COMP90015 DISTRIBUTED SYSTEM ASSIGMENT 01 REPORT

A PREPRINT

Xulin Yang
904904
The University of Melbourne
`xuliny@student.unimelb.edu.au`

April 20, 2020

Keywords Java · Dictionary Server · Multi-thread · Socket Programming

1 Problem Context

This project is required to design and implement a dictionary application. The architecture model is client-server architecture. The functional requirements for the whole system are searching the meaning(s) of a word, adding a new word with its meaning(s), and removing an existing word. And the server will read and write the dictionary from the disk. The whole system is required to use sockets and threads as the lowest level of abstraction for network communication and concurrency. The non-functional requirements for the whole system are the communication should be reliable and errors should be properly managed. The non-functional requirement for the client is to have a GUI for the user to interact with.

1.1 Error handling

There are errors can occur at client and server side. The dictionary system is able to handle all these errors. Errors will be discussed and categorised by the following sections.
Sometimes, the error will be prompted in a dialog to inform the user and terminate the program. As shown in figure 1.

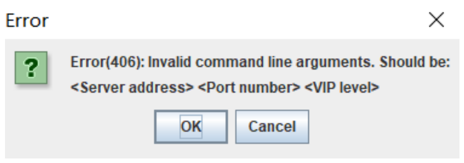

Error
Error(406): Invalid command line arguments. Should be:
<Server address> <Port number> <VIP level>

Figure 1: Example error prompt

1.1.1 Command-line input error

The following errors in table 1 will all be prompted in dialog for users as shown in figure 1.

Error Code	Error Message	Description
406	Insufficient command line arguments. Should be: <Server address> <Port number> <VIP level>	insufficient client arguments error
410	Insufficient command line arguments. Should be: <Port number> <Dictionary file path> <Thread pool size> <Inactive time> <Thread Pool Queue Limit>	insufficient server arguments error

Table 1: Command-line input error

Not only insufficient command-line input arguments error will be prompted to the user, but also invalid input parameter error will be prompted to the user as listed in the following table 2 as shown in figure 1.

Error Code	Error Message	Description
407	The port number should be an integer between 49152-65535.	wrong port number input error
408	The VIP level number should be an integer between 0-10	wrong user VIP number input error
411	The thread pool size should be an integer between 1-2147483647	wrong thread pool size input error
412	The inactive time should be an integer with second as unit between 1-1000	wrong inactive time input input error
419	The thread pool queue limit should be an integer between <Thread pool size>-2147483647	wrong thread pool size input error

Table 2: Command-line input parameter error

1.1.2 Network communication error

The following errors will occur at run-time as in table 3. As a result, error on client side will be displayed on the dashboard to inform the user. And error on server side will be displayed through the log.

Error Code	Error Message	Description
400	The words should not be empty!	empty word to be processed bad data error
402	The word's meaning should not be empty!	empty meaning to be processed bad data error
416	The word string should be meaningful(made by alphabet)!	word string input bad data error e.g.: @@@!!!****
417	The meaning string should be meaningful (made by alphabet or numbers(optional))!	meaning string input bad data error e.g.: @@@!!!****
418	The word length should be less that or equal to 45!	word too long error
401	Connection to server failed!	unsuccessful connection error, address is not reachable
420	The server is too busy please try again!	thread pool queue full error
405	Invalid response from the server!	invalid message format error for bad data
409	Invalid message format received from client!	invalid message format during communication error for bad data

Table 3: Bad Data Error during communication

1.1.3 I/O error

The following errors 413 and 414 will all be prompted in dialog for users as shown in figure 1. But error 415 will be printed on the server side.

Error Code	Error Message	Description
413	Can't read dictionary from given path!	wrong dictionary file path input error
414	The dictionary should be json formatted!	dictionary file path input wrong format
415	Error to write dictionary to disk!	writing dictionary to disk error

Table 4: I/O to and from disk error

1.1.4 Other errors

The following errors will occur for the functional requirements as in table 5. Similarly, if error is on client side, then it will be displayed on the dashboard to inform the user.

Error Code	Error Message	Description
403	The word to be added already exists in the dictionary!	duplicate word to be added error
404	No such word!	no such word found in dictionary error for search/remove

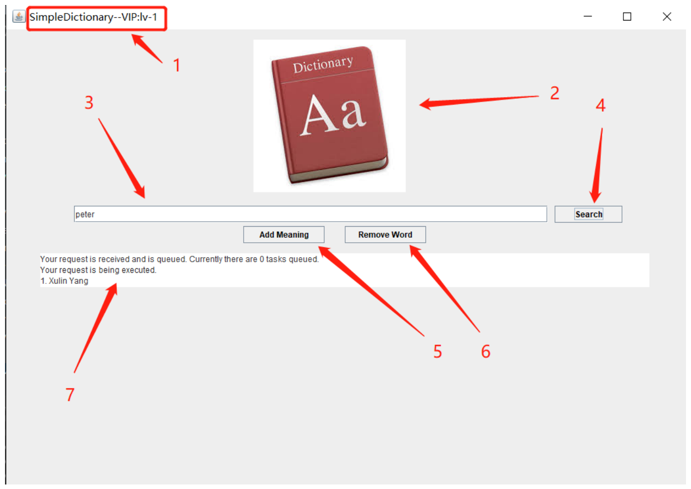
Table 5: Functional Requirements Error

2 Components

The system is made up of following components. The specification for each component will be discussed in the following sections.

2.1 Client

The client has following non-functional requirements. Firstly, it should have a GUI for users to interact with. Secondly, the user should be notified when errors occurred. Errors in table 3 and table 5 will be raised and displayed on the dashboard. The client has following functional requirement. It should process user's searching word, adding the word with its meanings and deleting the existing word in the dictionary operation from the GUI. And communicate with the server.



Your request is received and is queued. Currently there are 0 tasks queued.
Your request is being executed.
1. Xulin Yang

Figure 2: Client side GUI main page

The above figure 2 is the main page that the user will see when the user starts the program.

1. The is application name with user's VIP level. In this case, the user is using the application with VIP level 1.
2. This is the application logo.
3. This is the text input area for the user.
4. This is the button to search the word in the text input area and display the result in 7 which is the dashboard.
5. This is the button to add the word in the text input area with its meaning and display the result in 7 which is the dashboard.
6. This is the button to delete the word in the text input area with its meaning and display the result in 7 which is the dashboard.
7. This is the dashboard that display the result from the server.

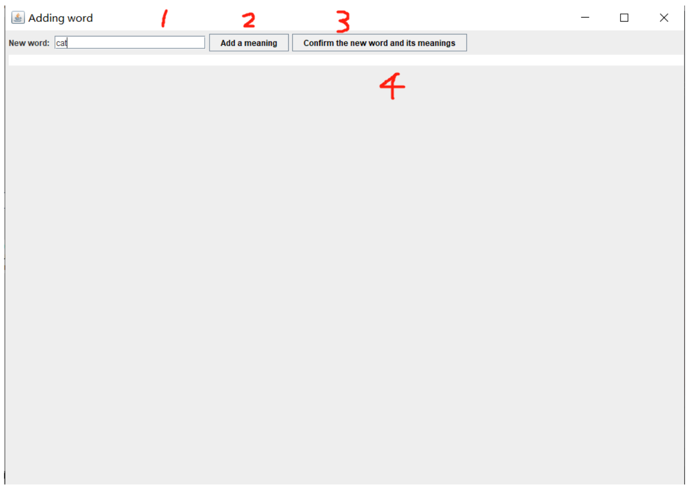


Figure 3: Client side GUI Add word page

The above figure 3 is the page that the user will see when the user clicks the *Add Word* button in figure 2.

