COMP28112 Exercise 1 Report

Account: a65914zc
Date: 1st March, 2023

Code Explanation

The python file simulated that two clients sent and received messages to each other on a provided server with the designed protocol. According to the lecture, some items of the "Four Particular Conditions of DeadLock " should be broken to avoid the deadlock occurring. In this protocol file, the second and the third item would be broken to minimize the likelihood of the deadlock.

For the *second* condition "Hold and Wait", the original design was there should be at least one resource held for a client and this client would also apply other occupied resources at the same time. It was changed to "it was possible for one client would not hold any resource" through creating a mutex lock. For the *third* condition "No pre-emption", the original design was once a client did the operation on the server (occupy the resource), nothing would give it up unless itself. It was changed to "for any doctor operation in the function like "Robert()", the other doctor function Ned() would be called at the end, so Robert was forced to give up the server and Ned started to occupy the resource, deriving the communication to each other".

To be more specific, for the content of code, firstly, useful libraries were imported and own URL server was linked to the gitlab server file. Then, some global variables were assigned. For example, "server_tag" was viewed as the mutex lock to avoid simultaneously occupying when sharing resources, so that it was possible that one client may hold no resource, breaking the second item "Hold and Wait". Doctor Robert could send the message when the tag was set to 1 and doctor Ned could send the message when the tag was set to 0. Besides, in order to *receive* the other's information, the special index was designed such as "Robert_time" and "Ned_time", representing the transmission number of this doctor in the "key" part for requests such as "server["Robert" + str(Robert_time)]" and "server["Ned" + str(Ned_time)]" (in next code area), so that doctors could gain latest communication information ("value" part for requests) from the server.

```
import im
import time
server=im.IMServerProxy("https://web.cs.manchester.ac.uk/a65914zc/comp28112_ex1/IMserver.php")
server_tag = 1
Robert_time = 0
Ned_time = 0
```

In terms of the main part, there would be two functions representing different doctors but with the same code logic. Firstly, three variables were referenced by the global, in this way, the operated variables later were the ones that were declared earlier. Then, the "server_tag" was considered, for example, Robert() would check whether the tag "server_tag" was set to 1, if not, the notice information was printed. If ture, the main operation of Robert's turn could start.

(Suppose Robert() was executing...) At first, the other doctor's latest information would be shown through the specific index.(But it will return null value initially, because two doctors both had no history on the server at the zeroth time). Then, the "key" would be added to one like "Robert_time += 1", which was helpful to update Robert's typing information to doctor Ned. And then, the given method was used to push the value to the server with the designed key and the sleep function simulated the program executing. After these things done, the tag was changed to 0. Now doctor Ned was allowed to input data in the server requests under the same logic with global variables such as "server_tag" and "Ned_time", to occupy the server resource and design the special index in requests like "server["Ned" + str(Ned_time)]", which would also terminate the process of Robert, considering the third item "No pre-emption". Finally, the communication system with the protocol avoiding the deadlock, was created between two doctors.

```python
def Robert():
    global server_tag, Robert_time, Ned_time
    if (server_tag == 1):
        print("Ned just said: " + str(server["Ned" + str(Ned_time)]) )

        Robert_time += 1
        server["Robert" + str(Robert_time)] = input("Hello, Robert! Please input your message
here: ")

        print("Sending to the server... (sleep 3s)")
        time.sleep(3)

        server_tag = 0
        print("Now, tag is set to 0! It's Ned's turn! ")
        Ned()  # GIVE UP RESOURCE
    else:
        print("Sorry, Ned is typing something, please wait! ")  # NOT ALLOWED


def Ned():
    global server_tag, Robert_time, Ned_time
    if (server_tag == 0):
        print("Robert just said: " + str(server["Robert" + str(Robert_time)]) )

        Ned_time += 1
        server["Ned" + str(Ned_time)] = input("Hello, Ned! Please input yout message here: ")

        print("Sending to the server... (sleep 3s)")
        time.sleep(3)

        server_tag = 1
        print("Now, tag is set to 1! It's Robert's turn! ")
        Robert()  # GIVE UP RESOURCE
    else:
        print("Sorry, Robert is typing something, please wait! ")  # NOT ALLOWED


Robert()
```

