Zak Sakata

# CSE434 Socket Project Milestone

**Design of IM Application**

The design of my IM application was kept as simple as possible with the socket logic for clients residing in UDPClient.py and the socket logic for the contact server residing in UDPServer.py. When booting up either the client or server processes, the passed in port number is first validated to ensure that it is within my dedicated range of port numbers (22000 to 22499); if it's not, the lower bound of my port number range is automatically assigned to the opened socket.

By default, the UDPClient creates its socket with the same port number used by the UDPServer; however, in the event that multiple socket processes are being run on a single machine, an unused port number is found and assigned to the new socket being created to prevent conflicting processes from binding to the same port. This is done through a recursive function that first attempts to bind the socket to the same UDPServer port; if this fails, it tries to bind to the socket again but with the UDPServer port number + 1; this executes recursively until a free port number is found and successfully bound to the socket.

The UDPClient reads input from the command line and uses Python's standard argparse library to evaluate and parse the input into one of the nine commands supported by the contact server. From here, the command arguments are further validated to ensure that all necessary arguments have been provided.

After the UDPClient has successfully parsed and validated the inputted command, it is then formatted such that the command identifier and its arguments are a single string separated by the "-_-" delimiter; this string is the message sent from the client to the server. For clarity, an example is provided below.

*The following inputted command...*

```
register zak "10.120.70.106" 22000
```

*is formatted to...*

```
"register-_-zak-_-10.120.70.106-_-22000"
```

When the UDPServer receives a new message from a client, it decodes the message string and splits it by the "-_-" delimiter. This results in a list where the first element is the command identifier and any subsequent elements are arguments for the command. For clarity, an example is provided below.

*The following inputted command...*

```
"register-_-zak-_-10.120.70.106-_-22000"
```

*is parsed to...*

```
["register", "zak", "10.120.70.106", "22000"]
```

Since all the milestone commands pertain to database actions, the UDPServer simply calls methods in ContactDatabase.py with the proper arguments after parsing the command args from the client into a list. For every ContactDatabase method, it always returns a 2-tuple. The first element of the tuple is the return code (see status code table below) and the second element includes any data to be printed on the client-side (used for commands like query-lists)—if there is no data to be printed, the second element is set to “None”. The UDPServer then formats the tuple such that the first and second element of the tuple are converted and combined into a single string separated again by the “-_-” delimiter; this string is sent back as the response from the server to the client.

Upon receiving a response back from the contact server, the UDPClient decodes and splits the message by the “-_-” delimiter to create a two-element list where the first element is the return code and second element is the data to be printed. If the return code was a success, it prints the data iff the data is not “None”; if the command failed, indicated by a failure return code, its respective failure message is printed.

**Design and Implementation Decisions**

When designing and implementing any project, I make it a clear goal to abstract and separate chunks of related logic into their own modules or files. The end result is a codebase that is DRY and highly decoupled allowing for easier maintenance and debugging down the road. When I find myself rewriting similar code multiple times, I make it a point to frequently refactor and create functions/methods that can take in a dynamic range of arguments while still producing a similarly structured output.

Any constant values such as the message delimiter or success/failure return codes are stored in Constants.py; this was done to prevent the need to continuously use hard coded constant values while also enabling the flexibility to change any constant value very easily. Additionally as a side effect, the code is easier to read and understand since all constants need to be named (e.g. 459 is less meaningful than Constants.FAILURE_CODE_UNREGISTERED_CONTACT).

The contact database is a Python module that only uses dictionaries to store all contact data. Active contacts are stored in a dictionary where each key is the contact name and its value is another dictionary with the keys, “ip” and “port” containing their respective values. Contact lists are stored in a dictionary where the key is the contact list name and the value is a list of contact name strings which can be used as the key in the contact dictionary to retrieve the contact’s related data. This makes it fast and easy to retrieve, join, and modify data from the database.

