Sistemas Distribuídos — Trabalho Prático
Armazenamento de dados em memória com acesso remoto

Engenharia Informática
Universidade do Minho
2024/2025

André Barbosa Teixeira - a104001
Luís Enrique Dias Freitas - a104002
João Andrade Costa - a104258
Rui Pedro Pires de Sá Cerqueira - a104006

Introdução

No presente projeto foi desenvolvido um serviço de armazenamento de dados partilhado, em que a informação é mantida num servidor e acedida remotamente. Clientes interagem com o servidor através de sockets TCP, para depois inserir ou consultar informação. O servidor armazena a informação em memória, e atende então clientes concorrentemente através de threads.

A comunicação do cliente com o servidor é feita de ambas formas: síncrona e assíncrona - para inícios de sessão ou registos, a comunicação é síncrona, uma vez que o cliente tem de receber confirmação de autenticação antes de poder usar os serviços do programa. Para as restantes operação, é tudo feito de forma assíncrona, evitando o cliente ter de estar à espera pelo servidor a cada operação.

O presente projeto cumpre todas as funcionalidades básicas e os requisitos impostos, para além de implementar ambas as funcionalidades avançadas.

Protocolo de comunicação (Tagged Connection/Frame)

Antes de entrar em especificações de cliente ou servidor, é importante primeiro referir a forma como os dados são transmitidos entre os mesmos. Toda a comunicação de dados é feita em bytes, ocorrendo a devida serialização e desserialização quando é necessário a conversão de tipos.

Tagged Connection

Como foi indicado, o protocolo de comunicação deve ser em formato binário, podendo recorrer apenas a *Data Input/Output Stream*, e é nesta classe que é usado, sendo que o construtor de um objeto da classe recebe uma *Socket* e com esta define o *Input/Output Streams*. Após a construção, a classe fornece as operações *send*() e *receive*(), abstraindo o processo de comunicação para esta classe. Como em todas as outras classes, para controlar o acesso, são utilizados *Read/Write locks* - apesar da sua complexidade, serão benéficos uma vez que operações de leitura serão bem mais comuns que de escrita, acabando por serem feitos menos bloqueios desnecessários.

Retomando o foco de volta na operação *send*(), esta para além de receber um array de bytes para ser enviado, recebe também uma etiqueta - *tag* - na forma de um inteiro, para determinar o tipo de informação a ser enviada, permitindo depois ser interpretada de acordo. A função escreve então para a output stream a tag e, se o array de bytes não for

nulo, envia-o também, caso seja, anota o valor 0. Já a função *receive()* reconstroi os dados enviados pela anterior função, retornando um *Frame*.

Por fim, o método *close()* pode ser invocado, fechando a input e output streams.

Frame

Como mencionado anteriormente, um objeto desta classe é constituído por duas variáveis de instância, sendo uma, um inteiro para a *tag* e outra um array de bytes. Também tem um construtor que recebe estes mesmos tipos para inicializar as suas variáveis. É usado para condensar os dados e a respetiva etiqueta num só objeto.

Cliente

O cliente implementado permite a interação com o servidor de forma eficiente e estruturada, cobrindo operações essenciais como autenticação, armazenamento e consulta de dados.

Funcionamento Geral

O cliente conecta-se ao servidor através de uma `Socket`, utilizando a classe `TaggedConnection` para envio e receção de mensagens que contêm uma chave, e um valor binário. Após a conexão inicial, o servidor verifica e valida o cliente antes de permitir acesso às funcionalidades.

A interação com o utilizador é feita através de um menu que permite selecionar entre as opções de login, registo de novos utilizadores, e operações relacionadas ao armazenamento e consulta de dados no servidor.

Principais Funcionalidades

1. **Autenticação e Registo:**
    - O cliente envia credenciais para o servidor, que valida a informação e devolve um estado (sucesso ou erro).
    - Caso o utilizador deseje registar-se, os dados são enviados para o servidor, que verifica a unicidade do username antes de proceder.

