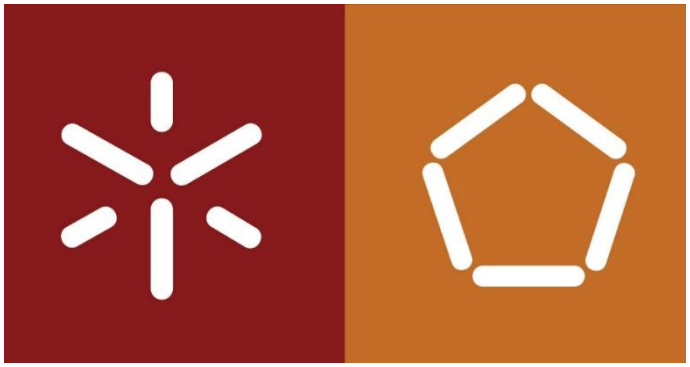
Universidade do Minho



Engenharia Informática

Trabalho Prático

Sistemas Distribuídos

Armazenamento de dados em memória com acesso remoto

Ano letivo 2024/2025

Grupo 20

Diogo Costa a95147,

Gonçalo Cunha a104003,

Nuno Ribeiro a104177,

Gonçalo Cruz a104346

Índice

Introdução

Este trabalho foi realizado no âmbito da unidade curricular de Sistemas Distribuídos e tem como objetivo a implementação de um serviço de armazenamento de dados partilhado. O sistema foi desenvolvido com um modelo cliente-servidor, em *Java*, utilizando *sockets* e *threads* para comunicação e processamento.

O serviço permite gerir dados, no formato chave-valor, e armazená-los em memória no servidor, bem como o acesso aos mesmos por parte de múltiplos clientes. O servidor é capaz de limitar o número de sessões de clientes ativas e atender a pedidos simultaneamente, o que otimiza a utilização dos recursos. Foram implementadas, também, algumas funcionalidades avançadas, como o suporte para clientes *multi-threaded* e a leitura condicional de dados.

O desenvolvimento deste projeto focou-se na eficiência e escalabilidade e foram adotadas estratégias para reduzir a contenção e otimizar o desempenho. O relatório descreverá a arquitetura, os protocolos utilizados e as decisões de design que suportam as funcionalidades implementadas.

Este projeto teve como objetivo desenvolver uma solução prática e funcional e procurou responder às necessidades do sistema. O presente relatório descreverá a estrutura da arquitetura, os protocolos que foram utilizados e as razões que levaram às principais decisões.

Arquitetura da aplicação

Servidor

O servidor implementado utiliza um *socket TCP* com o número de porta "12345", que é conhecido por todos os clientes. A cada conexão estabelecida por um cliente, o servidor cria uma nova *thread* para ser utilizada por ele ou coloca-o na fila de espera, caso o número máximo de sessões simultâneas tenha sido atingido.

Para garantir que o número de conexões simultâneas não exceda o limite definido (no nosso caso, 2), o servidor utiliza, como referido, uma fila de espera. Cada cliente conectado é atribuído a uma *thread* e a sincronização é garantida através de um *ReentrantLock*. Quando uma sessão é finalizada, o servidor verifica a fila de espera e inicia a próxima conexão disponível, se existir alguma.

A autenticação dos clientes é realizada através de um sistema de registo e login. A classe *ClientManager* armazena os utilizadores registados e as respetivas *passwords* num *HashMap*. Além disso, esta classe disponibiliza métodos para a leitura e a escrita dos dados armazenados em memória com acesso remoto. Para evitar concorrências nas *race conditions* (como o acesso à memória), são usados *locks*.

Após a autenticação, os clientes podem realizar operações de armazenamento e obtenção de dados, como *put*, *get*, *multiPut* e *multiGet*. Para as operações compostas (o *multiPut* e o *multiGet*), o servidor processa múltiplos pares chave-valor ou múltiplas chaves simultaneamente. Todos os dados são manipulados de forma segura através de *locks* que garantem a consistência do armazenamento.

A cada cliente autenticado o servidor atribui uma *thread*, que implementa a interface *Runnable*. Esta *thread* é responsável por interpretar as mensagens recebidas do cliente, executar as operações correspondentes ao armazenamento de dados e devolver as respostas apropriadas. Durante a execução, o *worker* também utiliza mecanismos de sincronização para evitar problemas de concorrência.

Sempre que uma sessão é encerrada, o servidor atualiza o contador de sessões ativas e verifica se há clientes na fila de espera. Caso existam, a próxima conexão é iniciada. Para isto, é usado o método *notifySessionEnd*.

A arquitetura do servidor foi projetada para que múltiplos clientes acedam a ele concorrentemente, com restrições controladas por *locks*, o que garante integridade e consistência dos dados.

