

# **COMP 90015 Distributed Systems**

## **Assignment 1 Report**

### **1. System description:**

In this assignment, a client-server architecture remote dictionary searching system was designed.

The communication between the client-server is based on the Socket communication. For every client access, the server will produce a new Socket and then start a new service thread to communicate to the client. Based on this multi-thread design in the server side, the system can serve multiple clients at the same time.

In the server side, an assistant word index array is used to speed the word searching. The index array store the word number starting at the corresponding letter stored in the dictionary which initialized at the system starting. When searching a word in the dictionary file, the server can only search the dictionary file section corresponding to that word's start letter and skip all the other file section.

As a distributed application, the communication abnormal is the main exception source. In this application, the major communication exception includes: the opposite side can not reach, the connection lost and the bad communication data. The system can detect all these exception and give the hint information. For in this client-server application, the communication starter is always from the client side, so the client side has a more burden than the server for this kind of exception.

The other exception come from the file management, this may includes file not found, file IO exception. For only in server side have the file operation, the file exception is processed in the server side.

The last exception comes from the command input error, both client and server side can detect this error and give the hint message.

In this system, the shared resources among different threads include the dictionary file and the previous mentioned assistant word index array. For during the system running time, the client service thread only read that info, so no special thread synchronization mechanism is needed in the system.

The final design of this assignment is based on GUI interface. In the client side, the user can input the search word, this word's search result can display on the client interface. The server side has the system log info display. The system status info also display on the both side interfaces.

### **2. Server design:**

## 2.1 The Server system class design:

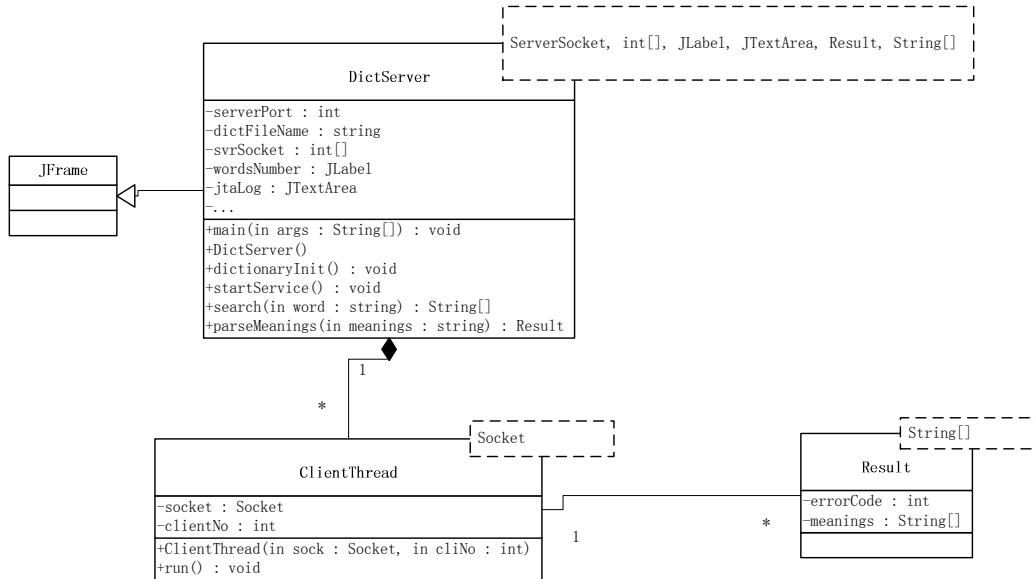
The remote dictionary server side consists of three classes, two public classes:

**DictServer** class which produce GUI interface, listen at the server port and create new client service thread;

**Result** class which define the communication structure between the client and server.

The third class is an inner class **ClientThread** resides in the DictServer which implements the Runnable interface and providing the multi-thread supporting base on per connection.

The UML class relation diagram in the server side can show in below:



## 2.2 Dictionary Searching:

The program assumes the every word occupy one line in the dictionary file. The word item start with “*word: wordMeans*”. In the *owerdMeans* section, different means of one word starts with a digit number and follow that mean string. So a complete word item defination example in the dictionary looks like below:

“*able: 1. having.... 2. Having... 3. showing talent,..*”

We can see, above dictionary item define a word “able” with three means itemes. The word searching algorithm in the system will base on above dictionary file structure.

