiente e escalável é um dos desafios mais proeminentes enfrentados por muitos sistemas distribuídos. O objetivo deste trabalho prático é implementar um sistema de armazenamento de dados em memória que pode ser acessado por vários clientes remotamente.

Este sistema é baseado na arquitetura cliente-servidor, no qual os clientes comunicam com o servidor através de ***sockets* TCP**, realizando operações com pares *key-value*. Além disso, devem ser tomadas medidas para minimizar a contenção de recursos e garantir um bom desempenho por meios de controlo de concorrência e limite de conexões simultâneas.

De modo a enriquecer o programa, foram desenvolvidas funcionalidades tais como, mecanismos condicionais de acesso a dados e operações compostas de leitura/escrita.

2. Arquitetura e Design do Sistema

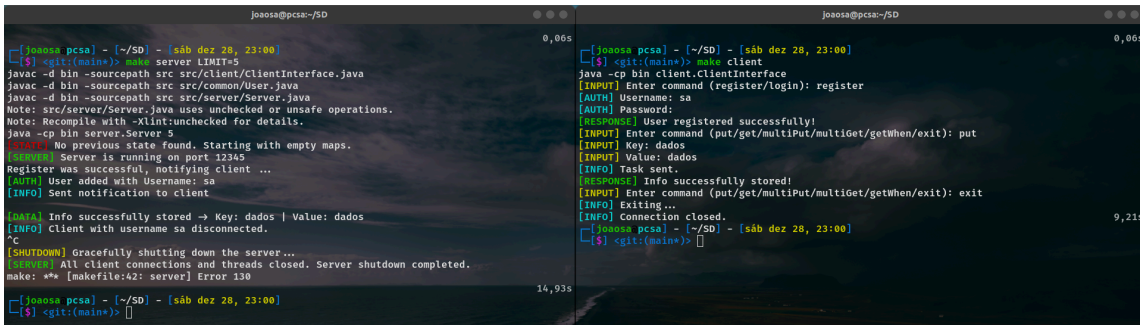
Tal como referido anteriormente, o sistema segue uma arquitetura cliente-servidor, na qual os clientes interagem com o servidor com o intuito de realizar operações sobre dados armazenados em memória. De modo a estabelecer comunicação entre os clientes e o servidor, foram utilizados *sockets* TCP, garantindo uma comunicação fiável e eficiente. Como pontos principais do sistema temos:

- **Cliente**: É responsável por enviar pedidos ao servidor (estes podem ser autenticação, escrita e leitura de dados)
- **Servidor**: É responsável por processar os pedidos (enviados pelos clientes), gerir a conexão concorrente de clientes e armazenar os dados, em memória. Este armazenamento consiste num mapa *key-value* que é serializado, quando o servidor é encerrado. Esta serialização permite o acesso a dados que foram introduzidos numa sessão anterior.

2.1. Comunicação cliente-servidor

A comunicação entre os clientes e o servidor, é efetuada através de mensagens em formato binário, utilizando as classes *DataInputStream* e *DataOutputStream*.

A mensagem enviada é transformada e enviada no formato binário. O servidor recebe e "traduz" a mensagem e faz o seu devido processamento. Por exemplo, a tarefa "put":



1. O cliente introduz a *key* e o *value* que deseja guardar
2. É escrito e enviado um pacote com a estrutura ["put"| key | value_length | value]
3. O servidor lê do *input stream* e retira a *key* e o *value* e guarda no *Map* esse par.
4. Por fim, envia a mensagem *"Info successfully stored!"* para o cliente, de modo a confirmar o sucesso da operação

2.2. Controlo de concorrência

Para lidar com múltiplos clientes em simultâneo, o servidor cria uma *thread* dedicada a cada cliente. De modo a evitar problemas de sobrecarga, foi estabelecido um limite de conexões . Quando este limite é atingido, a nova conexão é colocada numa fila de espera. Para evitar inconsistência nos dados, as operações de escrita e leitura no *Map* são sincronizadas. Ou seja, se um cliente introduzir um valor, um outro cliente consegue obter essa valor, mesmo estando numa *thread* diferente.

3. Funcionalidades implementadas

3.1. Put

O comando **Put** tem como objetivo armazenar uma *key* e um *value* no armazenamento de dados (Map). Esta operação é efetuada de forma segura, devido à utilização de um *lock*. Este *lock* serve para evitar que duas *threads* acedam ao mesmo tempo ao Map, originando *race conditions*. Os valores são obtidos e são adicionados (caso já exista uma *key* com o mesmo valor, a operação é cancelada).Após este processo, o *lock* é libertado e "acorda" as outras *threads*. Por fim, o cliente obtém (ou não) uma mensagem de confirmação.