Cliente

O cliente conecta-se ao servidor utilizando a porta "12345". Após a conexão, duas *threads* independentes são criadas: uma para escrita e outra para leitura. Este modelo *multi-threaded* garante que o cliente consegue realizar ambas as operações simultaneamente. A *thread* responsável pela leitura das mensagens do servidor aguarda notificações ou respostas e imprime-as no terminal do cliente. Para processar mensagens como o menu inicial é usado um *ReentrantLock* e uma *Condition* para sinalizar à *thread* de escrita que o cliente pode continuar. A *thread* de escrita espera que o servidor esteja pronto para receber entradas do cliente. Após o desbloqueio, lê comandos do utilizador através do terminal e envia-os ao servidor usando um *DataOutputStream*. Se o utilizador escrever "exit", a *thread* encerra a conexão com o servidor. A classe *Client* inclui métodos para controlar o estado do cliente, como *isEmEspera()*, que verifica o valor da *flag* do cliente de modo a ver se este está na lista de espera ou não. A *thread* de escrita utiliza um *join* para aguardar a sua conclusão, enquanto que a de leitura é interrompida, caso ainda esteja ativa, evitando possíveis *deadlocks*.

Comunicação Cliente-Servidor

O servidor utiliza um *ServerSocket* para receber novas conexões na porta definida e o cliente conecta-se à mesma utilizando um Socket. Assim que essa conexão é estabelecida, o servidor aceita-a e começa a gerir a interação com o cliente. Um aspeto importante da comunicação é o facto de o servidor apenas poder ter um certo número de conexões com clientes de cada vez, o que limita a quantidade de sessões concorrentes (MAX_SESSIONS = 2).

Quando um cliente se tenta conectar e o servidor já está com o número máximo de sessões ativas, é informado de que está na fila de espera e aguarda que um outro cliente se desconecte. Este comportamento é gerido recorrendo a uma *ReentrantLock* e uma *Queue* de espera, o que garante que a comunicação entre o cliente e o servidor seja coordenada de uma forma eficaz.

A comunicação entre o cliente e o servidor ocorre principalmente através de fluxos de entrada e saída de dados (*DataInputStream* e *DataOutputStream*). Quando o cliente se conecta, o servidor envia uma mensagem a informar sobre o estado da fila de espera ou que o cliente pode começar a interação.

Quando é permitida a interação a um cliente, o servidor envia uma mensagem a confirmar que ele pode iniciar a interação. Aí, a comunicação entre cliente e servidor é estabelecida de forma contínua, com o cliente a poder fazer pedidos.

Quando o cliente termina a sua interação, o servidor é notificado para libertar a sessão. A comunicação é então encerrada, com o *socket* a ser fechado após o fim da interação. Além disso, o servidor, ao detetar que o cliente terminou a sessão, verifica se há clientes na fila de espera para assumir a *slot* libertada.

A implementação da fila de espera permite que o servidor controle o fluxo de entrada dos clientes, garantindo que eles aguardem na fila até que uma *slot* esteja disponível. Além disso, o uso de *sockets* diferentes para cada cliente evita bloqueios mesmo que um dos clientes tenha um erro na execução.

Funcionalidades

Autenticação e registo de utilizador

Quando a aplicação cliente é inicializada, é estabelecida uma conexão ao servidor e, caso o cliente não vá diretamente para a lista de espera, são-lhe apresentadas as seguintes opções: **register** e **login**.

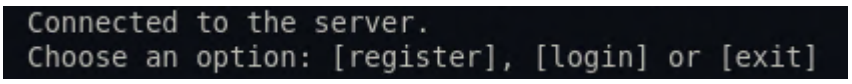


Figura 1 - Opções do menu inicial

No caso da opção **Register**, o utilizador insere o *username* e a *password* e o servidor verifica no *HashMap* **RegisteredUsers** se já existe alguma conta criada com o esse *username* e, em caso positivo, o servidor pede novamente o *username* e a *password*. Caso contrário, a conta do utilizador é criada.

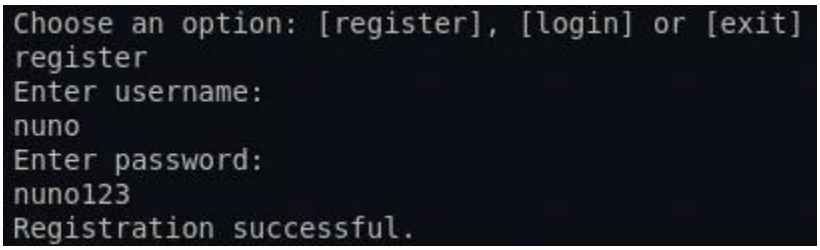


Figura 2 - Exemplo de registo de conta

Na opção **Login** o utilizador insere o seu *username* e a sua *password* e o servidor verifica se esse par *key-value* existe no *HashMap* **RegisteredUsers**. Em caso positivo, o utilizador fica autenticado. Caso contrário, o servidor pede que o utilizador insira novamente os dados.

