

SISTEMAS DISTRIBUÍDOS

ENGENHARIA DE TELECOMUNICAÇÕES E
INFORMÁTICA



02

INTRODUÇÃO

- O presente relatório descreve o desenvolvimento de um serviço de armazenamento de dados em memória, acessível remotamente por meio de sockets TCP. O trabalho inclui autenticação de utilizadores, operações de leitura e escrita (simples e compostas) e um mecanismo de limitação de utilizadores concorrentes, com uma fila de espera FIFO. Além das funcionalidades básicas, foi implementada a (opção b) para garantir a ordem de início de sessão no sistema, de modo a que nenhum cliente fique indefinidamente à espera.

ARQUITETURA DO SISTEMA

- O sistema é composto por três camadas principais:

Servidor (Server.java):

- Mantém uma estrutura de dados em memória (um `Map<String, byte[]>`) para armazenar os pares chave-valor.
- Aceita conexões TCP de múltiplos clientes, lançando uma nova thread para cada um deles (`ClientHandler`).
- Controla a fila de espera, garantindo que no máximo `S` sessões são mantidas simultaneamente.

Biblioteca do Cliente (ClientLibrary.java):

- Fornece métodos que encapsulam as operações oferecidas pelo servidor (ex.: `put`, `get`, `multiPut`, `multiGet`, etc.).
- Implementa o protocolo de comunicação binário via `DataInputStream` e `DataOutputStream`.
- Suporta chamadas assíncronas em clientes multi-threaded, permitindo que múltiplas solicitações sejam enviadas sem bloqueio.

Interface do Utilizador (ClientInterface.java):

- Aplicação de consola que utiliza a biblioteca do cliente.

03

ARQUITETURA DO SISTEMA

- O sistema é composto por três camadas principais:

Servidor (Server.java):

- Mantém uma estrutura de dados em memória (um `Map<String, byte[]>`) para armazenar os pares chave-valor.
- Aceita conexões TCP de múltiplos clientes, lançando uma nova thread para cada um deles (`ClientHandler`).
- Controla a fila de espera, garantindo que no máximo `S` sessões são mantidas simultaneamente.

Biblioteca do Cliente (ClientLibrary.java):

- Fornece métodos que encapsulam as operações oferecidas pelo servidor (ex.: `put`, `get`, `multiPut`, `multiGet`, etc.).
- Implementa o protocolo de comunicação binário via `DataInputStream` e `DataOutputStream`.
- Suporta chamadas assíncronas em clientes multi-threaded, permitindo que múltiplas solicitações sejam enviadas sem bloqueio.

Interface do Utilizador (ClientInterface.java):

- Aplicação de consola que utiliza a biblioteca do cliente.
- Permite ao utilizador interagir com o servidor de forma simples, enviando comandos e recebendo respostas.
- A comunicação entre o cliente e o servidor segue uma abordagem request-reply sobre a mesma conexão TCP.

04

PROTOCOLO DE COMUNICAÇÃO

O protocolo adotado é binário e implementado com `Data[Input|Output]Stream`. A troca de mensagens respeita uma sequência pré-acordada de strings (UTF) e inteiros (inteiros enviados via `writeln/readln`) que definem o tipo de operação, parâmetros e resultados:

- **Autenticação:**
 - a. O servidor envia a mensagem “Insira o seu nome de utilizador:”.
 - b. O cliente envia o username (UTF).
 - c. O servidor solicita a palavra-passe, e o cliente a envia.
 - d. O servidor verifica a autenticação ou questiona se é necessário registo.
 - e. Caso o username e password sejam válidos ou registados com sucesso, o servidor avança para controlar a disponibilidade de sessão.
- **Operações (exemplos):**
 - **put:**
 - i. Cliente envia a string “put”.
 - ii. Envia a key (UTF) e o tamanho do valor em bytes (int), seguido dos bytes do valor.
 - **get:**
 - iii. Cliente envia “get”.
 - iv. Envia a key (UTF).
 - v. Servidor devolve um int (tamanho) e o valor (bytes).
- **Resposta:**
 - Em alguns comandos (como `put`, `multiPut`), o servidor envia uma string simples confirmando a conclusão (“Chave-valor inseridos/atualizados com sucesso.”).
 - Em `get` ou `multiGet`, o servidor envia o valor (ou conjunto de valores) na forma de bytes, precedidos por um int indicando o tamanho.

Este design binário permite uma comunicação mais eficiente do que envio de strings no formato texto plano, embora mantenhamos alguma simplicidade de parsing (via `readUTF`).

05

FUNCIONALIDADES IMPLEMENTADAS

Autenticação e Registo:

- **Servidor:** Armazena utilizadores num `Map<String, String>`, associando `username` a `password`.
- **Cliente:** Fornece métodos que, ao receber perguntas do servidor (ex.: “Deseja registar-se?”), podem responder dinamicamente (sim/não).

