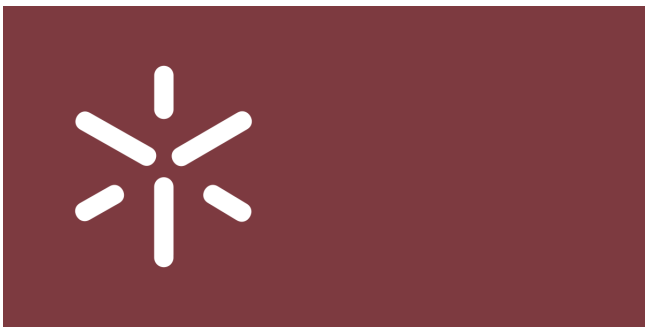

Universidade do Minho

Escola de Engenharia

Licenciatura em Engenharia Informática

Projeto da U.C. Sistemas Distribuídos

Ano letivo 2023/2024

Realizado por:

A91671 - João Silva
A91672 - Luís Ferreira
A91697 - Luís Vilas
A93258 - Bernardo Lima

Contents

1 Introdução

O presente relatório descreve o desenvolvimento de um sistema distribuído para a implementação de um serviço de **Function-as-a-Service** (**FaaS**). O projeto visa explorar a importância de **sistemas distribuídos** na execução eficiente de tarefas computacionais.

No contexto moderno da computação, a necessidade de um processamento distribuído tornou-se fundamental para solucionar cargas de trabalho intensivas e escaláveis. A abordagem de *Function-as-a-Service*, ou Função como Serviço, destaca-se como uma maneira eficaz de distribuir tarefas computacionais de maneira dinâmica e sob demanda.

O sistema desenvolvido utiliza a linguagem de programação `Java`, adotando `threads` e sockets `TCP` para a comunicação entre os diferentes componentes distribuídos.

2 Explicação das funcionalidades

O sistema foi projetado com uma arquitetura distribuída para otimizar o processamento de tarefas. A interação entre os principais componentes do sistema inclui clientes (`Client`), servidores de execução de tarefas (`Standalone Server`) e um servidor central (`Cloud Server`) para a correta distribuição de carga entre estes.

Os clientes, após o **registo** e o subsequente `login` são responsáveis por enviar **pedidos** ao sistema, dos quais se destacam a execução de **tarefas** onde são colocadas numa **fila de espera**. Estas poderão ser requisitadas através da **localização de um programa** na máquina ou através de um **ficheiro de** `input` e serão apresentadas quer na **consola**, quer em um **ficheiro de** `output` ou o **estado das tarefas** requisitadas. A fila gerencia as tarefas pendentes, priorizando-as e distribuindo-as aos servidores de execução conforme a **disponibilidade**.

3 Arquitetura do Sistema

3.1 Estruturas de execução de tarefas

- `Task.java`: Responsável pela execução do programa `sd23.jar` aplicado à informação enviada aos `standalone servers` pelo `client`, assim como a estrutura `Task` utilizada para a comunicação entre o `cloud server` e os *workers*, providenciando um `id` da tarefa e a informação enviada. Para além disso dispõe de um método de calculo de memória que uma dada tarefa esteja a ocupar.

```
public class Task {
    public int taskID;
    public byte[] data;
    public Connection c;
    ...
}
```

- `TasksExecutor.java`: É a classe utilizada pelos servidores para a criação de `queues`, assim como três `ReentrantLocks` e `threads` de forma a inserir `Tasks` do `Task.java` nas queues de cada `standalone server` por parte do `cloud server`, sendo acessível por ambos. Para este efeito, em cada servidor é inicializada uma nova `thread` com o método `executeTasks` que fica responsável pela inicialização de novas `threads`, pelo número de `Tasks` existentes na `queue`. Caso o limite de memória no servidor seja alcançado ao adicionar uma nova `Task`, esta espera que haja nova memória disponível.

```
public class TasksExecutor implements Serializable {
    private int MAX_MEMORY = 1 * (5000);
    private int ACTUAL_MEMORY = 0;
    private Queue<Task> taskQueue;
    ...
}
```

3.2 Estruturas de comunicação

Para uma boa comunicação entre as diferentes componentes foi necessário a implementação de diversas `classes` para garantir **encapsulamento** e possível **escalabilidade**.

Todas comunicam com o uso do TCP, porque para além de ser o protocolo referido nas aulas, ele assegura a fiabilidade da comunicação tratando da verificação de erros automaticamente.