The file structure of this project was kept simple. The src folder contains all code and directly includes UDPClient.py and UDPServer.py. The utils folder is meant for modules that have a wide range of uses and can be utilized on both the client and server-side (e.g. Constants.py). The models folder is intended to store any database related modules including ContactDatabase.py. The client and server folders contain modules/functions that are used only by either the client or server respectively.

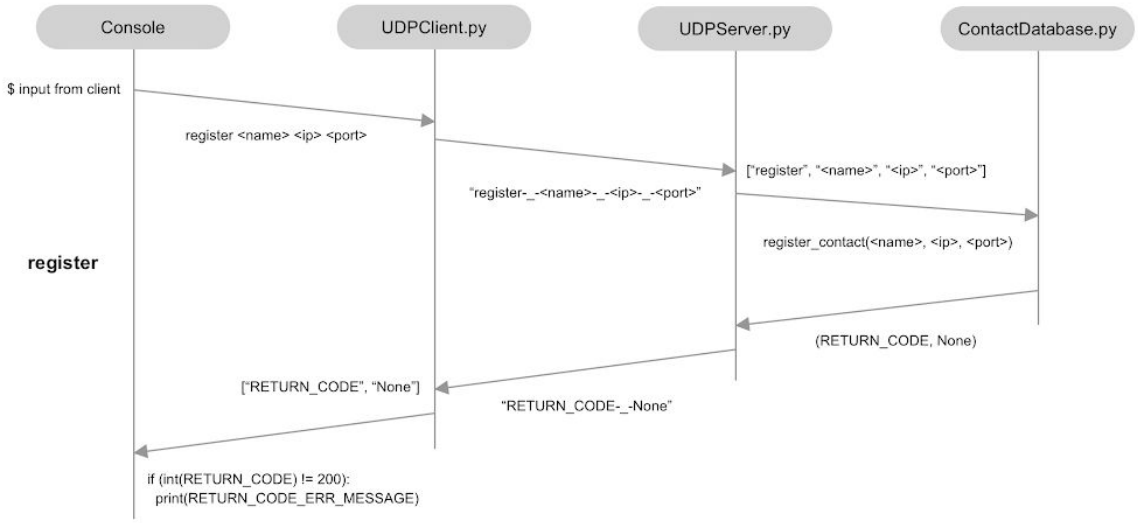
Zak Sakata

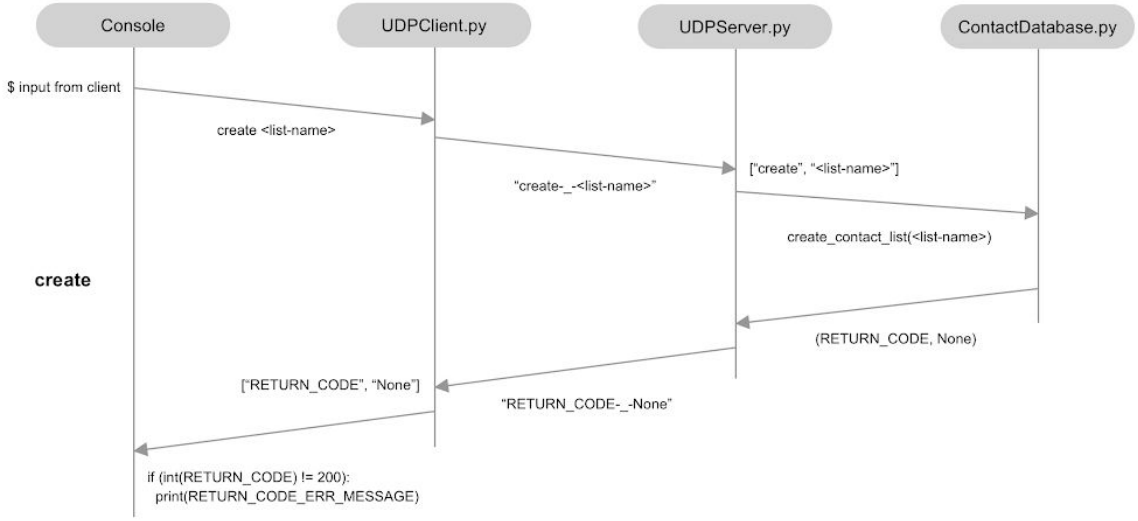
The following return code table identifies each code's meaning. Return codes were modeled after HTTP status codes and start at 455 since the last 4XX code ends at 451.

