Scatter and Gather Design Pattern Implementation

Bernardo De Marco Gonçalves - 22102557

Tecnologias utilizadas

A aplicação foi desenvolvida na linguagem de programação Java, e para a manipulação de sockets foram utilizadas suas implementações nativas (java.net.Socket e java.net.ServerSocket).

Para estruturação da aplicação, foi utilizado o gerenciador Apache Maven. Além disso, para manipulação de JSONs foi utilizada a bibliteca Gson, e para logs foi utilizado o Apache Log4j.

Setup da aplicação

Para execução da aplicação, é necessário ter o Java 17 e o Apache Maven 3.6.3 instalados e configurados.

Compilação

Para compilação, deve-se executar:

```
mvn compile
```

Execução

Para execução, devem ser abertos quatro terminais, um para cada processo a ser executado. Em seguida, devem ser executados, em ordem, os workers, root e o client.

Em um terminal, execute um worker:

```
mvn exec:java -Dexec.mainClass="com.github.bernardodemarco.textretrieval.worker.instances.Worker1"
```

Em outro, execute o outro worker:

```
mvn exec:java -Dexec.mainClass="com.github.bernardodemarco.textretrieval.worker.instances.Worker2"
```

Em outro, execute o root:

```
mvn exec:java -Dexec.mainClass="com.github.bernardodemarco.textretrieval.root.Root"
```

Em outro, execute o client:

```
mvn exec:java -Dexec.mainClass="com.github.bernardodemarco.textretrieval.client.Client"
```

Após executar o client, ele se conectará ao root e começará a enviar requisições. Um vídeo que exemplifica a execução do programa pode ser encontrado nesse link.

Estrutura dos diretórios/pacotes

```
├── pom.xml # Apache Maven pom.xml file
├── README.md # docs in markdown
├── docs.pdf # docs in pdf
├── src
│   └── main # source code container
│       ├── java
│       │   └── com
│       │       └── github
│       │           └── bernardodemarco
│       │               └── textretrieval
│       │                   ├── client # client node
│       │                   │   ├── Client.java
│       │                   │   └── dto
│       │                   │       └── QueryDTO.java
│       │                   ├── communication # reusable and encapsulated components, such as Client, Server and ScatterGather implementations.
│       │                   │   ├── client
│       │                   │   │   ├── ClientConnection.java
│       │                   │   │   └── TCPClientConnection.java
│       │                   │   ├── scattergather
│       │                   │   │   ├── ScatterGather.java
│       │                   │   │   └── ScatterGatherService.java
│       │                   │   └── server
│       │                   │       ├── Server.java
│       │                   │       └── TCPServer.java
│       │                   ├── root # root node
│       │                   │   ├── dto
│       │                   │   │   ├── KeywordDTO.java
│       │                   │   │   └── QueryOccurrencesDTO.java
│       │                   │   └── Root.java
│       │                   ├── utils # utilities
│       │                   │   └── FileUtils.java
│       │                   └── worker # worker node
│       │                       ├── dto
│       │                       │   └── KeywordOccurrencesDTO.java
│       │                       ├── instances
│       │                       │   ├── Worker1.java
│       │                       │   └── Worker2.java
│       │                       └── Worker.java
```

```
|       └── resources # resources file containing configurations files
|           ├── client
|           |   ├── client.properties
|           |   └── queries.json # queries to be performed by the client
|           ├── log4j2.xml # log4j2 config
|           ├── root
|           |   └── root.properties
|           ├── textfiles # text files
|           |   ├── text1.txt
|           |   ├── text2.txt
|           |   ├── text3.txt
|           |   ├── text4.txt
|           |   └── text5.txt
|           └── workers
|               ├── worker1.properties
|               └── worker2.properties
```

Design da aplicação

Componentes de comunicação

Todos os nós da aplicação usufruem de dependências customizadas desenvolvidas visando encapsular e reutilzar conceitos de programação distribuída. Dentre eles, foram implementadas classes representando conexões TCP, servidores TCP e um serviço Scatter/Gather.

TCPClientConnection

A classe TCPClientConnection encapsula operações de criação de um socket, conexão a um outro endpoint, envio e recebimento de mensagens, e o fechamento da conexão.

Para uma classe utilizá-la como dependência, basta instanciá-la passando como parâmetro o endereço de rede e a porta. A classe Client e a ScatterGatherService a utilizam internamente.

TCPServer

Essa classe encapsula o funcionamento de um servidor TCP. Portanto, provê operações para escutar requisições em um porta, enviar e receber dados, e fechar o canal de comunicação. As classes `Root` e `Worker` a utilizam como dependência.

ScatterGatherService

Essa classe representa a implementação do design pattern de programação distribuída Scatter/Gather. Ele provê operações para realizar o espalhamento (scatter) de requisições e a reunião (gather) das respostas.

Internamente, ela possui uma lista de conexões e um thread pool. Ao realizar o scatter, as requisições são distribuídas de maneira circular (round robin) para as conexões. No momento em que uma é enviada, uma task é enviada ao thread pool para escutar por um retorno.

