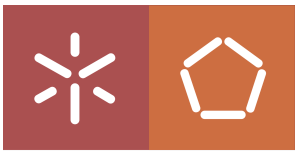

Universidade do Minho
Escola de Engenharia

Sistemas Distribuídos (2024/25): Relatório de Trabalho Prático

Licenciatura em Engenharia Informática, Dezembro 2024

-

Gustavo Barros A100656
Luís Figueiredo A100549
Enzo Vieira A98352
Pedro Ferreira A97646

1 Introdução

Como trabalho prático foi proposta a conceção, em Java, dum sistema servidor-cliente de armazenamento de dados. Os clientes interagem por operações de leitura (get/multiGet) e escrita (put/multiPut) em chaves solicitadas ao servidor.

1.1 Funcionalidades Básicas

1. *(satisfeito)* Autenticação e registo dum utilizador via nome e palavra passe.
2. *(satisfeito)* Operações de escrita e leitura simples.
3. *(satisfeito)* Operações de escrita e leitura compostas.
4. *(satisfeito)* Limite de utilizadores concorrentes.

1.2 Funcionalidades Avançadas

5. *(não satisfeito)* Suporte a clientes multi-thread.
6. *(não satisfeito)* Operação de leitura condicional.

1.3 Requisitos

- ➢ Implementação em Java.
- ➢ Programa servidor que aceite conexões concorrentes TCP de clientes via threads.
- ➢ Biblioteca de cliente que permita aceder, por socket TCP, ao servidor.
- ➢ Interface gráfica no cliente que permita interação com o servidor de forma abstraída à conexão.
- ➢ Conceção de vários cenários de teste com cargas de operação diferentes, para análise de escalabilidade.
- ➢ Permitida só uma única conexão entre o servidor e um cliente.
- ➢ Protocolo de comunicação em formato binário, possível recurso a `Data[Input|Output]Stream`.
- ➢ Servidor não pode ter múltiplos threads a escrever num único socket.
- ➢ Atomicidade em todas as operações.

2 Protocolo Aplicacional de Comunicação

A classe TaggedConnection abstrai a comunicação por socket entre o servidor e um cliente, sendo instanciada dada uma socket em argumento. Assim, constitui-se por um socket, byte streams de envio e receção, e locks para ser isolado o seu acesso. Os locks são necessários pois ambos os lados da conexão - cliente e servidor - precisam de acesso aos streams de dados.

```
public class TaggedConnection implements AutoCloseable {
    public static class Frame {
        public final int tag;
        public final Object data;
        public final Object optionalData;
    /***/
    }
    private final Socket socket;
    private final DataInputStream in;
    private final DataOutputStream out;
    private final ReentrantLock sendLock = new ReentrantLock();
    private final ReentrantLock receiveLock = new ReentrantLock();
```

No que diz respeito a métodos, aufere implementações atómicas de envio/receção de mensagens pelo socket, tanto para o cliente como o servidor. A payload é uniformizada com a classe interior Frame. Há vários overloads para os métodos `send*`, pois diferentes operações implicam envio de diferentes tipos de dados como argumento.

```
    public void sendToServer(int tag, ...) {/***/}
    public void sendToClient(int tag, ...) {/***/}
    public Frame receiveFromServer() {/***/}
    public Frame receiveFromClient() {/***/}
    public void close() {/***/}
}
```

O campo `tag` serve para dar contexto ao servidor sobre a mensagem enviada, facilitando a desmultiplexagem. O envio duma mensagem com certa tag implica uma tag igual numa potencial mensagem de resposta.

- 0 indica intenção de fecho da conexão.
- 1 indica registo de novo utilizador.
- 2 indica login de utilizador existente.
- 3 a 6 indica operações put, get, multiPut e multiGet respetivamente.

