CO3090/CO7090
Distributed Systems and Applications
Coursework 2

File System Search Engine

Important Dates:

Handed out: 3-March-2019
Deadline: 31-March-2019 at 23:59 GMT

- This coursework counts as 13.33% of your final module mark.
- This coursework is an individual piece of work. Please read guidelines on plagiarism on the University webpage: https://www2.le.ac.uk/offices/sas2/assessments/plagiarism/penalties
- This coursework requires knowledge about RPC/RMI and JMS.

Introduction

Your task is to develop a Distributed File Search Engine (DFSE) that allows users to query files and directory structures on a distributed file system. Your tasks are (1) to implement the RMI server that accepts and executes queries from clients, and (2) write and deploy a message-driven middleware to the application server.

Assume that

- The DFSE server has no prior knowledge about the implementation details of the queries submitted by the clients.
- Files may be stored on multiple file servers. Unless explicitly told otherwise, DFSE always search all file servers.
- The information about each file server is stored in a list of index files: `RemoteFileSystem1.txt, RemoteFileSystem2.txt …` `RemoteFileSystemN.txt` located inside the `filesystems` folder.
  (Please see the Appendix 1 for more information. Note: methods for reading these files are already provided in `FileUtility.java`)

Question 1 [15 marks]

Design the architecture for DFSE. Explain your design with an UML class diagram. Your design should allow clients to send arbitrary queries to the server without having to change the remote interfaces. **(Hints:** Your UML diagram should include any remote interface and the serializable object. Please read the comments in the source code)

Question 2 [10 marks]

Implement the following classes (or methods) for the DFSE server:

(2.1) Define the remote method signatures in `RFSInterface.java`
(2.2) Give the parameters and return type for the method(s) in `SearchCriteria.java`
(2.3) Implement the RMI server `RFSServer.java`

Question 3 [50 marks]

Implement the following query classes and methods for the DFSE client:

(3.1) QueryFileSearch.java

Given a file name, `QueryFileSerach.java` should return the full paths (absolute paths to the root directory) of the file name, if the file(s) exist on one of the file servers. If more than one file with the given name is found, then the result should include all paths found. For example, given the directory structure in `RemoteFilesystem1.txt` and `RemoteFilesystem2.txt`, when searching for "`hello.txt`", the RFSServer should return a JSON array:

```
{
 "list": [{
        "fs": "RemoteFilesystem1",
        "path": "//A/C/E/hello.txt"
    },
    {
        "fs": "RemoteFilesystem2",
        "path": "//A/B/hello.txt"
    },
    {
        "fs": "RemoteFilesystem2",
        "path": "//A/C/E/hello.txt"
    },
    {
        "fs": "RemoteFilesystem2",
        "path": "//A/C/E/F/hello.txt"
    }
 ]
}
```

(3.2) QueryMaxDepth.java

`QueryMaxDepth.java` should return the maximum (overall) directory depth on all file servers. For example, given the directory structure in `RemoteFilesystem1.txt` and `RemoteFilesystem2.txt`, The RFSServer should return:

```
{
 "list": [{
        "fs": "RemoteFilesystem2",
        "depth": "5"
    },
    {
        "fs": "RemoteFilesystem1",
        "depth": "4"
    }
 ]
}
```

(3.3) QueryTree.java

For each file server, `QueryTree.java` should return the directory structure as a string formatted according to the specified format. For example, given the directory structure in `RemoteFilesystem1.txt` and `RemoteFilesystem2.txt`. The RFSServer should return:

```
{
    "list": [{
            "fs": "RemoteFilesystem1",
            "path":
"A{B,C{books.xls,D,E{readme.txt,hello.txt},F{G}}}"
        },
        {
            "fs": "RemoteFilesystem2",
            "path":
"A{B{hello.txt,D{abc.txt,xyz.txt}},C{E{hello.txt,F{hello.txt}}}}"
        }
    ]
}
```

(3.4) In RFSClient.java, for each query class implemented in (3.1), (3.2) and (3.3), you should create an instance respectively to demonstrate how it works. Build two JAR files (server.jar, client.jar) containing all classes necessary to run the server and the client. Provide the commands (shell scripts or batch files) for running your server and the client.

Hints: (1) the order in which the folders/files are listed is not important, if they are located in the same folder. (2) You should use appropriate data types (e.g. `Vector, HashMap` etc) to store the results. (3) You should consider using multi-threading to maximise the degree of parallelism to improve the performance of searching. (The maximum number of threads running concurrently should be limited to MAX_THREAD_NUM=10).

Question 4 [25 marks]

Create a new EJB project and create a Message-Driven Bean (e.g. MyMDB.java) in the project to process the queries in a JMS queue. Create two queues in the Glassfish admin console (JNDI names /jms/queryQueue and /jms/resultQueue). This MDB must implement the MessageListener, and be able to process the queries in /jms/queryQueue. The query results should be stored in jms/resultQueue.

