Universidade do Minho
Escola de Engenharia

Sistemas Distribuídos

Licenciatura em Engenharia Informática

Ano Letivo de 2024/2025

Relatório de Desenvolvimento

Grupo 12

Diogo Gabriel Lopes Miranda (a100839)

João Ricardo Ribeiro Rodrigues (a100598)

Sandra Fabiana Pires Cerqueira (a100681)

Dezembro, 2024

SD

Índice

1 Introdução

Este trabalho foi realizado para a unidade curricular de Sistemas Distribuídos, onde nos foi proposto desenvolver um serviço de armazenamento de dados partilhado, implementando um sistema cliente-servidor com dados armazenados em memória e acedidos de forma concorrente via TCP. Os clientes interagem com o servidor através de uma interface chave-valor, permitindo operações de inserção e consulta de dados. A solução desenvolvida foca-se em estratégias que reduzem a contenção e o número de *threads* acordadas, focando na eficiência e escalabilidade. Além das funcionalidades básicas de autenticação, leitura e escrita de dados, o projeto suporta operações avançadas, como leituras condicionais. Este relatório descreve a arquitetura do nosso sistema, os protocolos de comunicação utilizados e as principais decisões tomadas. Também apresenta testes de desempenho e uma análise crítica dos resultados, destacando desafios enfrentados e possíveis melhorias futuras.

2 Arquitetura

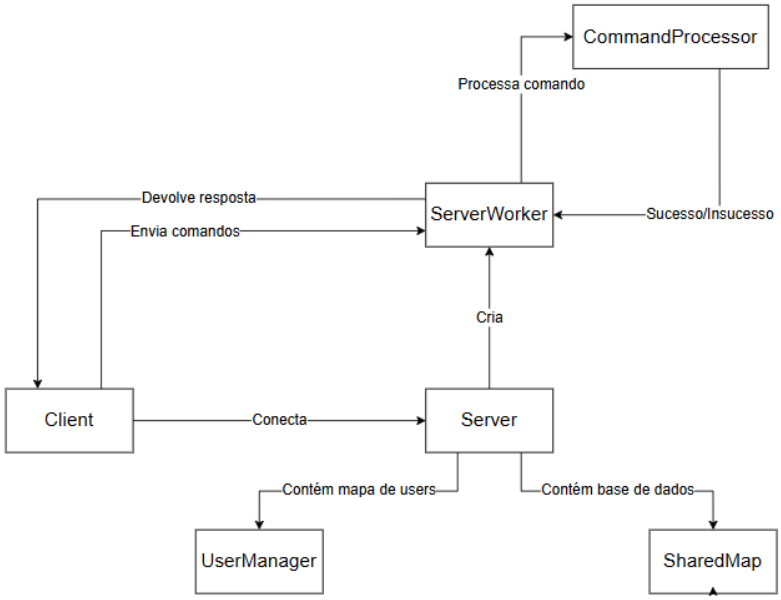

Figura 1-Esquema geral da arquitetura

2.1 Server

O *Server* é responsável por iniciar o *server socket* que irá receber os pedidos de conexão dos Clientes. Este possui um número limite, *S*, (definido numa variável da classe) de Clientes que podem estar conectados em simultâneo. Caso o limite ainda não tenha sido atingido a conexão do Cliente é aceite e é criada uma nova *thread* da classe *ServerWorker,* que é responsável pela comunicação com o Cliente.

2.1.1 ServerWorker

Após uma conexão ser aceite uma *thread* da classe *ServerWorker* é criada. Esta classe é responsável para comunicação com o Cliente, recebendo as suas mensagens e enviando as respostas. Cada mensagem recebida é desserializada (é esperado um determinado tipo de mensagem, definido em *MessageType)* e, caso não seja nula, é passada para o *CommandProcessor*. Após o término do recebimento de comandos, seja porque o *socket* fechou, a *thread* foi interrompida ou a mensagem é nula, o *socket* é encerrado pelo *ServerWorker*.