Ao executar o gather, cada retorno das tarefas enviadas é recuperado e retornado ao caller. Com isso, o processamento das tarefas é realizado em paralelo, o que potencializa o desempenho da aplicação.

Nós da aplicação

Client

O processo client é o responsável por realizar as requisições de busca de texto ao sistema. Para isso, ele lê uma lista de queries pré-definida, e as envia ao nó Root.

O protocolo de comunicação utilizado estabelece que as requisições do Client ao Root devem ser compostas por uma string JSON no seguinte formato:

```
{
    "query": "parallel and distributed computing"
}
```

Após enviar uma requisição, o Client aguarda pelo seu retorno e, antes de realizar a próxima, suspende sua execução por um intervalo de tempo entre 1000 e 2000 milisegundos.

A resposta que é retornada para cada query segue o seguinte formato:

```
[
   {
      "fileName": "text.txt",
      "fileContent": [
         "first line",
         "second line",
         "third and last line"
      ],
      "occurrences": 2
   },
   // ...
]
```

Root

O nó Root possui como dependência um servidor e um serviço Scatter/Gather.

Ao receber uma requisição, realiza o parse da query, a convertendo em um conjunto de palavras-chave. Essas, por sua vez, são distribuídas aos workers através do serviço Scatter/Gather. Cada requisição ao worker segue o seguinte formato:

```
{
    "keyword": "parallel and distributed computing"
}
```

Em seguida, aguarda as respostas dos workers (gather), trata os dados recebidos e retorna uma resposta ao cliente.

Worker

Cada worker recebe uma palavra-chave e busca as suas ocorrências nos arquivos de texto. Uma vez realizado o processamento, retorna uma resposta ao Root com o formato:

```
[
   {
      "fileName": "text.txt",
      "occurrences": 2
   },
   // ...
]
```

Exemplos de execução da aplicação

Ao realizar a execução da aplicação, no terminal de cada processo é possível observar a geração de logs. Os logs foram utilizados em três níveis:

- DEBUG: Logs utilizados para depurar e entender o contexto em que a aplicação está.
- INFO: Logs com informações importantes, como estabelecimento de conexões e retorno de requisições.
- ERROR: Logs para eventuais erros que possam ocorrer e que não possam ser tratados em tempo de execução, como a impossibilidade de ler um determinado arquivo.

Execução do Client

Ao executar o processo Client, os seguintes logs são interessantes:

```
# cliente se conectou com sucesso ao servidor Root
2024-06-23 20:43:34 [INFO] (c.g.b.t.c.c.TCPClientConnection) - Successfully connected to server
        [127.0.0.1:8000].


# request sendo enviado
2024-06-23 20:43:34 [DEBUG] (c.g.b.t.c.Client) - Sending query [{"query":"concurrent and parallel and
        distributed programming"}].


# response recebida

2024-06-23 20:43:34 [INFO] (c.g.b.t.c.Client) - Received query response [[

  {

    "fileName": "text5.txt",

    "fileContent": [
      "Parallel programming is not without its challenges. One of the primary difficulties is ensuring
        that parallelized tasks do not interfere with each other, leading to bugs and unpredictable
        behavior. Debugging parallel programs can be significantly more complex than debugging
        sequential programs due to the interactions between concurrently executing tasks. Tools and
        techniques for debugging and profiling parallel applications are essential for developers in
        this field.",

      "",
      "Moreover, not all problems can be easily parallelized. The extent to which a problem can be
        parallelized depends on its nature and the dependencies between tasks. Amdahl\u0027s Law
        provides a theoretical limit to the speedup achievable through parallelization, highlighting the
        diminishing returns of adding more processors to a parallel system. Understanding these
        limitations is crucial for effectively designing and optimizing parallel programs."

    ],

    "occurrences": 11

  },

  {

    "fileName": "text4.txt",

    "fileContent": [
      "Distributed programming enables the development of applications that can scale horizontally by
        adding more machines to the network. This scalability is achieved through the decomposition of
        tasks into smaller, independent units that can be executed on different machines. Distributed
        systems can handle increased load by simply adding more nodes, making them ideal for handling
        large-scale, resource-intensive applications.",

      "",
      "A significant advantage of distributed programming is fault tolerance. Distributed systems can
        continue functioning even if some nodes fail, as the workload can be redistributed among the
        remaining nodes. This redundancy is critical for maintaining the availability and reliability of
        applications, especially in environments where downtime can have severe consequences, such as
        financial systems or healthcare applications."

    ],

    "occurrences": 7

  },

  {

    "fileName": "text3.txt",

    "fileContent": [
      "In parallel programming, synchronization and data sharing between threads are critical issues.
        Proper synchronization mechanisms like locks, semaphores, and barriers are essential to prevent
        race conditions and ensure data consistency. These mechanisms allow threads to coordinate their
        activities and access shared resources safely, which is vital for the correctness of parallel
        programs.",

      "",
      "Another important aspect of parallel programming is load balancing, which ensures that all
        processors are utilized efficiently. Dynamic load balancing techniques can redistribute tasks
        among processors to avoid scenarios where some processors are idle while others are overloaded.
        This helps in achieving optimal performance and efficient resource utilization in parallel
        computing environments."
```

```
    ],
    "occurrences": 11
  },
  {
    "fileName": "text1.txt",
    "fileContent": [
      "Parallel programming is a computing paradigm that enables the execution of multiple processes
        simultaneously. By dividing large problems into smaller tasks, parallel programming can leverage
        multiple processors to solve these tasks concurrently. This approach significantly reduces the
        time required for computation and enhances the performance of applications, especially in
        scientific and engineering domains.",
      "",
        "The primary advantage of parallel programming is its ability to handle large datasets and
        complex computations efficiently. Techniques such as multithreading, multiprocessing, and using
        parallel algorithms are common in this field. Multithreading allows multiple threads to run
        concurrently within a single program, while multiprocessing utilizes multiple CPUs to execute
        processes simultaneously. These techniques are essential in modern computing environments where
        performance and speed are critical."
    ],
    "occurrences": 12
  },
  {
    "fileName": "text2.txt",
    "fileContent": [
        "Distributed programming involves the creation of software systems that run on multiple
        computers, which communicate and coordinate their actions by passing messages. These systems are
        designed to share resources and data processing tasks across a network, enhancing the
        scalability and reliability of applications. Distributed programming is fundamental to the
        operation of large-scale web services, cloud computing, and enterprise-level applications.",
      "",
      "One of the key challenges in distributed programming is ensuring consistency and fault tolerance
        across multiple nodes. Techniques such as distributed databases, consensus algorithms, and
        replication are used to manage data consistency and availability. Distributed systems must also
        handle network latency and potential failures gracefully, making robust error handling and
        recovery mechanisms crucial components of these systems."
    ],
    "occurrences": 17
  }
]].

# informando que vai esperar para realizar a próxima query
2024-06-23 20:43:36 [DEBUG] (c.g.b.t.c.Client) - Sleeping for 1329 milliseconds.
```