- `Frame.java`: Resultou da necessidade de uma comunicação padrão entre cliente e servidores. Possuí uma **tag**, um **taskid** e a **informação** a ser enviada. Esta `tag` é o que diferencia os diferentes tipos de pedidos, sendo o `taskid` responsável pela identificação do cliente no `cloud server` após a execução em um `standalone server`.

  ```
  public class Frame {
      public final int tag;
      public final byte[] data;
      public final int taskid;
      ...
  }
  ```

- `Connection.java`: Para uma correta utilização de `sockets` e **escrita** e/ou **leitura** das mesmas, esta classe é a principal responsável pelo envio de `frames` e `strings` entre os constituintes do projeto. Recorre a `DataStreams` assim como `ReentrantLocks`, de modo a impedir conflitos e possível perda de informação na comunicação.

  ```
  public class Connection implements AutoCloseable {
      private DataInputStream is;
      private DataOutputStream os;
      ...
          public Connection(Socket s) throws IOException {
          this.socket = s;
          this.is = new DataInputStream(new BufferedInputStream(s.getInputStream()))
              ;
          this.os = new DataOutputStream(new BufferedOutputStream(s.getOutputStream
              ()));
      }
  }
  ```

- `Users.java`: Esta classe surgiu para possibilitar a implementação do `login` e `registo` dos `clientes`, fazendo uso de um `hashmap` que é acedido através do `ReentrantReadWriteLock`, de modo a ser possível o registo/remoção de novos `clientes` em ficheiro, de forma segura.

3.3 Estrutura das principais classes

3.3.1 CloudServer

O `CloudServer` é o servidor central responsável pela coordenação de todas as operações no sistema. Ele é iniciado antes de todas as outras componentes e permanece à espera das conexões de outros clientes ou servidores. Além disso, ele é encarregue de enviar as tarefas pedidas pelos `Clients` para os `StandaloneServers`, atuando como uma espécie de roteador. Após os mesmos acabarem as suas tarefas, o `CloudServer` recebe a informação deles e envia-a de volta aos `Clients`.

Estrutura criada:

- `TasksExecutor.java`: Esta classe é a responsável pela execução das tarefas num `StandaloneServer`. Mantém uma fila de tarefas a serem executadas e gerencia a disponibilidade de memória. Quando uma tarefa é concluída, o resultado é enviado de volta para o `CloudServer`.

3.3.2 StandaloneServer

O `StandaloneServer` é um servidor "*worker*" dedicado à execução das tarefas. Ao contrário do `CloudServer`,o `StandaloneServer` não lida diretamente com a gestão da fila de tarefas ou notificações para os clientes. Em vez disso, ele foca-se na execução das tarefas recebidas. Múltiplos `StandaloneServers` podem ser conectados ao `CloudServer`, proporcionando escalabilidade ao sistema.

3.3.3 Client

O `Client` representa a interface de interação com o sistema de modo a o tornar mais acessível para o utilizador comum. Ele é o responsável por submeter os pedidos de execução de tarefas e de vizualização de estado delas ao `CloudServer`. Para além disso, através da sua interface somos capazes de introduzir ficheiros `input` e escrever em ficheiros `output`.

Estrutura criada:

- `ClientDemultiplexer.java`: Esta classe é a responsável pela receção e envio de `FrameClients` entre o `client` e o `cloud server`, sendo executada numa nova `thread`. Ao enviar novos pedidos, armazena estes em um `map` de FrameClients, onde a *key* de cada pedido é a sua `tag`. As `FrameClients` são uma nova estrutura com o objetivo de criar `queues` para cada tipo de `tag` existente, alterando o seu estado caso o `client` esteja à espera de respostas do `cloud server`.

```
public class ClientDemultiplexer {
    ...
    private final Map<Integer, FrameClient> map = new HashMap<>();
    ...
    private class FrameClient {
        int waiting = 0;
        Queue<Frame> queue = new ArrayDeque<>();
        Condition c = l.newCondition();
}
```

3.3.4 Tipos de Mensagens

Segue o tipo de tags existentes no projeto, primeiro o `integer` correspondente, seguido da orientação em que ocorre e terminando com a sua descrição:

Tipo	De → Para	Descrição
1	Client → CloudServer	Pedido de Login
2	Client → CloudServer	Pedido de Registo
30	Client → CloudServer	Pedido de Execução de Tarefa
31	CloudServer → Client	Resultado da execução da Tarefa
40	Client → CloudServer	Pedido de Estado do Servidor
41	CloudServer → Client	Resposta com Estado do Servidor
90	StandaloneServer → CloudServer	Notificação de abertura de StandaloneServer
91	CloudServer → StandaloneServer	Identificação por ID de StandaloneServer
100	Client → CloudServer	Tag de conexão de Cliente
1000	CloudServer → StandaloneServer	Pedido de Execução de Tarefa
1001	StandaloneServer → CloudServer	Resposta com Resultado da Tarefa
1002	StandaloneServer → CloudServer	Resposta com Resultado da Tarefa (em caso de exceção da JobFunction)

4 Implementação do Serviço

4.1 Utilização de threads e comunicação TCP

As threads no nosso trabalho são utilizadas para proporcionar concorrência no processamento de pedidos, e a sua implementação envolve diferentes tipos de threads para cada componente do sistema.