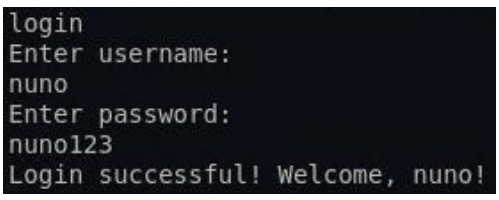


Figura 3 - Exemplo de Login

Pedidos do Utilizador

Após efetuar o **Login**, são apresentadas ao utilizador as seguintes opções: **put**, **get**, **multiput**, **multiget** e **getwhen**.

```
Choose an option: [put], [get], [multiput], [multiget], [getwhen] or [exit]
```

Figura 4 - Opções após Login

Na opção **put**, o utilizador escreve uma *key* e o valor associado à mesma, que é um *array* de *bytes*. O servidor guarda no *HashMap DataStorage* esse par *key-value*. Caso essa *key* já exista, o valor associado à mesma é modificado. O servidor confirma a operação com uma mensagem.

```
put
Write the key:
1
Write the data and finish with [END]
mensagem1END
Data stored successfully with key: 1
```

Figura 5 - Operação de Put

Quanto à operação **get**, o utilizador insere a *key* para a qual pretende obter o valor e o servidor envia o valor armazenado no mapa. Caso não exista essa *key* no mapa, o servidor avisa que essa chave não foi encontrada.

```
get
Write the key to retrieve:
1
Value for key 1:
mensagem1
```

Figura 6 - Operação de Get

Na operação **multiput**, o utilizador escreve quantos pares chave-valor deseja inserir e insere-os.

```
multiput
How many key-value pairs would you like to store?
2
Write the key for pair 1:
2
Write the data for key 2 and finish with [END]:
mensagem2END
Write the key for pair 2:
3
Write the data for key 3 and finish with [END]:
mensagem3END
Multiple data stored successfully.
```

Figura 7 - Operação de MultiPut

Já para a operação **multiget** o utilizador escreve quantas chaves vai querer introduzir e insere-as, de modo a obter os valores associados.

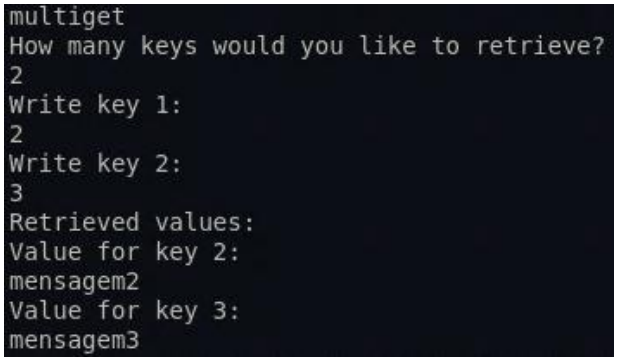


Figura 8 - Operação MultiGet

Por fim, na operação **getwhen**, o utilizador insere uma chave-condição, uma chave para a qual pretende obter o valor e um valor, que o servidor vai verificar se corresponde ao valor armazenado para a chave inicialmente inserida.

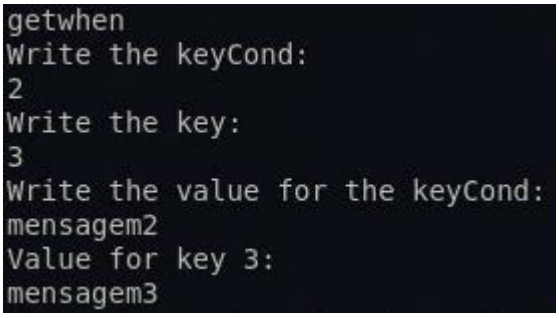


Figura 9 - Operação GetWhen

O utilizador pode, a qualquer momento, sair da aplicação, inserindo a opção **exit**.

Conclusão

Este projeto serviu para implementar um servidor capaz de gerir conexões de clientes com um limite de sessões simultâneas, recorrendo a uma fila de espera que garante que os clientes sejam atendidos por ordem. A comunicação entre os clientes e o servidor foi feita com *DataInputStream* e *DataOutputStream*, o que permite o envio de mensagens de estado, como por exemplo a indicação se um cliente está em lista de espera ou mão.

O uso de *ReentrantLocks* e *Conditions* garante que o servidor é capaz de controlar o número de sessões ativas sem causar bloqueios. De referir, ainda, que o objetivo proposto para este projeto foi cumprido.