1. This is the text area to display the word to be added and enable the user to modify the word.
2. This is the button to prompt the dialog as in figure 4 for the user to input the meaning for the word.
3. This is the button to prompt the dialog as in figure 5 for the user to submit the word with its inputted meaning to the server or cancel the operation of adding the word.
4. This is the text area to display the user inputted meaning for the word.

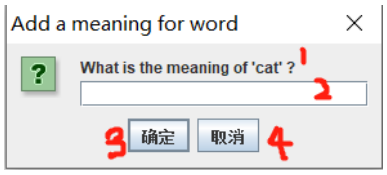


Figure 4: Client side GUI Add meaning page

The above figure 4 is the page that the user will see when the user clicks the *Add Meaning* button in figure 3.

1. This is the text area to display the word to be added.
2. This is the text area to display the user to input meaning for the word.
3. This is the button to confirm the adding of the meaning for the given word.
4. This is the button to cancel the operation to adding the meaning for the word.

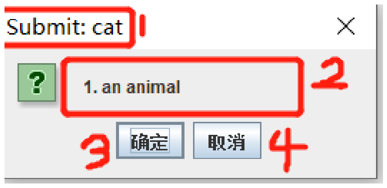


Figure 5: Client side GUI submit added word with its meaning page

The above figure 5 is the page that the user will see when the user click the *Confirm the new word and its meanings* button in figure 3.

1. This is the text area to display the word to be submitted for the adding.
2. This is the text area to display the meaning to be submitted for the adding.
3. This is the button to confirm the submission of the word with its meaning as listed above to the server.
4. This is the button to cancel the operation to adding the new word.

2.2 Server

The server is responsible for processing requests from multiple clients concurrently. Firstly, errors 409 in table 3 will be raised and logged. Secondly, it will smoothly notify the client as described in section 6.3.4 by *task code 0*. Thirdly, it will process the Task 1-3 as listed in table 6. And response with *task code -1* associated with the meaning of the word or successful adding word or successful deleting word or errors as listed in table 5 for three functional tasks respectively. The server is also responsible for dealing with the the dictionary file component as discussed in section 2.3 as well as handling the client's sockets as discussed in section 2.4. Similarly, the server uses JSON as the message format to be exchanged with the client during the communication.
The server side architecture uses the thread pool structure with the help of *Thread* class to enable the server to response to several clients' requests simultaneously. And disconnect inactive task as discussed in the section 6.3.3.

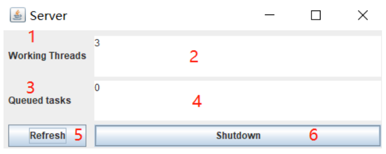


Figure 6: Server side GUI

The above figure 6 is the GUI to manage the server.

1. This is the label for the number of workers are currently working.
2. This is the text area to display the number of workers are currently working.
3. This is the label for the number of clients is currently queued.
4. This is the text area to display the number of clients is currently queued.
5. This is the button to refresh the display of server status.
6. This is the button to shutdown the server and exit the program.

2.2.1 Protocol and communication messages

Task Code	Task Message	Description
0	Server's response	value of task for receiving notification from the server
1	Client's request	value of task for search task
2	Client's request	value of task for add task
3	Client's request	value of task for delete task
-1	Server's response	value of successful task

Table 6: Server's task protocol

The request and response message will be in the following json string format between the communication of clients and the server.

```
{
    "task_code": value listed in table above or error code,
    "content": requested word or error message or successful tasks result
}
```

Listing 1: the sample request/response formatted JSON message for task 0, 1, 3, -1 and response for task 2

```
{
    "task_code": 2,
    "content": requested word or error message or successful tasks result,
    "meaning": the requested word meaning
}
```

Listing 2: the sample request/response formatted JSON message for task 2

Note: only in the request message, there will be the meaning key.

2.3 Dictionary File

The dictionary file is responsible for storing the meanings associated the word. So it should be a key value pair structure. As a result, I use the json file as the hard disk copy for the dictionary. The analysis for the choice of json file can be found in section 4.2. During the run-time, the *SimpleDictionary* class is responsible for loading/writing data from disk, process client's requests which are searching/adding/deleting words as stated in functional requirements. This component is responsible for raising errors as listed in table 5 The analysis for the implementation for the dictionary can be found in section 3.3.

2.4 Socket

The socket is responsible for coordinating the communication between the client and server. Firstly, it initiates the connection to server for the client or accepts client's connection. Secondly, it can send/receive strings during the communication. Thirdly, it can close the connection for the client or the server. At last, it is also responsible for exchange the user VIP number from the client to the server. During the run-time, it will raise the errors 401, 405 and 409 as listed in table 3. The detail for the user VIP number can be found in section 6.3.2. The implementation can be found in section 3.4 *ClientSocket* class. The application uses TCP for the communication.

3 Design Diagrams

3.1 Use Case Diagram

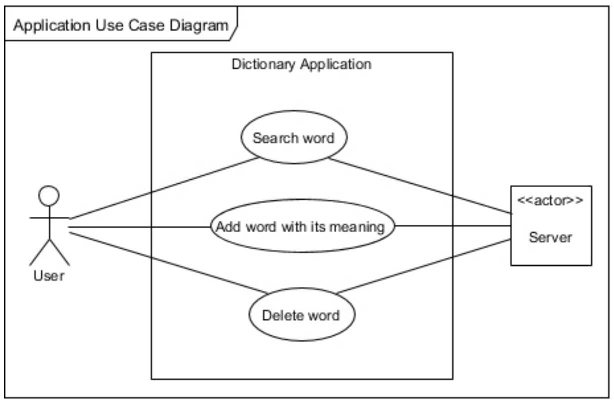


Figure 7: Application Use Case Diagram Diagram

The **system** *Dictionary Application* has the **actor** *User* and *Server*. The user has three **use cases** with the server which are "Search word", "Add word with its meaning" and "Delete word".