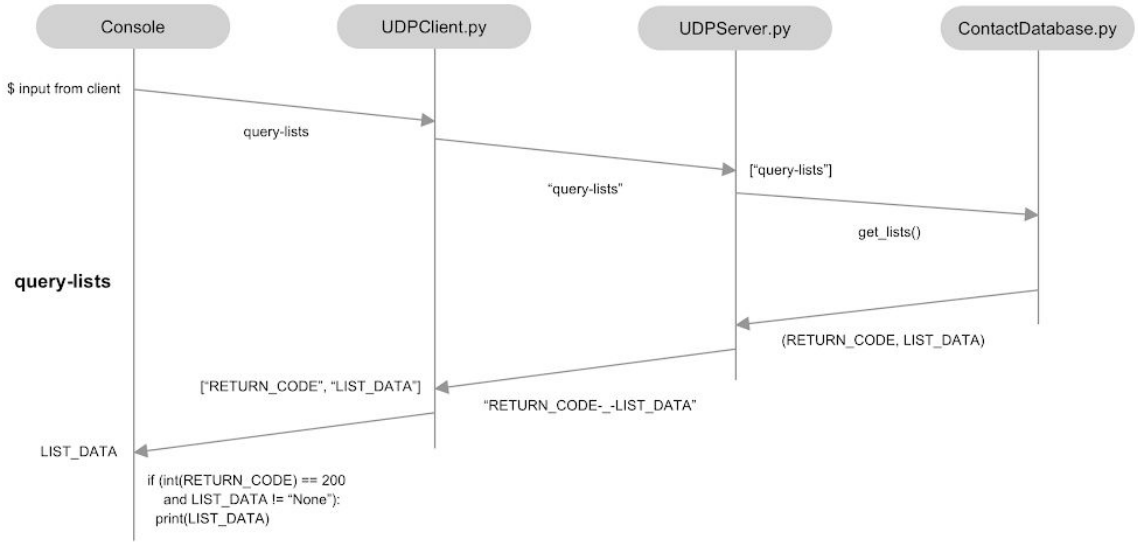
| **Return Code** | **Meaning** |
|---|---|
| 200 | The command executed successfully |
| 455 | The specified contact name already exists |
| 456 | The server has reached its max capacity of active contacts |
| 457 | The specified contact list name already exists |
| 458 | The server has reached its max capacity of contact lists |
| 459 | The specified contact name does not exist |
| 460 | The specified contact list does not exist |
| 461 | The specified contact name already exists in this list |
| 462 | The server was unable to save its data to the specified file |

Time-space diagrams for the implemented commands are on the following pages...