2.1.2 CommandProcessor

A mensagem, após ser desserializado, é passado para o *CommandProcessor*. Este divide a mensagem nas suas respetivas partes, comando e argumentos. Consoante o comando recebido é chamada o método responsável por executar o comando. No caso de comando relacionados com o user, *"Login"* e *"Register"*, irão ser executados métodos da classe *UserManager*. Para comandos relacionados com a base de dados, como *"Put"* e *"Get",* irão ser executados métodos da classe *SharedMap*.

2.1.3 UserManager

Contém o mapa com todos os clientes registados e os métodos para efetuar um registo e autenticação. A instância desta classe é partilhada por todas *threads*, de modo manter constantemente atualizado o mapa de clientes registados. Cada utilizador é representado por uma instância da classe *User,* onde é armazenado o nome e palavra-passe. Foi escolhido usar um *ReentrantLock*, ao invés de um *ReentrantReadWriteLock,* visto que o número de leituras verificado é superior ao número de escritas e por esse motivo a complexidade adicional do segundo não se relevou uma vantagem face ao primeiro.

2.2 Message

Representa a mensagem que, posteriormente, será serializada e enviada. Contém o identificador da mensagem, *id*, o tipo da mensagem, *type*, e o conteúdo da mensagem, *content*. O tipo da mensagem é representado pela classe *MessageType,* onde estão definidos os vários tipos de mensagens existentes, definidos a pensar na futura implementação do cliente-*multithread*.

2.2.1 MessageSerializer

Classe responsável por serializar/desserializar e enviar/receber as mensagens. No caso do método *serialize,* é passada à função, pelo *ServerSocket*, a mensagem para ser enviada, esta é serializada e em seguida enviada para o cliente, através do DataOutputStream recebido. No caso do método *deserialize*, o método recebe apenas o DataInputStream, do qual irá ler para receber uma mensagem, desserializá-la e retornar a mensagem desserializada.

2.1.4 SharedMap

Contém o mapa usado como base de dados. É com este mapa que todos os comandos irão interagir e, por esse motivo, os métodos que executam o comando e, portanto, envolvem acesso ao mapa, estão definidos nesta classe. Foi utilizado um *ReentrantReadWriteLock* neste contexto pois existem comandos que realizam exclusivamente leituras e, tendo em vista a escalabilidade e otimização dos comandos, desta forma são evitados bloqueios desnecessários sem comprometer a integridade dos dados.

2.3 Client

Classe responsável por simular um cliente, fazendo uso de outras classes existentes para as suas diversas funcionalidades. É responsável por estabelecer uma conexão com o servidor, através da porta definida, e instanciar as classes necessárias para processar o *input* do utilizador e enviar para o *ServerWorker* responsável pela conexão.
Através da classe *MenuManager* são apresentados ao utilizador dois menus, em momentos distintos, antes de efetuar login e após. O *input* do utilizador é captado pelo *MenuManager* e passado para a classe *InputProcessor,* onde irá ser feito o processamento do *input*.

No *InputProcessor* o input é transformado numa mensagem, com o tipo coincidente com o comando do input, e a mensagem é passada para a classe *ClientCommunicator,* onde será serializada ou deserializada pelo *MessageSerializer*, consoante necessário. No caso de a resposta do *ServerWorker* indicar que um *logout* foi realizado a conexão é encerrada.