2. **Operações de Manipulação de Dados:**
    - **Upload:** Envio de uma chave e respectivo conteúdo para armazenamento no servidor.

- **Download:** Solicitação de dados associados a uma chave específica.
- **Multiupload e Multidownload:** Suporte ao envio ou recuperação de múltiplos pares chave-valor em uma única operação, otimizando a comunicação.
- **Consultas Condicionais:** Operação avançada que permite a recuperação de dados apenas quando uma condição especificada pelo cliente é satisfeita.

3. **Comunicação Assíncrona:**
    - As operações não relacionadas à autenticação utilizam threads para processar respostas do servidor, permitindo que o cliente continue em execução sem bloquear a interação com o utilizador.

ServerItf/ServerOps

Antes de entrar na classe de servidor em si, é importante primeiro expor a interface que define os métodos que este vai utilizar para cumprir as funcionalidades do programa, e a classe que a implementa. A razão pela qual uma interface ter sido usada aqui foi para ter abstração total, retirando a dependência numa classe específica, só tendo para com a interface.

Começando com esta, apresenta todas as funções indicadas mais uma para adicionar um utilizador novo e outra para autenticar cada login. Para além destas estão presentes umas auxiliares que têm como objetivo transformar os dados de *byte[]* para o que as funções individuais pedem como tipos de entrada.

Em *ServerOps*, a classe que implementa esta interface, para além de definir os métodos da mesma, esta também criar os mapas para guardar os logins e os dados, uma vez que é tudo mantido em memória e no formato de chave-valor. para os logins, o mapa é do tipo *Map<String,String>*, sendo que a primeira String guarda o nome de cada utilizador, e o valor associado: a palavra passe para autenticação. Para os dados, o mapa é do formato *Map<String,byte[]>*, sendo a String a chave para aceder a conteúdo, e o array de bytes o conteúdo em si. Além disso, define também os locks a serem usados (*read/write*) e a variável de condição para a leitura condicional.

Server

A classe `Server` é responsável por aceitar conexões de clientes, gerenciar threads e delegar tarefas aos métodos definidos na `ServerOps`.

1. **Gerenciamento de Conexões:**
    - Utiliza um `ServerSocket` para aceitar conexões em uma porta fixa.

    - Limita o número de clientes simultâneos utilizando um contador atômico (`AtomicInteger`).
    - Envia mensagens ao cliente para confirmar a conexão ou informar que o limite de conexões foi atingido.
2. **Threads de Trabalhadores:**
    - Cada cliente aceito é atendido por um thread dedicado, gerenciado pela classe `Worker`.
    - O thread interpreta as mensagens recebidas, identifica o tipo de operação (usando tags) e chama os métodos correspondentes na `ServerOps`.

3. **Comunicação com o Cliente:**
    - Utiliza a classe `TaggedConnection` para enviar e receber mensagens em um formato padronizado (frames com tags e dados).
    - Implementa diferentes casos de operação, como login, upload, download, etc., enviando respostas ao cliente conforme necessário.
4. **Sincronização e Confiabilidade:**
    - Cada operação no `ServerOps` é protegida por locks para garantir consistência em ambientes multithread.
    - Em caso de erro ou desconexão, o servidor decrementar o contador de clientes e encerra a conexão de forma segura.

Conclusão

No processo de desenvolvimento deste trabalho construímos algo que engloba a matéria dada ao longo do semestre num produto final cumpridor dos requisitos impostos. Sistemas de armazenamento de dados são pilares em muitos projetos de informática, e o projeto ajudou na sua compreensão.

Apesar disso, é notável que foi feita uma versão simplificada, ocorrendo apenas armazenamento em memória, sem permanência fora isso, sendo um sistema real algo diferente e um pouco mais trabalhoso, todavia foram consolidados os conhecimentos cruciais neste tipo de sistema.