3.2 Client Package Class Diagram

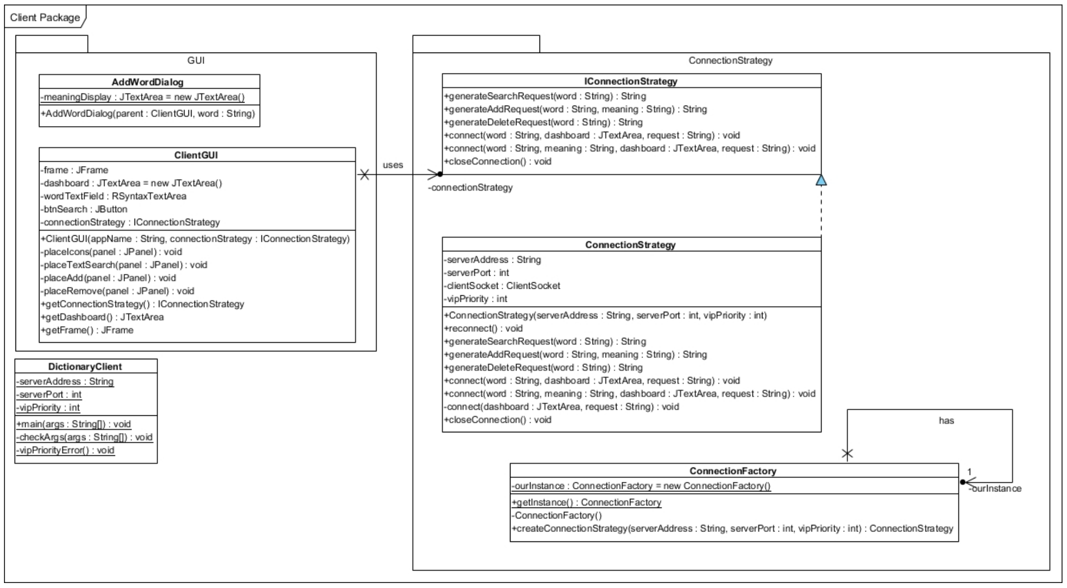

Client Package
GUI
AddWordDialog
-meaningDisplay : JTextArea = new JTextArea()
+AddWordDialog(parent : ClientGUI, word : String)
ClientGUI
-frame : JFrame
-dashboard : JTextArea = new JTextArea()
-wordTextField : RSyntaxTextArea
-btnSearch : JButton
-connectionStrategy : IConnectionStrategy
+ClientGUI(appName : String, connectionStrategy : IConnectionStrategy)
-placeIcons(panel : JPanel) : void
-placeTextSearch(panel : JPanel) : void
-placeAdd(panel : JPanel) : void
-placeRemove(panel : JPanel) : void
+getConnectionStrategy() : IConnectionStrategy
+getDashboard() : JTextArea
+getFrame() : JFrame
DictionaryClient
-serverAddress : String
-serverPort : int
-vipPriority : int
+main(args : String[]) : void
-checkArgs(args : String[]) : void
-vipPriorityError() : void
uses
-connectionStrategy
ConnectionStrategy
IConnectionStrategy
+generateSearchRequest(word : String) : String
+generateAddRequest(word : String, meaning : String) : String
+generateDeleteRequest(word : String) : String
+connect(word : String, dashboard : JTextArea, request : String) : void
+connect(word : String, meaning : String, dashboard : JTextArea, request : String) : void
+closeConnection() : void
ConnectionStrategy
-serverAddress : String
-serverPort : int
-clientSocket : ClientSocket
-vipPriority : int
+ConnectionStrategy(serverAddress : String, serverPort : int, vipPriority : int)
+reconnect() : void
+generateSearchRequest(word : String) : String
+generateAddRequest(word : String, meaning : String) : String
+generateDeleteRequest(word : String) : String
+connect(word : String, dashboard : JTextArea, request : String) : void
+connect(word : String, meaning : String, dashboard : JTextArea, request : String) : void
-connect(dashboard : JTextArea, request : String) : void
+closeConnection() : void
has
ConnectionFactory
-ourInstance : ConnectionFactory = new ConnectionFactory()
+getInstance() : ConnectionFactory
-ConnectionFactory()
+createConnectionStrategy(serverAddress : String, serverPort : int, vipPriority : int) : ConnectionStrategy
1
-ourInstance

Figure 8: Client Package Class Diagram

The DictionaryClient is the start point for the client side as it contains the main function. It stores the command line variable as static instances. This class is responsible for checking the correctness of the command-line arguments.

The GUI package contains the GUI class for the client as shown in the figure 2, figure 3 and figure 4. ClientGUI creates what presented in the figure 2. AddWordDialog creates what presented in the figure 3 and figure 4. It focuses on adding words and its meanings to the dictionary server. The GUI uses a given ConnectionStrategy as its controller.

The ConnectionStrategy package contains the class for the client to communicate with the server. It works as a controller for the client. I uses the strategy pattern for the connection to make it easier for future extension which follows the open-close principle. The ConnectionStrategy is responsible for creating the ClientSocket which is initializing the connection to the server, generating request strings with given variables, check validity of the inputs at the client side to reduce the amount of work at the server side and display server's response to the user. I use the simple factory pattern to pure fabric the creation logic to make the strategies extansible.

3.3 Server Package Class Diagram

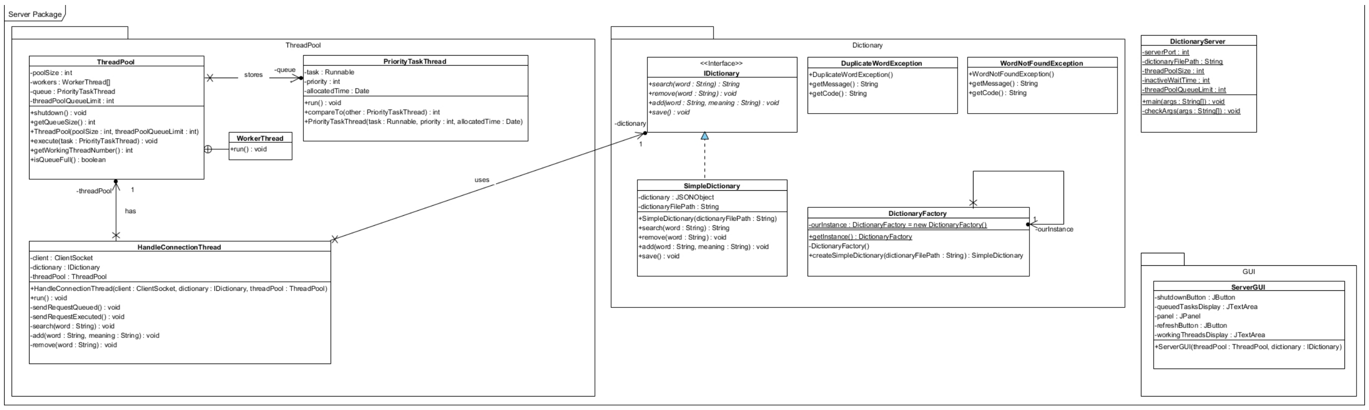

Server Package
ThreadPool
ThreadPool
-poolSize : int
-workers : WorkerThread[]
-queue : PriorityTaskThread
-threadPoolQueueLimit : int
+shutdown() : void
+getQueueSize() : int
+ThreadPool(poolSize : int, threadPoolQueueLimit : int)
+execute(task : PriorityTaskThread) : void
+getWorkingThreadNumber() : int
+isQueueFull() : boolean
stores
-queue
PriorityTaskThread
-task : Runnable
-priority : int
-allocatedTime : Date
+run() : void
+compareTo(other : PriorityTaskThread) : int
+PriorityTaskThread(task : Runnable, priority : int, allocatedTime : Date)
WorkerThread
+run() : void
-threadPool
1
has
uses
HandleConnectionThread
-client : ClientSocket
-dictionary : IDictionary
-threadPool : ThreadPool
+HandleConnectionThread(client : ClientSocket, dictionary : IDictionary, threadPool : ThreadPool)
+run() : void
-sendRequestQueued() : void
-sendRequestExecuted() : void
-search(word : String) : void
-add(word : String, meaning : String) : void
-remove(word : String) : void
Dictionary
<<Interface>>
IDictionary
+search(word : String) : String
+remove(word : String) : void
+add(word : String, meaning : String) : void
+save() : void
-dictionary
1
DuplicateWordException
+DuplicateWordException()
+getMessage() : String
+getCode() : String
WordNotFoundException
+WordNotFoundException()
+getMessage() : String
+getCode() : String
SimpleDictionary
-dictionary : JSONObject
-dictionaryFilePath : String
+SimpleDictionary(dictionaryFilePath : String)
+search(word : String) : String
+remove(word : String) : void
+add(word : String, meaning : String) : void
+save() : void
DictionaryFactory
-ourInstance : DictionaryFactory = new DictionaryFactory()
+getInstance() : DictionaryFactory
-DictionaryFactory()
+createSimpleDictionary(dictionaryFilePath : String) : SimpleDictionary
1
-ourInstance
DictionaryServer
-serverPort : int
-dictionaryFilePath : String
-threadPoolSize : int
-inactiveWaitTime : int
-threadPoolQueueLimit : int
+main(args : String[]) : void
-checkArgs(args : String[]) : void
GUI
ServerGUI
-shutdownButton : JButton
-queuedTasksDisplay : JTextArea
-panel : JPanel
-refreshButton : JButton
-workingThreadsDisplay : JTextArea
+ServerGUI(threadPool : ThreadPool, dictionary : IDictionary)

Figure 9: Server Package Class Diagram

The Server package has two packages and one class as illustrated in figure 9.