Operações de Escrita e Leitura:

- `put(String key, byte[] value)`:
 - Se a `key` não existir, cria uma nova entrada.
 - Se existir, atualiza o valor existente.
- `get(String key)`:
 - Retorna o valor em bytes ou null caso a `key` não exista.

Operações de Escrita e Leitura Compostas:

- `multiPut(Map<String, byte[]> pairs)`:
 - Várias chaves e valores são enviados e inseridos/atualizados atonicamente.
- `multiGet(Set<String> keys)`:
 - Retorna um map com todas as (chave, valor) correspondentes às chaves pedidas.

Limite de Utilizadores Concorrentes:

- O servidor recebe um parâmetro `S` (no caso, `MAX_SESSIONS = S`).
- Se o número de sessões ativas alcançar `S`, novas autenticações ficam em espera.
- É utilizado um mecanismo de fila FIFO com `wait()/notifyAll()` (ou no caso de Java, `await()/signalAll()`) dentro de um `Lock` e `Condition`.

06

FUNCIONALIDADES AVANÇADAS

Clientes Multi-threaded:

- A biblioteca do cliente (ClientLibrary) foi projetada para suportar chamadas concorrentes. Cada invocação, por padrão, escreve no socket e aguarda a resposta de forma independente. Em ambientes de thread pool, poder-se-iam empregar estratégias de futures ou callbacks para correlacionar cada resposta ao pedido original, embora nesta implementação básica seja feita sincronia imediata (bloco de envio e recepção).

Opção b) – Ordem de Início de Sessão e Evita Espera Indefinida:

- Para garantir que um cliente não fique para sempre bloqueado se novos clientes continuarem a chegar, foi construída uma fila de espera FIFO com listas ligadas. Desta forma, a ordem de chegada é respeitada (quem está no topo da fila é o próximo a obter vaga). Assim que um cliente sai, todos os que esperam são notificados e o primeiro na fila é atendido.

07

DECISÕES DE IMPLEMENTAÇÃO

Uso de synchronized ou ReentrantLock:

- Optou-se por ReentrantLock e Condition para controle fino sobre a fila de espera.

Estrutura de dados:

- O data store em memória (um HashMap) simplifica e agiliza o acesso, mas não garante persistência se o servidor cair.

Atendimento de cada cliente em thread independente:

- Facilita a escalabilidade com múltiplas conexões simultâneas.

Protocolo binário:

- Garante maior eficiência, seguindo as diretrizes de usar apenas DataInputStream/DataOutputStream

08

CENÁRIOS DE TESTE E RESULTADOS

Autenticação e Registro:

- Tentou-se autenticar com utilizador inexistente. Servidor pergunta se deseja registrar. Confirmada a eficácia do registro.
- Repetiu-se autenticação com username/password válidos.

Limite de Sessões (MAX_SESSIONS = 2):

- Conectaram-se três clientes simultaneamente. Verificou-se que o terceiro aguardou “Aguardando vaga... Você está na fila.” até um dos dois primeiros se desconectar.

Operações Simples (put e get):

- Verificou-se criação de novas chaves e atualização de valores.
- Efetuou-se leitura para chave inexistente, recebendo “Chave não encontrada.”.

Operações Compostas (multiPut e multiGet):

- Inseriu-se várias chaves de uma só vez e consultou-se o retorno atômico das mesmas.

Ordem FIFO:

- Executou-se testes em que clientes chegavam e saíam em diferentes ordens, garantindo que o primeiro a entrar em espera fosse o primeiro a obter sessão.

Os resultados mostram que o sistema funciona conforme esperado, respeitando o limite de sessões e a fila FIFO, além de atender as operações de escrita e leitura, simples e compostas.

09

CONCLUSÃO

O projeto atendeu aos requisitos de um serviço de armazenamento de dados em memória com acesso remoto via sockets TCP. A autenticação de utilizadores, a limitação de sessões ativas, as operações de leitura/escrita (simples e atômicas) e o suporte a clientes multi-threaded foram implementados com sucesso. A abordagem (opção b) de ordenação na fila de espera garante que nenhum cliente fique indefinidamente bloqueado.

Para trabalhos futuros, poder-se-ia adicionar:

- Persistência dos dados em disco para tolerância a falhas.
- Mecanismos de replicação e tolerância a falhas (cluster de servidores).
- Estratégias de escalonamento de threads para aumentar ainda mais a eficiência sob carga pesada.

Em suma, o sistema demonstrou boa performance nos cenários de teste propostos, atendendo aos objetivos definidos e evidenciando uma arquitetura robusta e escalável.