3 Cliente

A classe Client permite que um utilizador aceda ao servidor por um interface gráfico à base de menus. Estes menus são impressos via classe ClientView, que faz adicionalmente a captura de inputs do teclado necessários à navegação e inserção dos argumentos nas operações. Abaixo, uma simplificação comentada do método `main()`:

```
public class Client {
    static ClientView view = new ClientView();

    public static void main(String[] args) {
        try {
            Socket socket = new Socket(/***/);
            TaggedConnection c = new TaggedConnection(socket); // Estabelecer conexão
            boolean logged = login_or_register(c); // Comunicação para autenticar
            if (!logged) {/***/} // Autenticação falhou; terminar precocemente

            int option;
            do {
                option = view.printMainMenu(); // Retorna opção inserida
                switch (option) {
                case 1: /***/ // put
                case 2: // get
                    c.sendToServer(4, view.getKey()); // Obtém chave via teclado
                    TaggedConnection.Frame frame = c.receiveFromServer();
                    byte[] response = (byte[]) frame.data;
                    view.printMessage("Value received: " + new String(response));
                    break;
                case 3: /***/ // multiPut
                case 4: /***/ // multiGet
                case 5: /***/ // Exit: envia mensagem com tag=0
                }
            } while (option != 5); // Exit faz quebrar impressão do menu

            socket.close();
        } /***/
    }
}
```

A lógica das demais operações omitidas é análoga à do get, tirando que put/multiPut ignoram as respostas do servidor - só são úteis para a cronometragem realizada com clientes automatizados, abordados mais adiante.

A autenticação fica do encargo dos adicionais métodos `login_or_register()` e `handleUserAuthentication()` que, após inserção de nome e senha para registo ou login, enviam mensagem ao servidor com tag 1 ou 2 consoante a respetiva intenção. Se o servidor não aceitar a autenticação (se registo: por nome preexistente, se login: por senha errada), há um término precoce da execução do cliente.

4 Servidor

O servidor, implementado na classe Server, fica encarregue de aceitar cada novo cliente e, se houver vagas para sessão, inicializa nova instância ServerWorker concorrentemente. O máximo de vagas de sessão (S) é passado em argumento ao executar o servidor. Abaixo, uma simplificação comentada do método `main()`:

```
public class Server {
    private static int activeSessions = 0; // Nr sessões atuais (secção crítica)
    /***/

    public static void main(String[] args) {
       int S = Integer.parseInt(args[0]); // Arg dita limite máximo de sessões
       /***/
       try (ServerSocket ss = new ServerSocket(12345)) {
           SharedMemory memoria = new SharedMemory();

           while (true) { // Loop em que o servidor escuta por novos clientes
               Socket socket = ss.accept();
               lock.lock();
               try {
                   while (activeSessions >= S)
                       condition.await(); // Espera até que haja vaga p/ sessão
                   activeSessions++;      // Incrementa contador quando acorda
               }
               finally { lock.unlock(); }
               Thread t = new Thread(new ServerWorker(socket, memoria));
               t.start();
           }
       } /***/
    }
}
```

O campo `activeSessions` é um contador dos clientes atualmente em sessão. Como isto é informação atualizada por workers concorrentes, é protegida por um `lock` e uma `condition` que permite adormecer a escuta de novos clientes até que se volte a ter menos sessões que S. Sinalização de tal é feita logo após decrementação do contador via método `sessionEnded()`, invocado por um worker ao receber uma mensagem de tag 0 (i.e. intenção de fim de conexão, como visto atrás).

O campo de classe SharedMemory contém tanto os dados acessíveis aos clientes como também as credenciais de autenticação válidas (i.e. tudo o que é pertinente para os workers). É a partir dele que se aplicam as operações que os workers recebem, segundo a lógica padrão do problema produtor-consumidor.