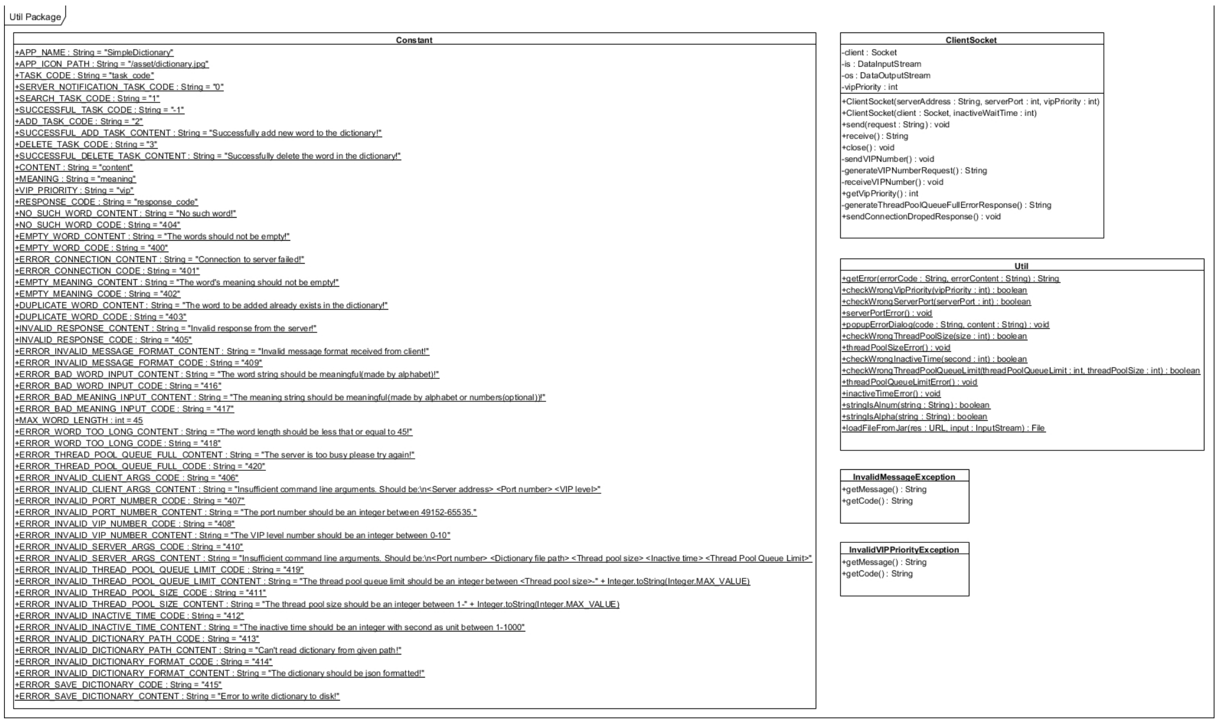
The DictionaryServer class is the start point for the server side as it contains the main function. It stores the command line variable as static instances. This class is responsible for checking the correctness of the command-line arguments.

The Dictionary package contains the model for the dictionary component. Here are two execeptions for the errors listed in table 5. I uses the simple factory pattern to hide and pure fabricate the object creation logic as well as to benefits the lazy initialization which can improve the creation efficiency. SimpleDictionary implements the IDictionary interface. I uses the adapter pattern here. The creations are handelled by the factory. The using of interfaces and polymorphism adds a level of indirection to varying APIs in other components. As it uses polymorphism pattern, it has low coupling. As a result, it is very wasy to extend and modify when we change the choice of the dictionary implementation. And the SimpleDictionary has synchronized access to search, add, delete operations for the data consistency concern.

The ThreadPool package contains the implementation of thread-pool architecture. The ThreadPool class has a inner class WorkerThread which is the worker threads in the thread-pool architecture. And ThreadPool has an array of workers with size equals to the poolSize attribute. The ThreadPool has an attribute for a queue for PriorityTaskThread for the server to be processed. The ThreadPool has an attribute for the maximum number of tasks to be queued which is ThreadPoolQueueLimit in order to handle queue memory error when too much tasks are waiting in the system. When a worker finished its work, a thread will be polled from the queue based on the priority comparison method defined in the PriorityTaskThread class. The HandleConnectionThread is the class which processes a client's connection and functional requests. It has IDictionary to perform functional tasks, ThreadPool for the smooth notificaiton (details can be found in 6.3.4) and a clientSocket for the connection with the client. It has ThreadPool to get the size of queued tasks and use this information to smoothly notify the user.

The GUI package contains the class which implements the GUI to manage the server as presented in the figure 6.

3.4 Util Package Class Diagram



```
Constant
+APP_NAME : String = "SimpleDictionary"
+APP_ICON_PATH : String = "/asset/dictionary.jpg"
+TASK_CODE : String = "task_code"
+SERVER_NOTIFICATION_TASK_CODE : String = "0"
+SEARCH_TASK_CODE : String = "1"
+SUCCESSFUL_TASK_CODE : String = "-1"
+ADD_TASK_CODE : String = "2"
+SUCCESSFUL_ADD_TASK_CONTENT : String = "Successfully add new word to the dictionary!"
+DELETE_TASK_CODE : String = "3"
+SUCCESSFUL_DELETE_TASK_CONTENT : String = "Successfully delete the word in the dictionary!"
+CONTENT : String = "content"
+MEANING : String = "meaning"
+VIP_PRIORITY : String = "vip"
+RESPONSE_CODE : String = "response_code"
+NO_SUCH_WORD_CONTENT : String = "No such word!"
+NO_SUCH_WORD_CODE : String = "404"
+EMPTY_WORD_CONTENT : String = "The words should not be empty!"
+EMPTY_WORD_CODE : String = "400"
+ERROR_CONNECTION_CONTENT : String = "Connection to server failed!"
+ERROR_CONNECTION_CODE : String = "401"
+EMPTY_MEANING_CONTENT : String = "The word's meaning should not be empty!"
+EMPTY_MEANING_CODE : String = "402"
+DUPLICATE_WORD_CONTENT : String = "The word to be added already exists in the dictionary!"
+DUPLICATE_WORD_CODE : String = "403"
+INVALID_RESPONSE_CONTENT : String = "Invalid response from the server!"
+INVALID_RESPONSE_CODE : String = "405"
+ERROR_INVALID_MESSAGE_FORMAT_CONTENT : String = "Invalid message format received from client!"
+ERROR_INVALID_MESSAGE_FORMAT_CODE : String = "409"
+ERROR_BAD_WORD_INPUT_CONTENT : String = "The word string should be meaningful(made by alphabet)!"
+ERROR_BAD_WORD_INPUT_CODE : String = "416"
+ERROR_BAD_MEANING_INPUT_CONTENT : String = "The meaning string should be meaningful(made by alphabet or numbers(optional))!"
+ERROR_BAD_MEANING_INPUT_CODE : String = "417"
+MAX_WORD_LENGTH : int = 45
+ERROR_WORD_TOO_LONG_CONTENT : String = "The word length should be less that or equal to 45!"
+ERROR_WORD_TOO_LONG_CODE : String = "418"
+ERROR_THREAD_POOL_QUEUE_FULL_CONTENT : String = "The server is too busy please try again!"
+ERROR_THREAD_POOL_QUEUE_FULL_CODE : String = "420"
+ERROR_INVALID_CLIENT_ARGS_CODE : String = "406"
+ERROR_INVALID_CLIENT_ARGS_CONTENT : String = "Insufficient command line arguments. Should be:\n<Server address> <Port number> <VIP level>"
+ERROR_INVALID_PORT_NUMBER_CODE : String = "407"
+ERROR_INVALID_PORT_NUMBER_CONTENT : String = "The port number should be an integer between 49152-65535."
+ERROR_INVALID_VIP_NUMBER_CODE : String = "408"
+ERROR_INVALID_VIP_NUMBER_CONTENT : String = "The VIP level number should be an integer between 0-10"
+ERROR_INVALID_SERVER_ARGS_CODE : String = "410"
+ERROR_INVALID_SERVER_ARGS_CONTENT : String = "Insufficient command line arguments. Should be:\n<Port number> <Dictionary file path> <Thread pool size> <Inactive time> <Thread Pool Queue Limit>"
+ERROR_INVALID_THREAD_POOL_QUEUE_LIMIT_CODE : String = "419"
+ERROR_INVALID_THREAD_POOL_QUEUE_LIMIT_CONTENT : String = "The thread pool queue limit should be an integer between <Thread pool size>-" + Integer.toString(Integer.MAX_VALUE)
+ERROR_INVALID_THREAD_POOL_SIZE_CODE : String = "411"
+ERROR_INVALID_THREAD_POOL_SIZE_CONTENT : String = "The thread pool size should be an integer between 1-" + Integer.toString(Integer.MAX_VALUE)
+ERROR_INVALID_INACTIVE_TIME_CODE : String = "412"
+ERROR_INVALID_INACTIVE_TIME_CONTENT : String = "The inactive time should be an integer with second as unit between 1-1000"
+ERROR_INVALID_DICTIONARY_PATH_CODE : String = "413"
+ERROR_INVALID_DICTIONARY_PATH_CONTENT : String = "Can't read dictionary from given path!"
+ERROR_INVALID_DICTIONARY_FORMAT_CODE : String = "414"
+ERROR_INVALID_DICTIONARY_FORMAT_CONTENT : String = "The dictionary should be json formatted!"
+ERROR_SAVE_DICTIONARY_CODE : String = "415"
+ERROR_SAVE_DICTIONARY_CONTENT : String = "Error to write dictionary to disk!"
```

```
ClientSocket
-client : Socket
-is : DataInputStream
-os : DataOutputStream
-vipPriority : int
+ClientSocket(serverAddress : String, serverPort : int, vipPriority : int)
+ClientSocket(client : Socket, inactiveWaitTime : int)
+send(request : String) : void
+receive() : String
+close() : void
-sendVIPNumber() : void
-generateVIPNumberRequest() : String
-receiveVIPNumber() : void
+getVipPriority() : int
-generateThreadPoolQueueFullErrorResponse() : String
+sendConnectionDropedResponse() : void
```

```
Util
+getError(errorCode : String, errorContent : String) : String
+checkWrongVipPriority(vipPriority : int) : boolean
+checkWrongServerPort(serverPort : int) : boolean
+serverPortError() : void
+popupErrorDialog(code : String, content : String) : void
+checkWrongThreadPoolSize(size : int) : boolean
+threadPoolSizeError() : void
+checkWrongInactiveTime(second : int) : boolean
+checkWrongThreadPoolQueueLimit(threadPoolQueueLimit : int, threadPoolSize : int) : boolean
+threadPoolQueueLimitError() : void
+inactiveTimeError() : void
+stringIsAlnum(string : String) : boolean
+stringIsAlpha(string : String) : boolean
+loadFileFromJar(res : URL, input : InputStream) : File
```

```
InvalidMessageException
+getMessage() : String
+getCode() : String
```

```
InvalidVIPPriorityException
+getMessage() : String
+getCode() : String
```