3.2. Multi Put

O comando **Multi Put**, tal como o nome indica, é um variação do comando *put*. Em primeiro lugar, começa por obter o número de pares que o cliente pretende introduzir. O processo de bloqueio e inserção é análogo ao comando *put*. Após a inserção de todos os pares, a mensagem de sucesso é enviada ao cliente. Caso uma chave já exista, esse par é ignorado.

3.3. Get

O comando **Get** recebe uma *key* e tenta obter o valor associado à mesma no Map. Tal como nos outros comandos, de forma a evitar *race conditions* existe um *lock* a bloquear o acesso das *threads*. Se a chave existir, o *value* e o seu comprimento (valor que irá servir de referência para o cliente ler, de uma maneira correta, todos os dados) são enviados. Caso a *key* não exista, é enviada uma mensagem a informar sobre a ausência dos dados.

3.4. Multi Get

Tal como no caso do comando *Multi Put*, este comando é uma variação do *Get*. O processo é semelhante, começa por obter o número de pares desejado e processa as *keys* enviadas pelo cliente. Se a *key* existir, o seu *value* é adicionado a um mapa (*pairs*). Este processo de inserção no novo mapa, é protegido através de um *lock*. Por outro lado, caso a *key* não exista, é enviada uma mensagem a informar sobre a ausência de dados.

3.5. Get When

O comando **getWhen**, tal como o *get*, recupera um *value* associado a uma *key*, porém, apenas, após verificar uma condição (com base noutro par <*key,value*>. Se a condição não se verificar, a *thread* "bloqueia" até ser "acordada". Quando esta condição se verificar, o *value* é devolvido. Caso não exista, tal como no comando original, é enviada uma mensagem a informar sobre a ausência de dados.

4. Testes e Resultados

De modo a avaliar o desempenho do sistema, desenvolvemos três conjuntos de testes distintos, nos quais os utilizadores inserem 15 valores. 5 valores pequenos (10 caracteres), 5 médios (100 caracteres) e 5 grandes (1000 caracteres).

1. **Single User PUT**: Um utilizador faz o seu registo, insere os valores e mede o tempo da execução
2. **Single User GET**: Um utilizador faz o seu registo e insere os valores. Após a inserção, faz login e faz o pedido de requisição dos dados, medindo o tempo de leitura para estes.
3. **Multi Users PUT**: Vários utilizadores fazem o registo, e inserem os valores, simulando um cenário de concorrência.

Os testes foram estruturados de uma forma semelhante para permitir comparar os dados. Os tempos obtidos foram analisados para identificar e tentar encontrar padrões no desempenho, diferenças entre operações de leitura/escrita e diferença no desempenho consoante tamanho de dados e número de utilizadores.

Para além de apresentrar os valores, os scripts geram ficheiros *JSON* para facilitar a análise e tratamento dos dados.

4.1. Primeiro Cenário

No primeiro cenário, fizemos 4 testes para obter uma medição mais precisa do tempo de execução. O valor que foi obtido neste teste foi de **822.5 milissegundos**. Este valor vai de encontro às nossas expectativas visto que as operações *put* em *Maps* são rápidas.

4.2. Segundo Cenário

No segundo cenário, fizemos (novamente) 4 testes. A média dos valores obtidos foi **1,44 segundos**. Novamente, este valor era esperado. A operação de *get* vai procurar no *Map* comparando a key, o que leva mais tempo. A diferença entre as duas tarefas não é muito notória, apenas de alguns milissegundos.

4.3. Terceiro Cenário

No último cenário, decidimos criar vários utilizadores em paralelo. Fizemos 4 medições para 30 utilizadores, ao mesmo tempo, que efetuam a tarefa de *put* do primeiro cenário. Em comparação ao valor obtido no primeiro cenário, o valor nesta situação aumentou, atingindo os **2.42 segundos**. Este valor tende a aumentar, chegando aos **5.00 segundos** com 60 clientes em simultâneo.

5. Conclusões

Este sistema alcançou, na totalidade, os objetivos propostos, oferecendo uma solução eficiente e confiável na gestão de dados no formato *key-value*. As funcionalidades básicas implementadas são a autenticação de utilizadores, operações básicas e compostas de leitura/escrita e mecanismos de acesso a dados condicional.

De todos as vertentes abordadas ao longo do trabalho, é destacada a importância do uso de *sockets* TCP de modo a garantir uma comunicação confiável entre clientes e servidor. Em comparação a outros sockets, utilizados em trabalhos práticos de outras UC's, o *socket* TCP já possui todos as ferramentas para garantir a fiabilidade da comunicação, simplificando o desenvolvimento do sistema.

Como trabalho futuro, surgiu a ideia de desenvolver uma aplicação (desktop ou web) de modo a garantir uma interface mais *user friendly*, facilitando a utilização do sistema, alcançando mais utilizadores. No entanto, o suporte a clientes *multi-threaded* seria a prioridade número um.