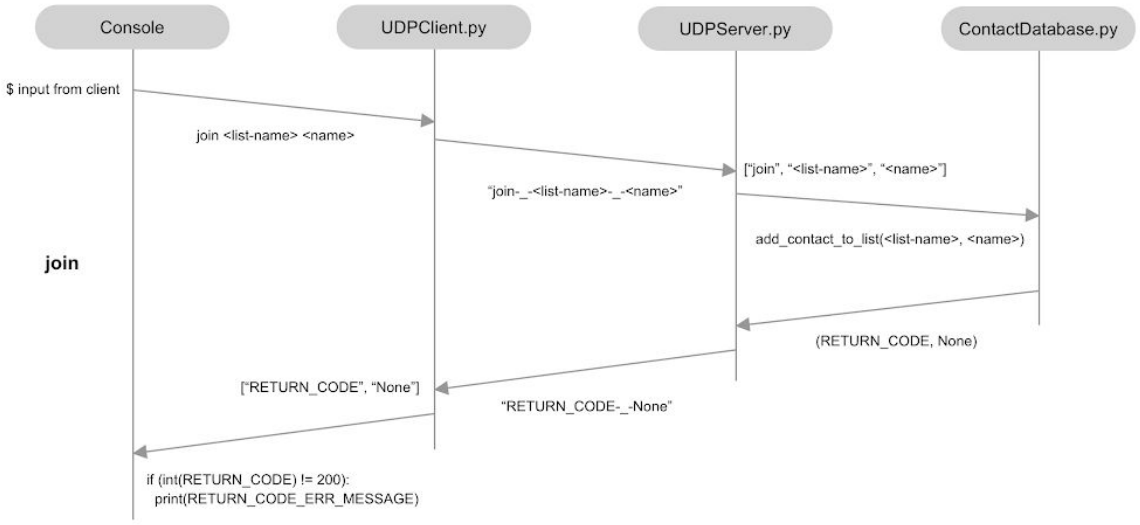
Zak Sakata

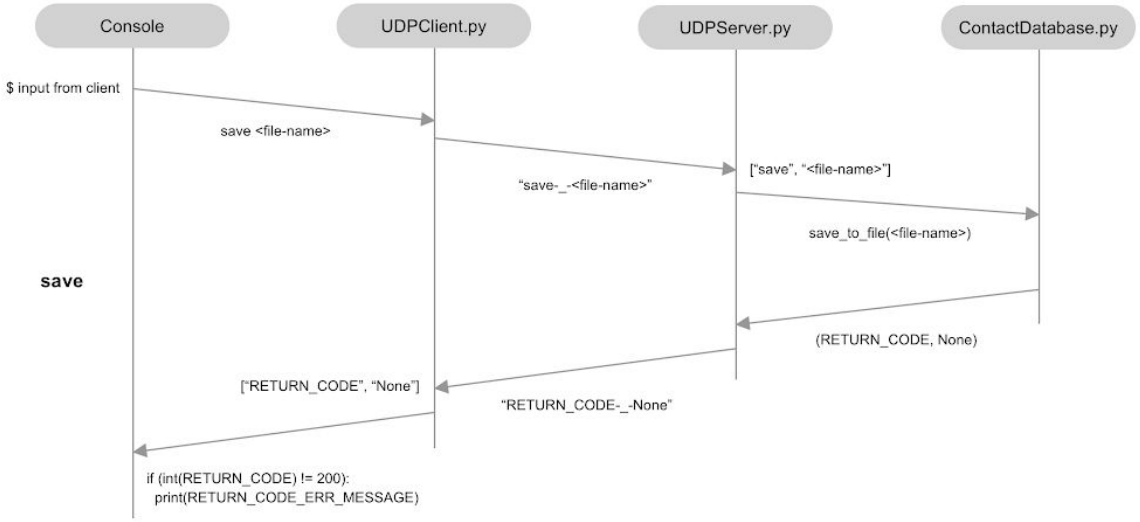

Console
UDPClient.py
UDPServer.py
ContactDatabase.py
$ input from client
register <name> <ip> <port>
["register", "<name>", "<ip>", "<port>"]
"register-_-<name>-_-<ip>-_-<port>"
register_contact(<name>, <ip>, <port>)
register
(RETURN_CODE, None)
["RETURN_CODE", "None"]
"RETURN_CODE-_-None"
if (int(RETURN_CODE) != 200):
print(RETURN_CODE_ERR_MESSAGE)