As we mentioned in the section 1, to speed the dictionary searching, the following assistant array is used in the DictServer class:

```
static int[] wordsNumber = new int[26];
```

where, every element in the *wordsNumber* array corresponds to one English letter. It contains how many words in the dictionary file which starting at that letter.

When searching a word, the server get the first letter of the input word first, then read the *wordsNumber* index of that letter and search that letter’s section in the dictionary file.

The relative code show in below:

```
String lowWord = word.toLowerCase(); //convert the word to lower case
.....
int index = ch-'a'; //get the input word's index in the wordsNumber array
int skipLine = 0; // init the skipLine

for(int i=0;i<index;i++)skipLine += wordsNumber[i]; //get this word's skipLine number,
.....
for(int i=0;i<skipLine;i++) input.nextLine(); //skip the skipLine lines in the dictionary file

for(int i =0; i<wordsNumber[index];i++){ //searching the word in its index range
String key = lowWord+ "."; //create the input word match pattern
meaningStr = input.nextLine();

if(meaningStr.indexOf(key)==0){ //the read line starts with input word match pattern, the word found
.....
```

### 2.3 Word Means Parsing :

After found the input word's dictionary item, the next important task in the server side is to parse the word means as *String[]* form from that dictionary item. This is implemented with *DictServer* class's *String[] parseMeanings(String)* Method. This method uses two helper structres to help implement the means parsing:

```
ArrayList<String> meansList = new ArrayList<String>(); // a arrayList help variable
StringBuilder stringBuilder = new StringBuilder(); // a StringBuilder help variable
```

The *StringBuilder* variable *stringBuilder* used to pasre and build a single means string from the dictionry item. And the *ArrayList* variable *meansList* used to temp store the already parsed means. When all the means has been parsed, the method stores *meansList's* contents into a *String[]* variable and return it. The more detail implementation can see in the source code.

### 2.4 Multi-Thread supporting:

The concurrency support in the server side is based on the Socket connection, i.e., per thread per connection.

The main thread port listening and new thread creating code section show in below:

```
Socket sock = svrSocket.accept(); //Listening at the port
.....
ClientThread task = new ClientThread(sock,clientNo); //creating a new task for the socket connection
new Thread(task).start(); //creat a new Thread for the task and starting it
clientNo++; //increase client Number
```

The client service thread relative code shows in below:

```
class ClientThread implements Runnable {
```

```

...
    public ClientThread(Socket sock,int cliNo){    // constructor method, initializing the task socket
        this.socket = sock;
        this.clientNo = cliNo;
    }

    public void run(){
        .....
    }

```

In this application, the shared resources among different server thread include the dictionary file and its search speed help array *wordsNumber*. But these resources are only read by the server during the running, so no special thread synchronization mechanism are needed in this application.

## 2.5 Error and Exception Handling :

As we mentioned previous, the error and exception in the server side may come from the socket communication exception, file IO exception and command line input mistake.

The command line input parameters error is with following code to detect:

```

if(args.length!=2){    // check input paramateres length
    System.out.println("Usage: java DictServer port Dictfile");
    System.exit(0);
}

```

The file opening exception process with:

```

catch(FileNotFoundException e){
    System.out.println("Dictionary file name wrong: please check the dictioanry file name
and restarting as following:");
    System.out.println("Usage: java DictServer port Dictfile");
    System.exit(0);
}

```

The socket creating exception with:

```

catch(IOException e){
    System.out.println("Server Socket Creation Failed. Please Restart");
    return false;
}

```

The connection lost exception is with:

```

public void run(){
...
catch(IOException e){
    System.out.println("Socket Connection Lost, Exiting the "+ clientNo+"th thread.");
}

```

```
jtaLog.append("The "+clientNo+"th clients' thread exited.\r\n");
}
```

### 3. Client design:

#### 3.1 Client system class design:

In this application, the client side is a single thread application. Its function is to connect to the server and provide a User Interface so user can send the search word and display the word means in the interface.