2.4 Diagrama ilustrativo da execução de um Comando

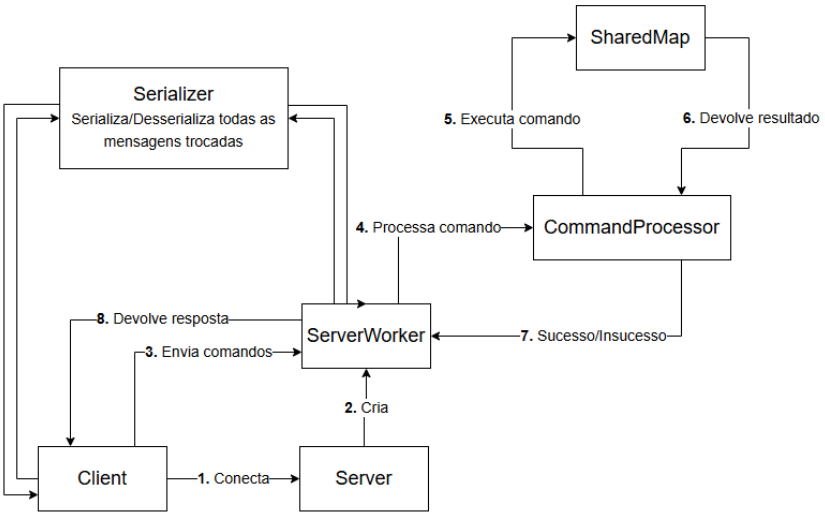

Figura 2- Ilustração simplificada da execução de um comando

3 Funcionalidades

3.1 Menu inicial , registo e autenticação de um user

Ao iniciar o programa, é apresentado um menu com opções de registo, autenticação e logout.

- ***Create Account**:* O utilizador fornece um nome de utilizador e uma password, que são enviados ao servidor para registo, se ainda não existirem.
- ***Log In***: O utilizador insere as suas credenciais, que são validadas pelo servidor antes de permitir o acesso.

3.2 Menu do user autenticado

Após autenticação, o utilizador acede a um menu com as operações que o cliente pode efetuar, como ***PUT, GET, MULTIPUT, MULTIGET*** *e* ***GETWHEN***.

3.3 Operações e Respostas

PUT: Envia um par key-value ao servidor para armazenamento. Se a *key* já existir, o valor é atualizado, caso contrário, é criada uma nova entrada.
GET: Solicita o valor associado a uma *key*. Se a key existir, o valor é retornado. caso contrário, é enviada uma mensagem de erro.
MultiPut: Permite inserir ou atualizar múltiplos pares *key-value* de forma atómica, ou seja, todas as alterações são efetuadas ou nenhuma.
***MultiGet**:* Permite solicitar os valores associados a várias keys de uma vez. O servidor retorna os pares correspondentes.
***GetWhen**:* Obtém o valor associado a uma key quando uma condição é satisfeita, bloqueando o pedido até que a condição seja cumprida.

4 Testes

O package Tests contém os testes que foram implementados para avaliar o funcionamento do projeto. Os primeiros testes *(testPut, testGet, testMultiPut, testMultiGet, testPutAndGet e testMultiPutAndMultiGet)* permitem verificar o funcionamento das operações básicas. O *testMix* simula cenários variados combinando diferentes comandos, *como Put, Get, MultiPut e MultiGet*, permitindo ver a capacidade do sistema de lidar com cargas mistas e operações intercaladas. O *testAcessoConcurrente* verifica o comportamento do sistema quando múltiplos clientes realizam operações Put simultâneas na mesma key. O *testReconnect* avalia a persistência de dados após desconexões, e o *testGetWhen* permite testar o correto funcionamento do getWhen. Por fim, o *testAdvancedOperations* combina todas as funcionalidades implementadas. Segue-se uma análise dos testes realizados.

4.1 Análise melhor número de clientes em paralelo

Para determinar o número ideal de clientes em paralelo para o nosso servidor, utilizámos os testes mencionados anteriormente, ajustando dois fatores principais: o número total de clientes nos testes e o número de clientes configurados para executar em paralelo no servidor. Variando estes parâmetros e usando uma máquina com as seguintes características: processador (11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz) , elaboramos a tabela que está no Anexo 1, através da análise da mesma fomos capazes de chegar às seguintes conclusões:

Quando o servidor está configurado para aceitar 5 clientes em paralelo, os testes apresentam uma baixa latência, mas o tempo total de execução aumenta à medida que o número total de clientes aumenta. Com 10 clientes em paralelo, o sistema mostrou um equilíbrio, uma vez que, tanto a latência média como o tempo total foram ligeiramente melhores em testes simples quando comparamos com uma configuração de 20 clientes em paralelo. No entanto, em testes mais complexos, como o testMix e o testAdvancedOperation, que simulam cenários concorrentes mais próximos do uso real porque temos várias operações misturadas, a configuração de 20 clientes em paralelo apresenta os melhores resultados, sendo melhor que as configurações de 5 e 10 clientes em termos de tempo total de execução e latência.