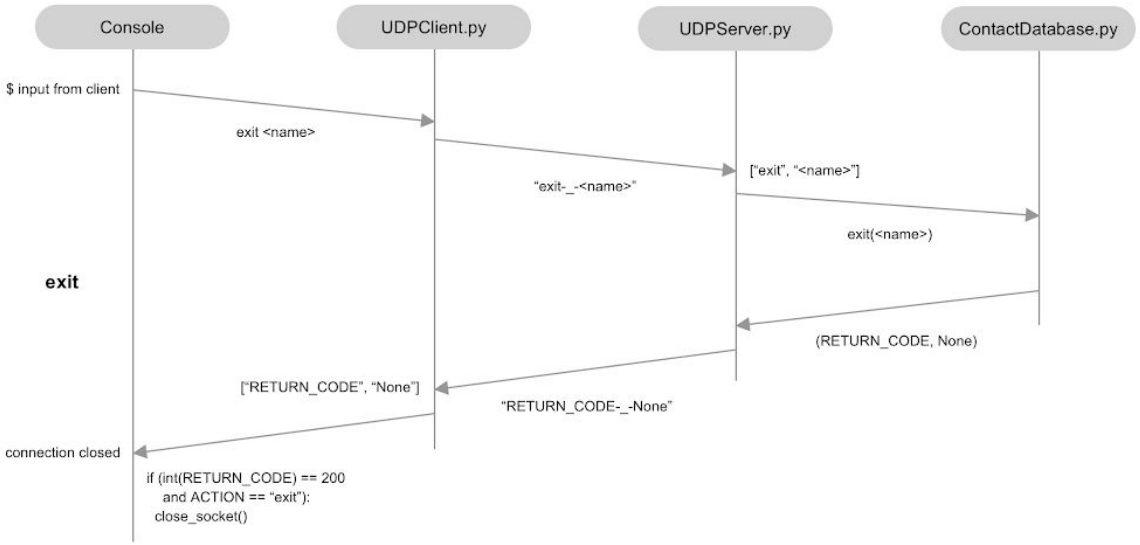

Console
UDPClient.py
UDPServer.py
ContactDatabase.py
$ input from client
create <list-name>
["create", "<list-name>"]
"create-_-<list-name>"
create_contact_list(<list-name>)
create
(RETURN_CODE, None)
["RETURN_CODE", "None"]
"RETURN_CODE-_-None"
if (int(RETURN_CODE) != 200):
print(RETURN_CODE_ERR_MESSAGE)


Console
UDPClient.py
UDPServer.py
ContactDatabase.py
$ input from client
query-lists
["query-lists"]
"query-lists"
get_lists()
query-lists
(RETURN_CODE, LIST_DATA)
["RETURN_CODE", "LIST_DATA"]
"RETURN_CODE-_-LIST_DATA"
LIST_DATA
if (int(RETURN_CODE) == 200
and LIST_DATA != "None"):
print(LIST_DATA)