Appendix

The directory structure on the file server “RemoteFilesystem1” can be found in `RemoteFilesystem1.txt`:

```
 # Each line contains TYPE, NAME and PARENT, separated by commas:
 # ITEM TYPE (DIR or FILE), FILE/DIRECTORY, PARENT DIRECTORY
 #
 #        A-+
 #          +-B
 #          +-C-+ (books.xls.)
 #              +-D
 #              +-E (read.txt. hello.txt)
 #              +-F-+
 #                  +G
 #
 # Note: the root folder's parent is labeled as "?"
 #       Assume directories all have different names.
 #
 DIR,A,?
 DIR,B,A
 DIR,C,A
 FILE,books.xls,C
 DIR,D,C
 DIR,E,C
 FILE,readme.txt,E
 FILE,hello.txt,E
 DIR,F,C
 DIR,G,F
```

The directory structure on file server “RemoteFilesystem2” can be found in `RemoteFilesystem2.txt`:

```
# Each line contains TYPE, NAME and PARENT, separated by commas:
# ITEM TYPE (DIR or FILE), FILE/DIRECTORY, PARENT DIRECTORY
#
#        A-+
#          +-B-+  (hello.txt)
#          +   D  (abc.txt, xyz.txt)
#          +-C-+
#              +-E (hello.txt)

#
# Note: The root folder's parent is labeled as "?"
#        Assume directories all have different names
DIR,A,?
DIR,B,A
FILE,hello.txt,B
DIR,D,B
FILE,abc.txt,D
FILE,xyz.txt,D
DIR,C,A
DIR,E,C
FILE,hello.txt,E
DIR,F,E
FILE,hello.txt,F
```

Note: there could be more than two index files. The DFSE server should be able to search all index files in filesystems folder.

Classes

Server/Client-side

`RFSInterface.java`:
Remote Interface, the RMI server must implement this interface.
`SearchCriteria.java`:
Serializable object, all queries must implement this interface.
`FileItem.java`:
Representing a file or a directory
`FileItemType.java`:
Enum class for `FileItem.java`

Server-side only:

`RFSServer.java`
Main program of the RMI Server
`FileUtility.java`
Contains methods for reading `RemoteFilesystem*.txt`

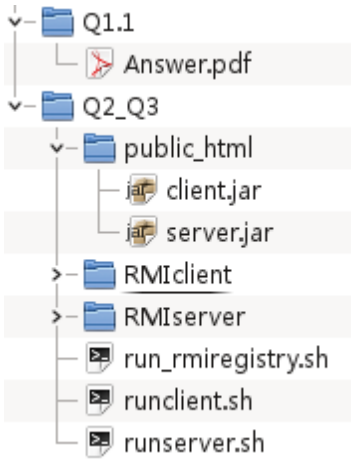
Client-side only:

`QueryFileSearch.java`
`QueryMaxDepth.java`
`QueryTree.java`

Please refer to question (3.1), (3.2) and (3.3) for more detail.

Submission

Please create a directory structure as in the figure below,



Where

- Subdirectory **Q1.1** contains **Answers.pdf** (Answer to Q1.1)
- Subdirectory **Q2-Q3** contains all Java files of your solution (Answers to Q2, Q3)
- Subdirectory **public_html** contains two JARs archives.
- Subdirectory **RMIclient** contains source codes for the client.
- Subdirectory **RMIserver** contains source codes for the server.
- Compress all files required for **Q4** in folder Q4.

Please compress your coursework using zip**.**

The archive should be named **(your_id)_DSA_cw2.zip** (e.g. yh37_DSA_cw2.zip). Your submission should also include a completed coursework coversheet (print and signed pdf or image). You need to submit the zip file via Blackboard and you are allowed to re-submit as many times as you like **before** the deadline.

Marking Criteria

(G) <30%

- The submitted code does not compile.
- The shell scripts or batch files provided for the RMI server/client could not execute.
- The message-driven bean does not compile.

(F) 30-40%

- The submitted code compiles but there are major issues with the RMI server deployment.
- The RMI server fails to register itself with the rmiregistry.
- The RMI clients are partially implemented but establish communication with the server.
- There are still issues with the JNDI configuration.
- Fail to deploy the Message-Driven Bean to the application server.

(E) 40-50%

- The shell scripts (or Windows batch files) for the RMI server successfully execute.
- RMI Server successfully binds with the rmiregistry.
- Three RMI clients are partially working but the remote interface does not allow the server to accept new queries without changing the remote interface.
- The RMI server can send the messages to the destination JMS queue, but the Message-driven bean still has some problem processing the data in the queue.

(D) 50-60%

- The shell scripts (or Windows batch files) for RMI server/client successfully execute.
- RMI Server successfully binds with the rmiregistry, and the RMI server allows the clients to introduce new queries without changing the remote interface dynamically.
- Three queries can be executed on the server,
- There are still major issues with some of the queries, in some cases, the server crashes or fails to return the correct results.
- Queries can be sent to the destination JMS queue with the given JNDI name.
- Message-driven bean can process the queries from the queue but the results are not saved to (/jms/resultQueue).

(C) 60-70%

- Meets all the criteria specified in (C).
- Mostly correct query results, though there are minor issues.
- Message-driven bean can process the queries from the queue, and the results are saved to /jms/resultQueue.

(B) 70-80%

- Meets all the criteria specified in (C).
- All test cases passed (RMI queries/Message-driven bean)

(A) 80+%

- Meets all the criteria specified in (B)
- Apart from the RMI server/client and the MDB, design and implement a Stateless session bean and expose the remote interface as a SOAP web service, write a non-java client (e.g. in .NET, Python) to retrieve the results stored in /jms/resultQueue.