Figure 10: Util Package Class Diagram

The Util package has several classes as illustrated in figure 10. Its purpose is for constants and utility function to be reused in the system.

The class Contents stores constants for errors and tasks.

The class Util stores methods for parameter checking and error handling.

And two exceptions are for error 408 and 409 respectively.

The ClientSocket class is responsible for the socket management for the client side and the server side. As you can see there are two constructors, one for each side respectively. The class hides the user VIP level number sending process as discussed in 6.3.2. For the client side, once the connection is established, it will generate the message for user VIP level number and send it to the server. For the server side, once the connection is accepted, it will start to receive the user VIP level number sent from the client and make this information accessible. It is also responsible for generating the server too busy error message and send it to the client side.

3.5 High-level System Sequence Diagram

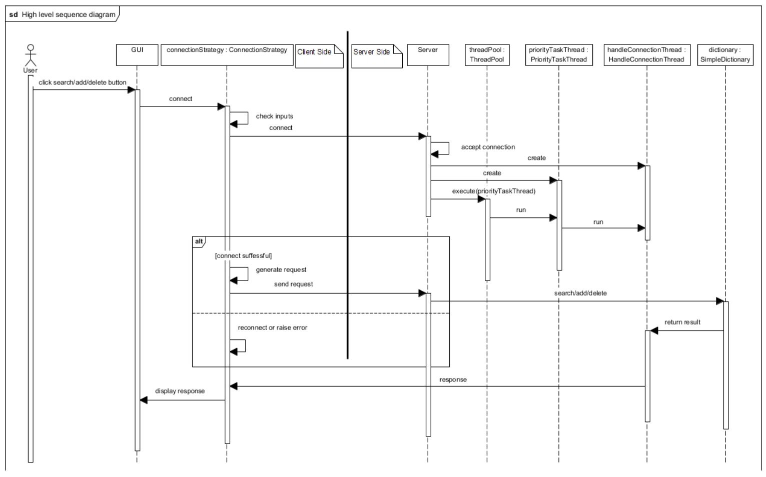


Figure 11: High level client side operation sequence diagram

The sequence for the client side operation is:

1. Client performs the operation on GUI
2. GUI invokes the operation in ConnectionStrategy which is works as the controller.
3. The ConnectionStrategy tries to initiate the connection to the server.
4. If the connection successes, then send user's VIP number to the server.
5. If the connection fails, then try to reconnect.
6. If the reconnection fails, thrn raise the error and display it to the user.
7. After the connection estabishes, the client is free to send requests to the server.
8. And receive response from the server and display it to the user.

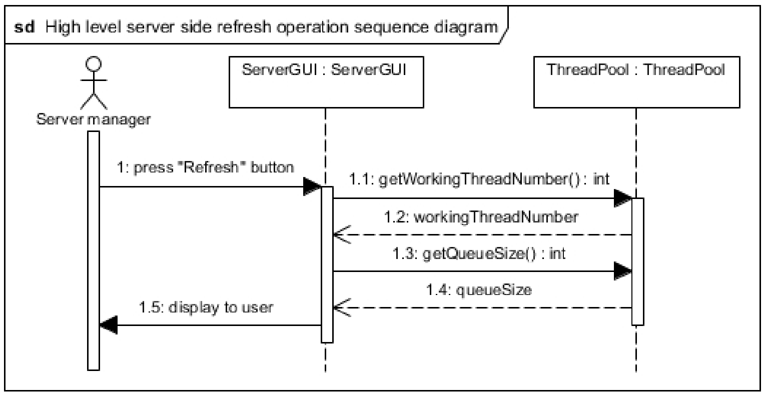

sd High level server side refresh operation sequence diagram
Server manager
ServerGUI : ServerGUI
ThreadPool : ThreadPool
1: press "Refresh" button
1.1: getWorkingThreadNumber() : int
1.2: workingThreadNumber
1.3: getQueueSize() : int
1.4: queueSize
1.5: display to user

Figure 12: High level client side Refresh operation sequence diagram

The sequence for the server side refresh operation is:

1. Server manager press the refresh button
2. GUI invokes the methods in ThreadPool to get required values to be updated on UI