Execução do Root

```
# conectou-se ao worker 1
2024-06-23 20:43:34 [INFO] (c.g.b.t.c.c.TCPClientConnection) - Successfully connected to server
        [127.0.0.1:8001].


# conectou-se ao worker 2
2024-06-23 20:43:34 [INFO] (c.g.b.t.c.c.TCPClientConnection) - Successfully connected to server
        [127.0.0.1:8002].
```

```
# recebeu requisição do cliente
2024-06-23 20:43:34 [INFO] (c.g.b.t.r.Root) - Received query [{"query":"concurrent and parallel and
        distributed programming"}] from client.


# espalhando as requisições para os workers
2024-06-23 20:43:34 [DEBUG] (c.g.b.t.c.s.ScatterGatherService) - Scattering data
        [[{"keyword":"programming"}, {"keyword":"concurrent"}, {"keyword":"and"}, {"keyword":"parallel"},
        {"keyword":"distributed"}]] to [2] connections using round-robin algorithm.


# retorno dos jobs dos workers
2024-06-23 20:43:34 [DEBUG] (c.g.b.t.r.Root) - Received [[[{"fileName":"text1.txt","occurrences":3},
        {"fileName":"text2.txt","occurrences":3},{"fileName":"text3.txt","occurrences":2},
        {"fileName":"text4.txt","occurrences":2},{"fileName":"text5.txt","occurrences":1}], [],
        [{"fileName":"text1.txt","occurrences":5},{"fileName":"text2.txt","occurrences":9},
        {"fileName":"text3.txt","occurrences":5},{"fileName":"text4.txt","occurrences":1},
        {"fileName":"text5.txt","occurrences":5}], [{"fileName":"text1.txt","occurrences":4},
        {"fileName":"text3.txt","occurrences":4},{"fileName":"text5.txt","occurrences":5}],
        [{"fileName":"text2.txt","occurrences":5},{"fileName":"text4.txt","occurrences":4}]]] from
        workers.
```

Execução do Worker

```
# servidor Worker escutando requisições

2024-06-23 20:43:34 [INFO] (c.g.b.t.c.s.TCPServer) - Listening on port 8001.


# recebeu keyword

2024-06-23 20:43:34 [INFO] (c.g.b.t.w.Worker) - Received keyword [{"keyword":"programming"}].


# buscas das keywords nos arquivos de texto
2024-06-23 20:43:34 [DEBUG] (c.g.b.t.w.Worker) - Keyword [programming] has appeared [3] times in
        [text1.txt]
2024-06-23 20:43:34 [DEBUG] (c.g.b.t.w.Worker) - Keyword [programming] has appeared [3] times in
        [text2.txt]
2024-06-23 20:43:34 [DEBUG] (c.g.b.t.w.Worker) - Keyword [programming] has appeared [2] times in
        [text3.txt]
2024-06-23 20:43:34 [DEBUG] (c.g.b.t.w.Worker) - Keyword [programming] has appeared [2] times in
        [text4.txt]
2024-06-23 20:43:34 [DEBUG] (c.g.b.t.w.Worker) - Keyword [programming] has appeared [1] times in
        [text5.txt]
```