Concluímos que, apesar de configurações com 5 ou 10 clientes em paralelo serem mais eficientes para testes simples, a configuração com 20 clientes em paralelo mostrou-se mais robusta para lidar com operações mais exigentes e ambientes reais, onde múltiplas operações ocorrem simultaneamente.

4.2 Análise da escalabilidade

Como se pode observar na tabela, em cenários com 10 ou 100 clientes a tentar conectar-se, obtemos tempos de execução e latência reduzidos, mas com um aumento constante da latência. Quando o número de clientes aumenta para 500, tanto os tempos de execução quanto a latência sobem, e a latência atinge o seu ponto limite, mas permanecem aceitáveis para a maioria das operações. Para cenários com um número de clientes superior a 500 a latência mantém-se aproximadamente constante. No entanto, em testes avançados, como o *test*Mix o *testAdvancedOperations*, os tempos aumentam significativamente devido ao elevado número de comandos e à espera dos clientes pela satisfação das condições. Com a utilização de um *timeout* de 5 segundos que definimos para o getWhen, garantimos que nenhum cliente fica em espera infinitamente a impedir

que outros se conectem ao servidor. Concluímos assim, que o sistema apresenta latência baixa, o que demonstra que o processo de execução dos comandos está otimizado, e que esta aumenta de forma constante até aos 500 clientes e, a partir desse valor, tende a estabilizar.

5 Conclusão e Trabalho futuro

Este trabalho fez com que conseguíssemos explorar e por em prática os conceitos fundamentais de Sistemas Distribuidos, implementando um sistema cliente-servidor funcional e escalável. O servidor gere as conexões de clientes e delega a execução de comandos para *ServerWorkers*, enquanto a comunicação ocorre através de um protocolo binário. No método *GetWhen* usamos Conditions para acordar apenas as threads relevantes, otimizando os recursos. O cliente oferece uma interface funcional que suporta comandos como PUT, GET, MULTIPUT, MULTIGET e GETWHEN, sendo que os testes realizados demonstraram um desempenho consistente em diversos tipos de cenários. Embora a maior parte das funcionalidades tenha sido implementada com sucesso, gostaríamos de destacar como trabalho futuro o desenvolvimento do cliente multi-threaded. A ideia seria criar uma estrutura onde diferentes threads tratariam comandos específicos, coordenadas por uma thread principal que delegaria a execução dos pedidos, melhorando a eficiência em cenários de alta concorrência. No geral, consideramos o projeto bem-sucedido, pois consolidou os conhecimentos adquiridos e demonstrou a eficácia das estratégias aplicadas.