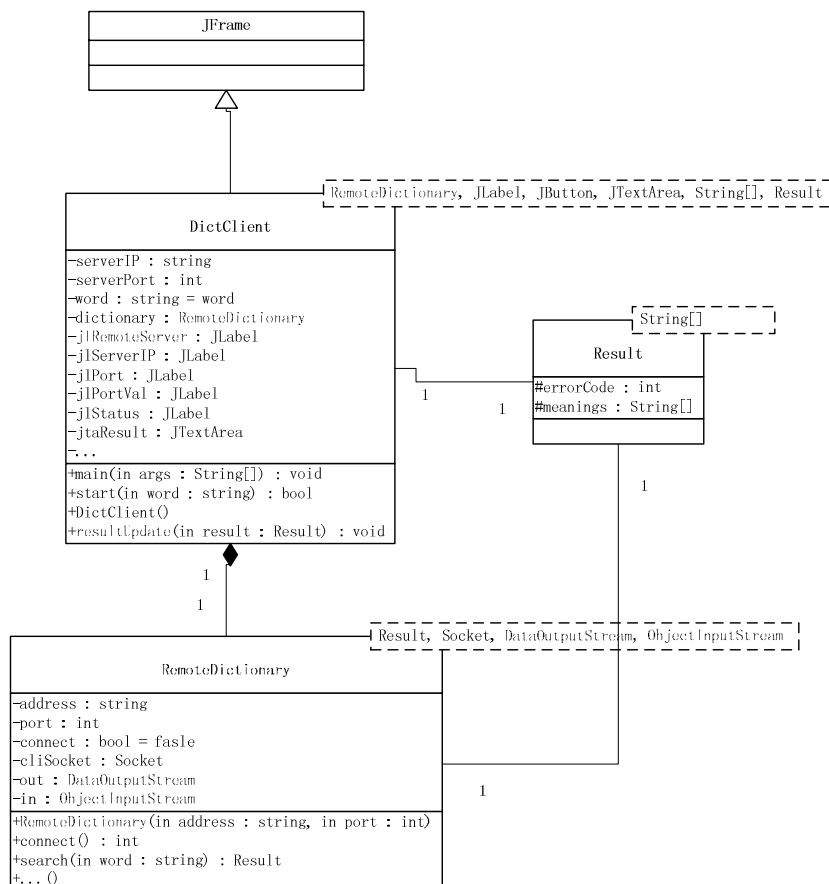
The client side consists of three classes:

**DictClient** class is the main class which produces GUI interface, create the **RemoteDictionary** object input search word and displays the search result. It also updates the system status info;

**RemoteDictionary** class is communication class to the server side. It produces the Socket connection, sending the search word and receiving the search result;

**Result** class defines Remote the communication structure between the client and server.

The UML class relation diagram in the client side can show in below:



#### 3.2 Error and Exception Handling

In the client side, the exceptions come from two kind of exception: the Socket communication exception and command line input error.

For the communication exception, we should start from the **Result** class definition. In client side, it has the following form:

```
public class Result implements Serializable{
    int errorCode; // errorcode
                // 0: success
                // -1: not found
                // -2: invalid word
                // -3: connection lost
                // -4: other error
    String[] meanings; //
}
```

So, there are five error code type defined in the client side.

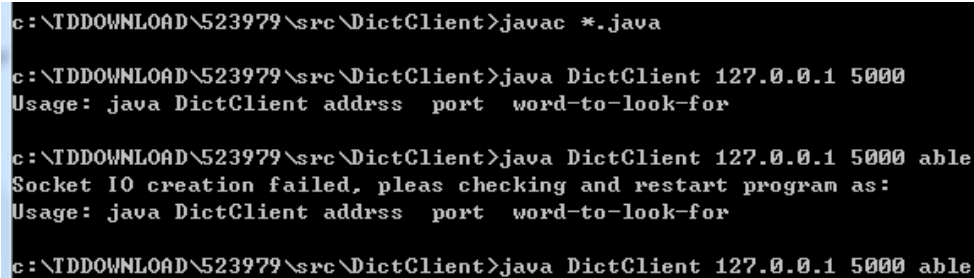
The Socket exception code are mainly located in the **RemoteDictionary** class, especially in its *connect()* and *search()* methods. The exception hint messages display are located in the **DictClient** class, especially in its *resultUpdate(Result result)* method. The more detail can see in the source code.

The command line input error detecting in client side is very similar to the server side. So don't repeat.