No `CloudServer`, são utilizadas duas threads principais. Uma thread aguarda novos pedidos, enquanto outra lida com cada cliente e servidor, mantendo a conexão aberta. Esta abordagem permite que o servidor principal trate simultaneamente de múltiplos clientes e servidores.

No `StandaloneServer`, há uma thread a comunicar com o `CloudServer` e um número dinâmico de threads adicionais para lidar com os pedidos de execução. Cada pedido recebido é tratado numa thread separada, permitindo que o servidor atenda a vários pedidos simultaneamente.

No `Client`, é utilizada uma abordagem similar ao `StandaloneServer`, com uma thread dedicada para cada pedido. Isso permite que o cliente aguarde respostas enquanto mantém a capacidade de enviar novos pedidos.

Sempre que usamos threads em paralelo, houve a necessidade de garantir a exclusão mútua e a proteção dos dados. Para cada região crítica, foram criados e implementados locks, garantindo assim que nunca duas threads usariam a mesma região crítica ao mesmo tempo.

Algumas dessas regiões críticas incluem:

- No `CloudServer`, aonde a adição de tarefas à fila e ao histórico, a manipulação de servidores (adição ou remoção) e a inclusão de clientes têm de ser protegidas. Além disso, abrange o recebimento de respostas do `StandaloneServer` e outras áreas que envolvem o acesso e modificação das estruturas de dados.

- No `Users`, que está presente no `CloudServer`, o processo de login e registro é tratado para evitar a adição e remoção de entradas simultaneamente.

- No `ClientDemultiplexer`, localizado no `Client`, ele realiza um bloqueio (*lock*) sempre que recebe uma nova *frame*.

4.2 Estratégias de Gestão de Memória

A utilização de filas (*dequeue*) desempenha um papel crucial na gestão de tarefas e memória.

Os programas ao serem executados são colocados numa fila, permitindo que o sistema verifique continuamente a disponibilidade de memória antes da execução. Caso não haja memória disponível imediatamente, os programas permanecem na fila e são posteriormente executados pela sua ordem de chegada.

A gestão da memória disponível é realizada por um processo de verificação do tamanho dos programas em espera na fila. Caso a memória seja suficiente, os programas são enviados para os servidores através de um loop que percorre os servidores em busca de um candidato adequado para executar os programas, garantindo uma distribuição eficiente de tarefas.

5 Testes e Resultados

A validação e teste do sistema foram realizados através de uma série de casos de teste com o objetivo de garantir a robustez e eficiência do sistema distribuído. Esses testes foram conduzidos utilizando um ficheiro em formato de texto que ao ser lido pelo programa executava uma sequência de tarefas. Esse ficheiro continha informações como (id, tamanho, data). O sistema gerou resultados correspondentes, incluindo o id e a data em formato binário. Esses casos de teste foram essenciais para verificar se o sistema operava corretamente em diferentes cenários.

A seguir, apresentamos alguns dos testes realizados:

- **Teste de Execução de Tarefas:** Verificamos a capacidade do sistema de receber pedidos de execução de tarefas dos clientes, colocá-los na fila e distribuí-los aos servidores disponíveis para execução.
- **Teste de Estado do Servidor:** Fizemos pedidos ao sistema para obter o estado dos servidores. Isso inclui informações sobre a capacidade de memória, carga de trabalho atual e disponibilidade.
- **Teste de Login e Registo de Clientes:** Validamos o processo de autenticação e registo de clientes no sistema. Garantimos que apenas clientes autorizados pudessem aceder ao sistema.
- **Teste de Comunicação Cliente-CloudServer:** Verificamos a comunicação adequada entre os clientes e o `CloudServer`. Isso incluiu a transmissão de pedidos, o tratamento de respostas e a garantia de que as operações ocorressem sem perdas de dados ou erros.
- **Teste de Concorrência:** Avaliamos o desempenho do sistema em situações de carga elevada. Isso incluiu a submissão simultânea de vários pedidos de diferentes clientes. Observamos como o sistema gerenciava concorrência e distribuía tarefas eficientemente.

6 Conclusão

Com a realização deste Trabalho Prático, foi possível relembrar e consolidar a linguagem de programação Java, assim como os conhecimentos adquiridos nas aulas, tanto a nível prático como teórico. Destacam-se os aprendizados relacionados à implementação de sistemas distribuídos, protocolos de comunicação, uso de threads e sockets TCP. Em suma, foram desenvolvidas as três aplicações-chave requisitadas: o **StandaloneServer**, o **CloudServer** e o **Client**. Cada uma dessas aplicações desempenha um papel fundamental no sistema, permitindo uma execução eficiente e distribuída de tarefas. No entanto, é importante também destacar que o desenvolvimento do trabalho não esteve isento de desafios. Um dos desafios mais significativos foi a dificuldade em determinar como o servidor reduziria a memória disponível após a execução de uma thread. Essa complexidade envolveu a gestão de valores corretos da memória e exigiu soluções cuidadosas para garantir a integridade do sistema.