Anexos

Anexo 1

nº clientes	clientes em paralelo	testPutAndGet	testMultiPutAndMultiGet	testMix	testAcessoConcurrente	testReconnect
		tmp_total/latência_med	tmp_total/latência_med	tmp_total/latência_med	tmp_total/latência_med	tmp_total/latência_med
10	5	92 ms / 7.2333 ms	125 ms / 19.9333	119 ms / 6.66 ms	90 ms / 7.4666 ms	161 ms / 16.7666 ms
10	10	104 ms / 4.1 ms	100 ms / 3.03 ms	137 ms / 2.78 ms	115 ms / 3.2333 ms	146 ms / 7.9 ms
100	5	728 ms / 79.16 ms	359 ms / 42.3566 ms	487 ms / 33.966 ms	466 ms / 55.9066 ms	583 ms / 94.03 ms
100	10	461 ms / 43.7666 ms	359 ms /41.9533 ms	798 ms / 51.41 ms	497 ms / 58.8633 ms	632 ms / 122.5633 ms
100	20	500 ms / 60.18 ms	352 ms / 42.4933 ms	767 ms / 61.098 ms	643 ms / 76.4266 ms	653 ms / 122.7466 ms
500	5	1692 ms / 159.502 ms	853 ms / 117.14733 ms	1794 ms / 119.7728 ms	1129 ms / 111.344 ms	1689 ms / 258.502 ms
500	10	1316 ms / 170.9433 ms	1008 ms / 137.756 ms	1532 ms / 134.1334 ms	1027 ms / 109.4571 ms	1275 ms / 218.6893 ms
500	20	884 ms / 89.6646 ms	693 ms / 80.7873 ms	1233 ms / 71.5412 ms	1701 ms / 209.3273 ms	2318 ms / 400.1546 ms
1000	5	1161 ms / 173.121 ms	1250 ms / 155.7216 ms	1775 ms / 99.8874 ms	1699 ms / 134.1963 ms	2200 ms / 395.3436 ms
1000	10	1511 ms / 133.3013 ms	979 ms / 84.0796 ms	2123 ms / 149.9352 ms	1603 ms / 142.856 ms	2225 ms / 312.7373 ms
1000	20	1930 ms / 197.6551 ms	1270 ms / 62.956 ms	1951 ms / 132.5004 ms	1439 ms / 75.2746 ms	2556 ms / 377.8953 ms

testGetWhen	testAdvancedOperations	testPut	testGet	testMultiPut	testMultiGet
tmp_total/latência_med	tmp_total/latência_med	tmp_total/latência_med	tmp_total/latência_med	tmp_total/latência_med	tmp_total/latência_med
5113 ms / 7.8 ms	10106 ms / 299.8236 ms	121 ms / 15.8 ms	123 ms / 15.3 ms	141 ms / 23.8 ms	102 ms / 14.25 ms
5108 ms / 5.55 ms	10119 ms / 1.6538 ms	205 ms / 12.35 ms	179 ms / 8.35 ms	172 ms / 7.75 ms	204 ms / 6.2 ms
25375 ms / 4684.5622 ms	135490 ms / 7379.6588 ms	468 ms / 87.175 ms	302 ms / 67.175 ms	251 ms / 48.83 ms	355 ms / 68.22 ms
5352 ms / 1134.4677 ms	70366 ms / 4411.7463 ms	582 ms / 86.81 ms	400 ms / 70.52 ms	388 ms / 59.08 ms	677 ms / 146.995 ms
5354 ms / 846.2537 ms	40423 ms / 1494.4201 ms	767 ms / 182.27 ms	575 ms / 80.88 ms	566 ms / 151.18 ms	816 ms / 215.37 ms
80862 ms / 6525.972 ms	636895 ms / 49144.7736 ms	1230 ms / 207.407 ms	865 ms / 170.236 ms	878 ms / 209.967 ms	810 ms / 145.199 ms
40761 ms / 4503.7352 ms	306098 ms / 23366.1177 ms	1505 ms / 214.912 ms	888ms / 222.078 ms	1301 ms / 288.6361 ms	1128ms / 167.6283 ms
30671 ms / 3173.32 ms	165911 ms / 11702.4723 ms	1463 ms / 352.019 ms	1433 ms / 247.927 ms	877 ms / 100.419 ms	1214 ms / 195.95 ms
266365 ms / 34454.1034 ms	1217957 ms / 96904.5 ms	2810 ms / 357.156 ms	1616 ms / 325.6068 ms	1653 ms / 227.7817 ms	1677 ms / 316.9585 ms
165934 ms / 26116.4517 ms	621489 ms / 48989.4262 ms	2127 ms / 328.7474 ms	1915 ms / 282.5323 ms	1881 ms /235.5122 ms	1633 ms / 195.2885 ms
75993 ms / 10568.1981 ms	331046 ms / 23501.2622 ms	2027 ms / 263.9815 ms	1669 ms/ 261.7403 ms	1277 ms / 144.1416 ms	1855 ms / 438.0349 ms

Figura 3 - Tabela Resultados Testes