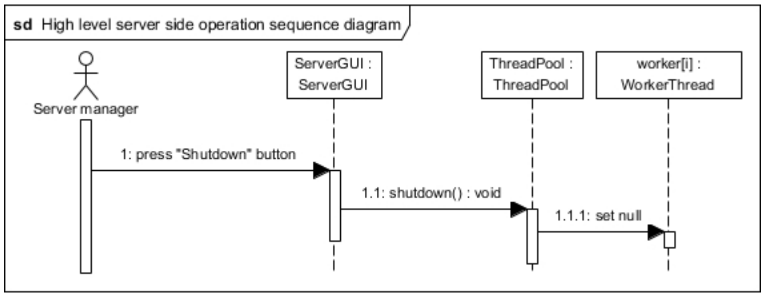


Figure 13: High level server side Shutdown operation sequence diagram

The sequence for the server side shutdown operation is:

1. Server manager press the shutdown button
2. GUI invokes the operation in ThreadPool to shutdown
3. each worker thread in ThreadPool shutdown by setting its value to null

3.6 System Sequence Diagram

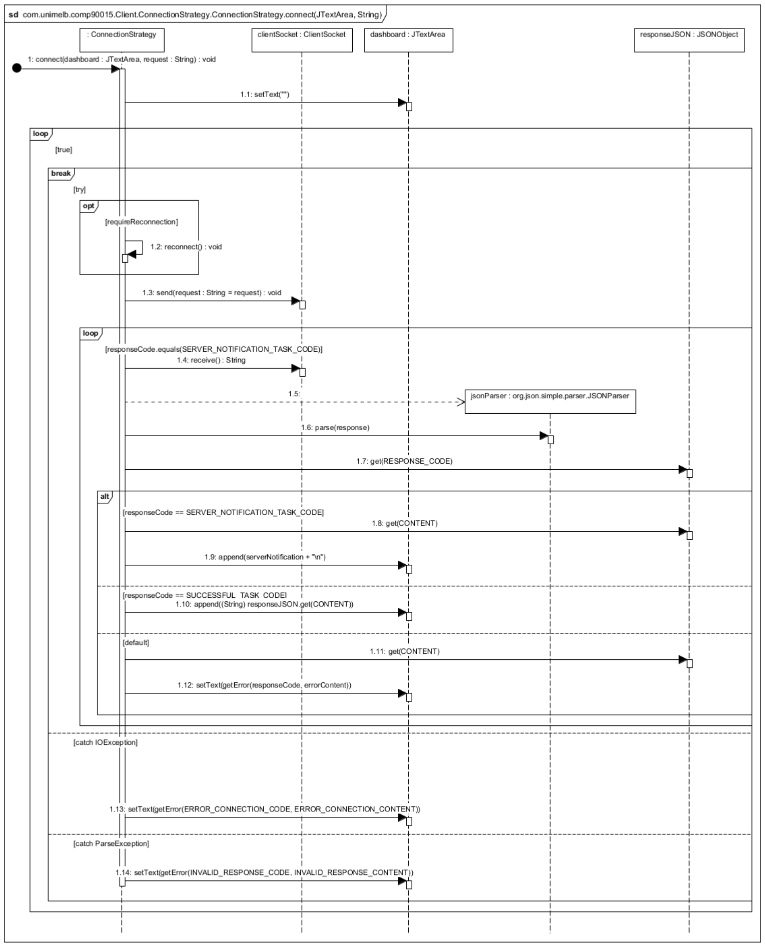

sd com.unimelb.comp90015.Client.ConnectionStrategy.ConnectionStrategy.connect(JTextArea, String)
: ConnectionStrategy
clientSocket : ClientSocket
dashboard : JTextArea
responseJSON : JSONObject
1: connect(dashboard : JTextArea, request : String) : void
1.1: setText("")
loop
[true]
break
[try]
opt
[requireReconnection]
1.2: reconnect() : void
1.3: send(request : String = request) : void
loop
[responseCode.equals(SERVER_NOTIFICATION_TASK_CODE)]
1.4: receive() : String
1.5:
jsonParser : org.json.simple.parser.JSONParser
1.6: parse(response)
1.7: get(RESPONSE_CODE)
alt
[responseCode == SERVER_NOTIFICATION_TASK_CODE]
1.8: get(CONTENT)
1.9: append(serverNotification + "\n")
[responseCode == SUCCESSFUL_TASK_CODE]
1.10: append((String) responseJSON.get(CONTENT))
[default]
1.11: get(CONTENT)
1.12: setText(getError(responseCode, errorContent))
[catch IOException]
1.13: setText(getError(ERROR_CONNECTION_CODE, ERROR_CONNECTION_CONTENT))
[catch ParseException]
1.14: setText(getError(INVALID_RESPONSE_CODE, INVALID_RESPONSE_CONTENT))

Figure 14: Client's communication to server

The figure 14 shows the sequence for the step 3-8 as presented in the figure 11. The request is already generated and inputted as a parameter. And ClientSocket is created in the creation of ConnectionStrategy.

1. Clear the dashboard to make it to be prepared for displaying new response to the client.
2. A infinite loop for the connection to the server:
3. Try to send requests to the server.
4. If the sending fails, the set reconnection true and goes to next iteration.
5. If the reconnection fails, then raise the connection fail error and break the loop.
6. If the sending success, then read and parse for various responses from the server. After this, display server's response on the dashboard and break the loop.

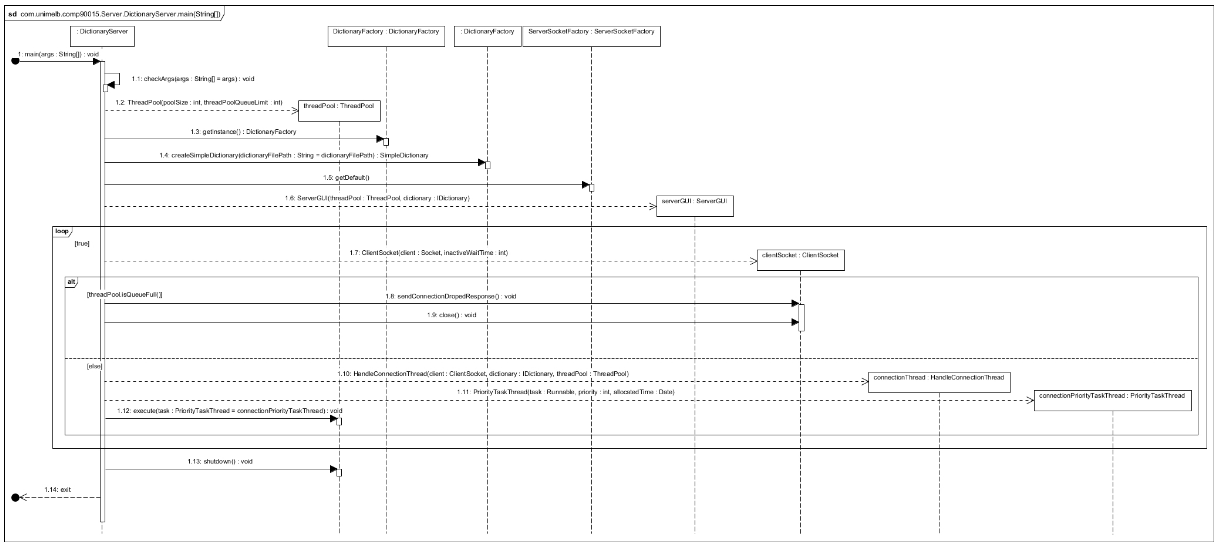

sd com.unimelb.comp90015.Server.DictionaryServer.main(String[])
: DictionaryServer
DictionaryFactory : DictionaryFactory
: DictionaryFactory
ServerSocketFactory : ServerSocketFactory
1: main(args : String[]) : void
1.1: checkArgs(args : String[] = args) : void
1.2: ThreadPool(poolSize : int, threadPoolQueueLimit : int)
threadPool : ThreadPool
1.3: getInstance() : DictionaryFactory
1.4: createSimpleDictionary(dictionaryFilePath : String = dictionaryFilePath) : SimpleDictionary
1.5: getDefault()
1.6: ServerGUI(threadPool : ThreadPool, dictionary : IDictionary)
serverGUI : ServerGUI
loop
[true]
1.7: ClientSocket(client : Socket, inactiveWaitTime : int)
clientSocket : ClientSocket
alt
[threadPool.isQueueFull()]
1.8: sendConnectionDropedResponse() : void
1.9: close() : void
[else]
1.10: HandleConnectionThread(client : ClientSocket, dictionary : IDictionary, threadPool : ThreadPool)
connectionThread : HandleConnectionThread
1.11: PriorityTaskThread(task : Runnable, priority : int, allocatedTime : Date)
connectionPriorityTaskThread : PriorityTaskThread
1.12: execute(task : PriorityTaskThread = connectionPriorityTaskThread) : void
1.13: shutdown() : void
1.14: exit

Figure 15: Server's Start

The figure 15 shows the sequence for the initialization of the server application.