4.1 Gestão de Contas de Utilizador

O armazenamento e teste de credenciais para autenticação fica ao encargo da classe UserAuthentication, uma base de dados encapsulada em SharedMemory. Segue a lógica padrão dum sistema de transações atómicas, pois sujeita-se ao acesso por vários workers.

```
public class UserAuthentication {
   // Mapa de utilizadores registados: username -> password
   HashMap<String, User> users = new HashMap<>();
   ReentrantLock lock = new ReentrantLock();
```

4.2 Server Worker

A existência de vaga para sessão suscita o instanciamento dum worker para um cliente conectado. Primeiro, é esperada do cliente uma mensagem contendo uma instância da classe User (composta por nome e senha) para registo ou login. Essa instância User é então avaliada por UserAuthentication, que não retorna sucesso se…

- ➢ … o nome recebido já existe, na intenção de registo.
- ➢ … o nome recebido existe mas a senha está incorreta, na intenção de login.
- ➢ … o nome recebido não existe, na intenção de login.

```
class ServerWorker implements Runnable {
    private final SharedMemory memoria;
    private TaggedConnection connection;
    /***/
    public void run() {
        System.out.println("Client connected");
        TaggedConnection.Frame frame; // Recebe instância User do cliente
        try { frame = connection.receiveFromClient(); } /***/
        boolean exit = false;
        switch (frame.tag) {
            case 0: // Exit
            case 1: /***/ // Register: regista instância se nome for novo
            case 2: /***/ // Login: testa a sua preexistência
        }
        if (exit) {/***/} // Termina worker (i.e. fecha sessão)
        System.out.println("User authenticated");
```

De seguida, esperam-se mensagens com tags de operação (3-6) ou saída (0). A receção duma mensagem de operação leva à execução do método correspondente na instância SharedMemory, pois é aqui que os dados são armazenados. Seguidamente, envia-se uma mensagem de resposta. As respostas a put/multiPut constituem-se por tag e corpo nulo, pois servem apenas para cronometrar o tempo que a operação levou a ser cumprida.

```
        TaggedConnection.Frame frame2;
        boolean exitTag = false;
        do { // Recebe operação do cliente
            try { frame2 = connection.receiveFromClient(); } /***/
            switch (frame2.tag) {
                case 0: // Exit
                case 3: // put
                case 4: // get
                    String getKey = (String) frame2.data;
                    System.out.println("get " + getKey);
                    byte[] value = memoria.get(getKey);
                    try { connection.sendToClient(4, value); } /***/
                    break;
                case 5: // multiPut
                    /***/
                    memoria.multiPut(pairs);
                    try { connection.sendToClient(5); } // Responder só tag
                    /***/
                case 6: // multiGet
            }
        } while (!exitTag);
        Server.sessionEnded(connection);
    }
}
```

5 Análise de Escalabilidade

Para análise da escalabilidade do sistema, foi desenvolvido o programa TestPlatform. Este executa Workloads, isto é, conjuntos de N clientes concorrentes que se conectam ao servidor para repetir certo número de vezes uma certa bateria de operações (ClientConfiguration) predefinida no ficheiro clientconfigs.json.

Cada cliente dum Workload termina com uma lista dos tempos de execução para cada operação enviada ao servidor. Consegue então calcular a sua velocidade média de operações/segundo. Isto permite à classe Workload determinar a mediana e média de velocidade de entre os vários no fim de todos terminarem. Foram definidos os seguintes workloads:

- smallput/mediumput/bigput: 10/50/100 clientes a enviar operações 100% put
- smallget/mediumget/bigget: 10/50/100 clientes a enviar operações 100% get
- smallhalves/mediumhalves/bighalves: 10/50/100 clientes a enviar operações 50% put, 50% get

Observe-se um excerto do output:

```
Workload smallput:
| 10 put_tester clients
| Latest results:
| | half the clients had avg speed below 7160.14 operations/s.
| | the avg speed throughout the clients was 7492.08 operations/s.
(...)
Workload bighalves:
| 50 put_tester clients
| 50 get_tester clients
| Latest results:
| | half the clients had avg speed below 575.72 operations/s.
| | the avg speed throughout the clients was 533.40 operations/s.
```

Foi construído o seguinte gráfico, em que os $v_{p,g,h}$ indicam as velocidades medianas para os workflows *put, *get, *halves de vários tamanhos n.