## 4. Testing result:

### 4.1. Client Side:

A client side startup command line input example can show in below:

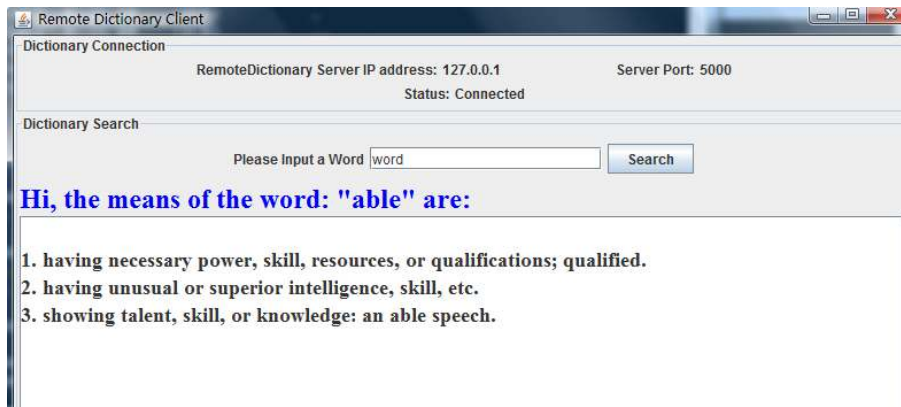


```
c:\TDDOWNLOAD\523979\src\DictClient>javac *.java
c:\TDDOWNLOAD\523979\src\DictClient>java DictClient 127.0.0.1 5000
Usage: java DictClient address port word-to-look-for
c:\TDDOWNLOAD\523979\src\DictClient>java DictClient 127.0.0.1 5000 able
Socket IO creation failed, please check and restart program as:
Usage: java DictClient address port word-to-look-for
c:\TDDOWNLOAD\523979\src\DictClient>java DictClient 127.0.0.1 5000 able
```

In above command line startup, in the first time, we input the program parameters but lacking the search word, so the program gives the hint message and stop. In the second times, the server side don't starting, so the socket connection setup failed, system also gives the hint message and stop startup.

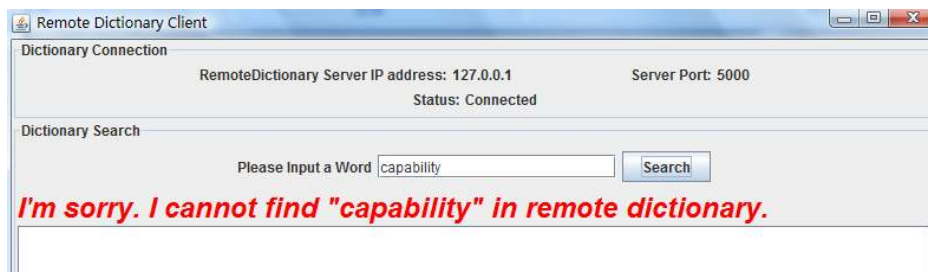
In the last time, the server starting and we give the correct command line input, so the client side interface gives the following output:

Client Side Interface after startup (Found):



If we input a word not defined in the dictionary, the system will give the not found information as following:

Client Side Interface after startup (Not Found):



When we broken the socket connection by closing the server, in the next search time, system will give the following connection lost information:



## 4.2. Server Side:

Server side startup command example as following:

```
c:\TDDOWNLOAD\523979\src\DictServer>java DictServer 5000 smallDict
Dictionary file name wrong: please check the dictioanry file name and restarting
as following:
Usage: java DictServer port Dictfile
c:\TDDOWNLOAD\523979\src\DictServer>java DictServer 5000 smallDict.txt
```

In the first time, we purposely input a wrong dictionary file name, the system can not find the file, so it gives the error hint message and halt.

In the second time, we input the correct command line input parameters, so the system starting as following:

Server Interface after startup:



In above server side user interface, we can see that in the system log area, the client side activities were logged. The log information including the client No, client host name, client IP address, search word and client exit information.

## 5. Conclusion:

In this assignment we implement a client/server architecture remote dictionary searching system. In the server side, for every client access, the system will create a new Socket and start a new service thread. Because the server side's this per thread per connection multi-thread support design, the system can serve multi-client access concurrency.

The server also implements a dictionary words index array structure to speed the dictionary search. This array store how many words in the dictionary file which starting at corresponding letter. With this array, the word can only search in that word's start letter section. Thus improve the word searching speed.

For a word with multiple means, the server side's *DictServer.parseMeanings(String)* method can retrieve the meanings from the dictionary definition item and transform it into the *String[]* form. So every single mean can display in one line.

The server GUI interface also provides system log information displaying. The log info includes: *client No*, *client host name*, *client IP address*, *search word* and *client exit information*.

The client side is a single thread application. With its GUI interface, the user can input the search word and display the search result. GUI interface also show the system status info.

The exception in the system includes: Communication abnormal, File operation exception and the Command line input error. Both the client and server side have the proper process for this kind of exceptions.