1. Check the correctness of the inputted parameters.
2. Create the ThreadPool with threadPoolSize
3. Create the Dictionary object by loading the dictionary from the disk.
4. Create the server Socket
5. Create the server GUI as shown in the figure 6
6. An infinite loop to accept client's connection
7. If the queue in the ThreadPool exceeds the ThreadPoolQueueLimit, then reject the connection and send error 420 in table 3 to the client.
8. Otherwise, create ClientSocket for the socket used for communication
9. And, create the HandleConnectionThread for processing client's task
10. And, wrap the HandleConnectionThread with priority with PriorityTaskThread
11. And, execute thread in the ThreadPool
12. shutdown ThreadPool if exiting the program.

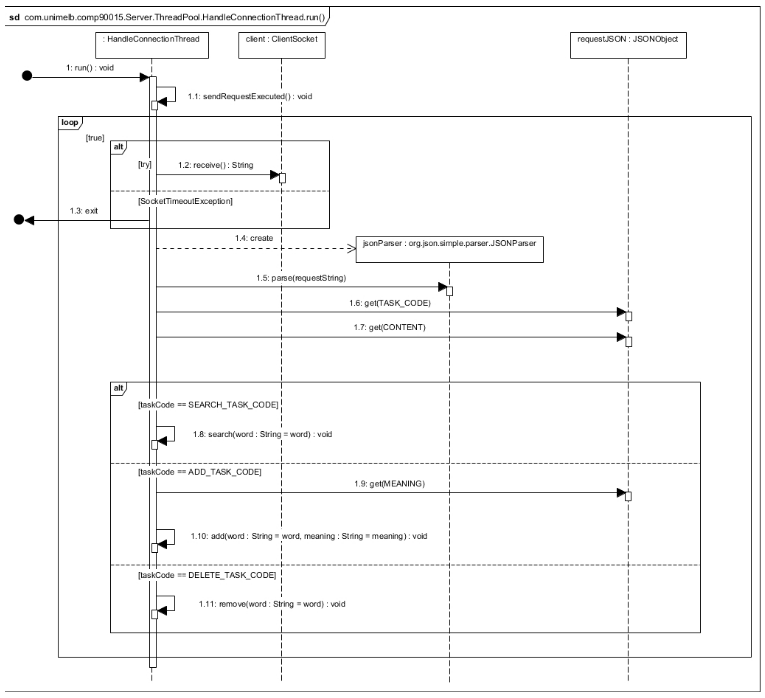

sd com.unimelb.comp90015.Server.ThreadPool.HandleConnectionThread.run()
: HandleConnectionThread
client : ClientSocket
requestJSON : JSONObject
1: run() : void
1.1: sendRequestExecuted() : void
loop
[true]
alt
[try]
1.2: receive() : String
[SocketTimeoutException]
1.3: exit
1.4: create
jsonParser : org.json.simple.parser.JSONParser
1.5: parse(requestString)
1.6: get(TASK_CODE)
1.7: get(CONTENT)
alt
[taskCode == SEARCH_TASK_CODE]
1.8: search(word : String = word) : void
[taskCode == ADD_TASK_CODE]
1.9: get(MEANING)
1.10: add(word : String = word, meaning : String = meaning) : void
[taskCode == DELETE_TASK_CODE]
1.11: remove(word : String = word) : void

Figure 16: Server's connection handling

The figure 16 shows the sequence for the server processing user's tasks in the running worker thread.

1. Notify the user the task starts processing
2. An infinite loop to receive requests from the client
3. If receive fails, then the client has disconnected and exit the thread.
4. Receive the request from the user
5. Parse the request
6. Using the Dictionary to solve the task and send the task result or any errors back to the client

4 Analysis

4.1 Dictionary Server Architecture

We have following choices of server architecture available, which are Thread-pool Architecture, Thread-per-request Architecture and Thread-per-connection Architecture according to the specification. I am going to analysis the benefit and disadvantage for each of them in the following sections as well as justify the choice for the dictionary system.

4.1.1 Thread-pool Architecture

The thread-pool architecture generally has following advantages:

1. provide fast response time because workers already created when request comes
2. easy to implement
3. bound the number of operating system resources as workers consume
4. be able to treat requests with various priorities
5. request queue provides a straightforward producer/consumer design

However, it generally has following disadvantages:

1. inflexibility: the number of worker threads in the pool may be too few to deal adequately with the current rate of request arrival
2. excessive context switching and the synchronization required to manage the request queue: the high level of switching between the I/O and worker threads as they manipulate the shared queue
3. request-level priority inversion caused by connection multiplexing: Since different priority requests share the same transport connection, a high-priority request may wait until a low-priority request that arrived earlier is processed.

In this case, our system can benefit from all the advantages listed above. The system is enabled to **response quickly**, **fast** to build, bound the resource limit for better **scalability**, **prioritize** requests for various demand, synchronize for better **thread security** and process requests **concurrently**. For the disadvantages, inflexibility can be a short-come, but it can be solved in future by dynamically change the number of threads in the thread pool. Context switching can be a problem, but it can be further solved by the method described in [1]. However, I didn't make improvements in these two points in the project. As the implementation of the improvements is difficult. What's more, for the dictionary application, the thread pool architecture's advantage benefits the system so much which make these advantages negligible. The last one can be our strategy for the **economic benefit**. As the higher VIP priority level the user's request has, the sooner the request to be processed by the system. This will encourage the clients to purchase the VIP service to create **enormous economic benefits** with the design. So this one no longer becomes the disadvantage.

4.1.2 Thread-per-request Architecture

The thread-per-request architecture generally has following advantages:

1. throughput is potentially maximized because the I/O thread can create as many workers as there are outstanding requests
2. easy to implement
3. useful for handling long duration requests

However, it generally has following disadvantages:

1. the overhead of the thread creation and destruction operations. Because it can consume a large number of OS resources if many clients make requests simultaneously. Moreover, it is inefficient for short-duration requests because it incurs excessive thread creation overhead.
2. not suitable for real-time applications since the overhead of spawning a thread for each request can be non-deterministic.

In this case, it is straight-forward that our system can **benefit from the first two advantages**. We can't benefit from the third one as the task requests in the system are **not long-time requests**. These requests are expected to be responded as fast as possible. Thus the first disadvantage will be the **main drawback** for the system. And the second disadvantage won't affect the system as the system doesn't require real-time response. And compared to thread pool, it has no prioritization between users.

4.1.3 Thread-per-connection Architecture

The thread-per-connection architecture generally has following advantages:

1. lower thread management overheads compared with the thread-per-request architecture
2. easy to implement
3. **extremely** useful for handling long duration requests

However, it generally has following disadvantages:

1. clients may be delayed while a worker thread has several outstanding requests but another thread has no work to perform
2. does not support load balancing effectively

In this case, the difference between the thread-per-connection architecture and the thread pool architecture is that thread-per-connection has **no prioritization between users**. The thread pool architecture has a limit on the threads to be processed at the same time to make the system resource usage **scalable**. Compared to the thread-per-request architecture, it has some benefits in thread management overheads. But it has the main drawback that clients may be delayed and no support load balancing effectively make the system hard to be scalable.