Zak Sakata

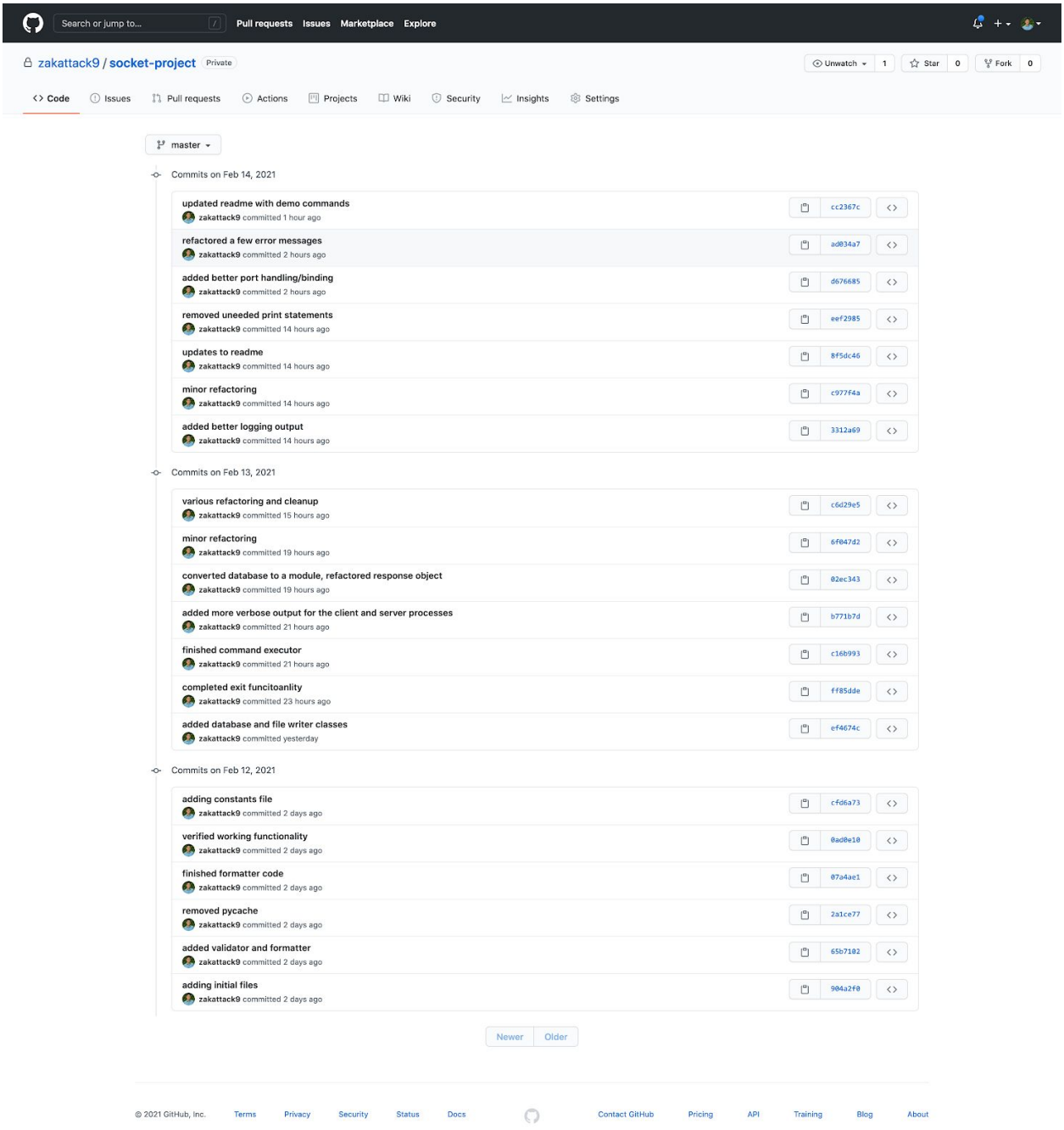

Console
UDPClient.py
UDPServer.py
ContactDatabase.py
$ input from client
join <list-name> <name>
"join-_-<list-name>-_-<name>"
["join", "<list-name>", "<name>"]
add_contact_to_list(<list-name>, <name>)
join
(RETURN_CODE, None)
["RETURN_CODE", "None"]
"RETURN_CODE-_-None"
if (int(RETURN_CODE) != 200):
print(RETURN_CODE_ERR_MESSAGE)


Console
UDPClient.py
UDPServer.py
ContactDatabase.py
$ input from client
save <file-name>
"save-_-<file-name>"
["save", "<file-name>"]
save_to_file(<file-name>)
save
(RETURN_CODE, None)
["RETURN_CODE", "None"]
"RETURN_CODE-_-None"
if (int(RETURN_CODE) != 200):
print(RETURN_CODE_ERR_MESSAGE)


Console
UDPClient.py
UDPServer.py
ContactDatabase.py
$ input from client
exit <name>
"exit-_-<name>"
["exit", "<name>"]
exit(<name>)
exit
(RETURN_CODE, None)
["RETURN_CODE", "None"]
"RETURN_CODE-_-None"
connection closed
if (int(RETURN_CODE) == 200
and ACTION == "exit"):
close_socket()

Zak Sakata

## Screenshot of GitHub commits




---

## Link to demo video on YouTube:

https://youtu.be/V6trnoBKt0s