In the context of the challenges for the distributed system:

1. *Heterogeneity*: The application is .jar file which can be run on any OS that has a JVM of a compatible version installed.
2. *Transparency*: The system achieves the failure transparency as it has error handling and notify the user with the error messages while the system continuous to work.
3. *Failure handling*: As mentioned above, the system has error handling.
4. *Openness*: The communication protocol between the client and server side is described above.
5. *Security*: The application uses TCP for the network communication which is reliable.
6. *Scalability*: The server has ThreadPoolQueueLimit to make sure the server is scalable when there are too much tasks submitted. And the implementation of the ThreadPool class can has various options for resource allocation.
7. *Concurrency*: The server side uses java Thread to achieve the concurrency and has synchronized access to SimpleDictionary operations to achieve the shared object concurrent consistenty.

To sum it up, although both thread-per-connection architecture and thread-per-request architecture have some benefits in the design of the system, there are drawbacks as mentioned above that cannot be avoided. What's more, the thread pool architecture has more advantages compared to these two advantages. And its disadvantages can be avoided by future improvement or justified by the system design. As a result, I choose the thread pool architecture as the server architecture.

4.2 Message exchange protocol data format

There are many options available such as XML, JSON, Java Object Serialization. I didn't choose Java Object Serialization because it is not human readable which makes the communication hard to debug. While XML and JSON are human readable. I finally choose JSON because there is some commonly used java JSON library which make the system more reliable and easy to implement. What's more, the JSON object is naturally a key-value data structure which makes it more suitable for the dictionary data model. So I finally choose JSON as the message exchange protocol data format as well as the data format for the dictionary file.

5 Conclusion

In conclusion, a user-friendly dictionary application is delivered. The client side can search a word, add a new word with its meanings or delete a word in the dictionary. The multi-threaded server is implemented with thread-pool architecture for the efficiency concern. The single server is able to process multiple client's connecetion concurrently. Various errors are handelled by the system properly.

6 Excellence Elements

6.1 Excellent Implementation

6.1.1 Application Logo

The user GUI has a logo to directly tell the user what this application is about as soon as the user open the application as shown in the figure 2.

6.1.2 Add meanings one-by-one

User can add multiple meaning for one word as shown in the figure 3 and the figure 4.

6.1.3 Auto correction

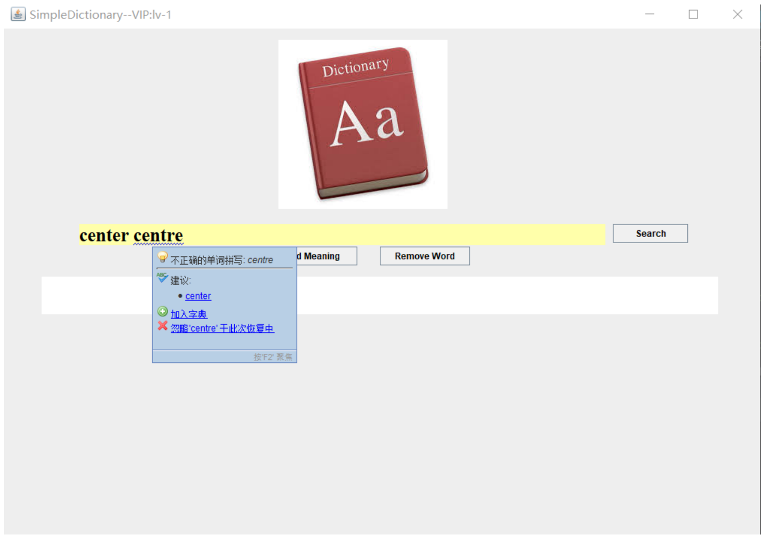


Figure 17: Auto correction

On my computer, the only available system language is Chinese so you will see Chinese description in the above figure. But I will explain what happened. As you can see "center" is a correct word, so there is no red line under it. However "centre" is British spelling, and it is not uniform with the US spelling so it can be auto corrected and a red line is under it. Just hovering the mouse on the word and a correction dislog will popup. And other word like: "asd" will also be detected and suggested with the possible correct word.

6.1.4 Shortcut key

Press "enter" when typing text in the text field (3 in figure 2) will directly search the word instead of pressing the "Search" button.

6.1.5 Smooth user request notification

For details see 6.3.4.

6.1.6 Text Input GUI

The text word input box requires the word not be empty and not with length larger than 45 on the client side to reduce the work load on the server side and inform the user to input meaningful words.

6.2 Excellent Report

6.2.1 Use Case Diagram

A use case diagram is given in figure 7.

6.2.2 Sequence Diagram

Both high-level and system sequence diagram presented.

6.2.3 Graphs

All the following figures in the report are presented with explanation.

List of Figures

6.2.4 Analysis

The class design is justified with design pattern. The sequences of interaction are give with explanation. The choice of the design for the system is analysized and justified.

6.3 Creativity

6.3.1 Thread-pool

Thread pool is implemented with java Thread class.

6.3.2 User VIP Priority

The client can initiate the application with a VIP level number zero to ten. The larger the number is, the more important the user's request will be. As a result, the user can enjoy a faster service compared to the user with a lower VIP level number.

6.3.3 Inactive waiting time

The server is started with a inactive time period parameter which defined the length of the period of time that the client's connection will be auto disconnected in order to improve the server efficiency. As a result, server's resources won't be occupied by the connection has no requests for a long time. The reconnection mechanism is implemented in the ConnectionStrategy class as illustrated by the figure 14.

6.3.4 Smooth user request notification

As user's connection will be queued in the ThreadPool if there are too much tasks to be executed. For a friendly user interface design. Every time a connection from the client is established, the server will send the current ThreadPool queue status message to the client by the function as shown in the listing below. And Response code *0* as mentioned in table 6. The process of the server response handelling at the client side is demonstrated in the figure 14.

Listing 1: server ThreadPool queue status notification

```
/**
 * notify client's task is queued and number of tasks queued in thread pool
 * @throws IOException io exception
 */
private void sendRequestQueued() throws IOException {
    JsonObject jsonObject = new JsonObject();
    jsonObject.addProperty(RESPONSE_CODE, SERVER_NOTIFICATION_TASK_CODE);
    jsonObject.addProperty(CONTENT, "Your␣request␣is␣received␣and␣is␣queued." +
            "␣Currently␣there␣are␣" + threadPool.getQueueSize() + "␣tasks␣queued.");
    String result = jsonObject.toString();
    client.send(result);
}
```

Similarly, when the queued task is running, a notificaiton will also be sent to the user in the same but with the message generated by the following function.

Listing 2: server task starts running notification

```
/**
 * notify client's task is under execution
 * @throws IOException io exception
 */
private void sendRequestExecuted() throws IOException {
    JsonObject jsonObject = new JsonObject();
    jsonObject.addProperty(RESPONSE_CODE, SERVER_NOTIFICATION_TASK_CODE);
    jsonObject.addProperty(CONTENT, "Your␣request␣is␣being␣executed.");
    String result = jsonObject.toString();
    client.send(result);
}
```

6.3.5 Thread pool queue limit

If there is too much tasks queued, the thread pool will reject client's connection and write error 420 in table 3 to the client.

6.3.6 Server management GUI

A GUI interface for the server manager to scale the management of the server as illustrated by the figure 6.

6.3.7 Dictionary efficient save

The dictionary is saved to disk when the server is going to be shutdown which saves the system resource.

References

[1] Schmidt, D. Evaluating architectures for multithreaded object request brokers. In *Comms. ACM on*, Vol. 44, No. 10, pp. 54–60.