Run and Test code

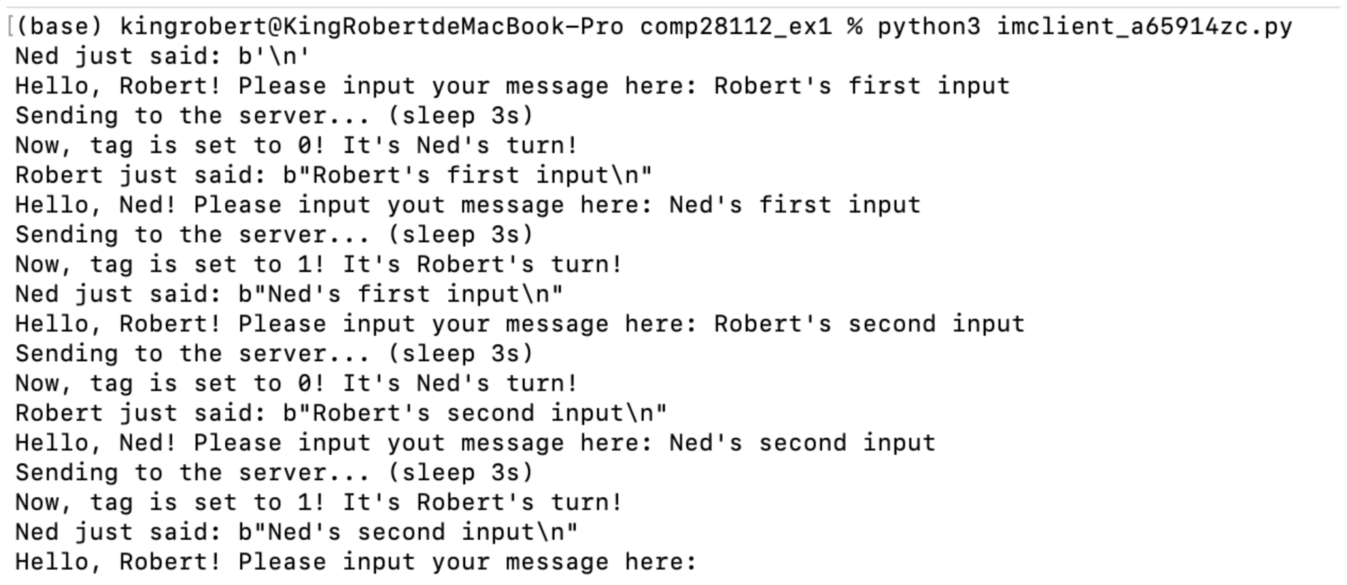
Robert() function had been called at the end of the python file and Ned() function would be called at the end of the Robert() function. Therefore, in order to start communication between two doctors, the single file could just be complied using compile command "*python3 imclient_a65914zc.py*" and did following requirements:

1. Wait Robert to input the infomation (default Robert is first)
2. Send this information to the server
3. Wait Ned to input the information

4. Display what Robert just sent
5. Continue the communication, wait Robert to input the information

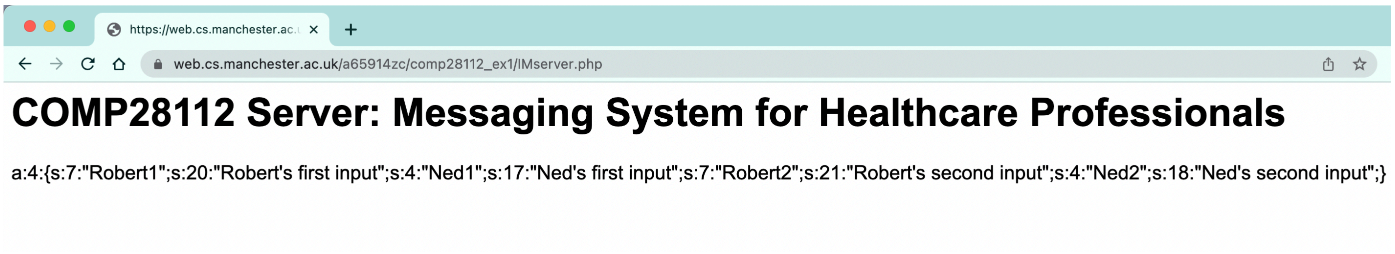
At the same time, the server would receive related data and update. Because there was no value history for Robert0 or Ned0 on the server, it would return the default null value. *The "Control + C" could be used to terminate the program.* These two figures showed the process of results.

· **Terminal:**



```
[(base) kingrobert@KingRobertdeMacBook-Pro comp28112_ex1 % python3 imclient_a65914zc.py
 Ned just said: b'\n'
 Hello, Robert! Please input your message here: Robert's first input
 Sending to the server... (sleep 3s)
 Now, tag is set to 0! It's Ned's turn!
 Robert just said: b"Robert's first input\n"
 Hello, Ned! Please input yout message here: Ned's first input
 Sending to the server... (sleep 3s)
 Now, tag is set to 1! It's Robert's turn!
 Ned just said: b"Ned's first input\n"
 Hello, Robert! Please input your message here: Robert's second input
 Sending to the server... (sleep 3s)
 Now, tag is set to 0! It's Ned's turn!
 Robert just said: b"Robert's second input\n"
 Hello, Ned! Please input yout message here: Ned's second input
 Sending to the server... (sleep 3s)
 Now, tag is set to 1! It's Robert's turn!
 Ned just said: b"Ned's second input\n"
 Hello, Robert! Please input your message here:
```

· **Server:**


https://web.cs.manchester.ac.
web.cs.manchester.ac.uk/a65914zc/comp28112_ex1/IMserver.php

COMP28112 Server: Messaging System for Healthcare Professionals

a:4:{s:7:"Robert1";s:20:"Robert's first input";s:4:"Ned1";s:17:"Ned's first input";s:7:"Robert2";s:21:"Robert's second input";s:4:"Ned2";s:18:"Ned's second input";}

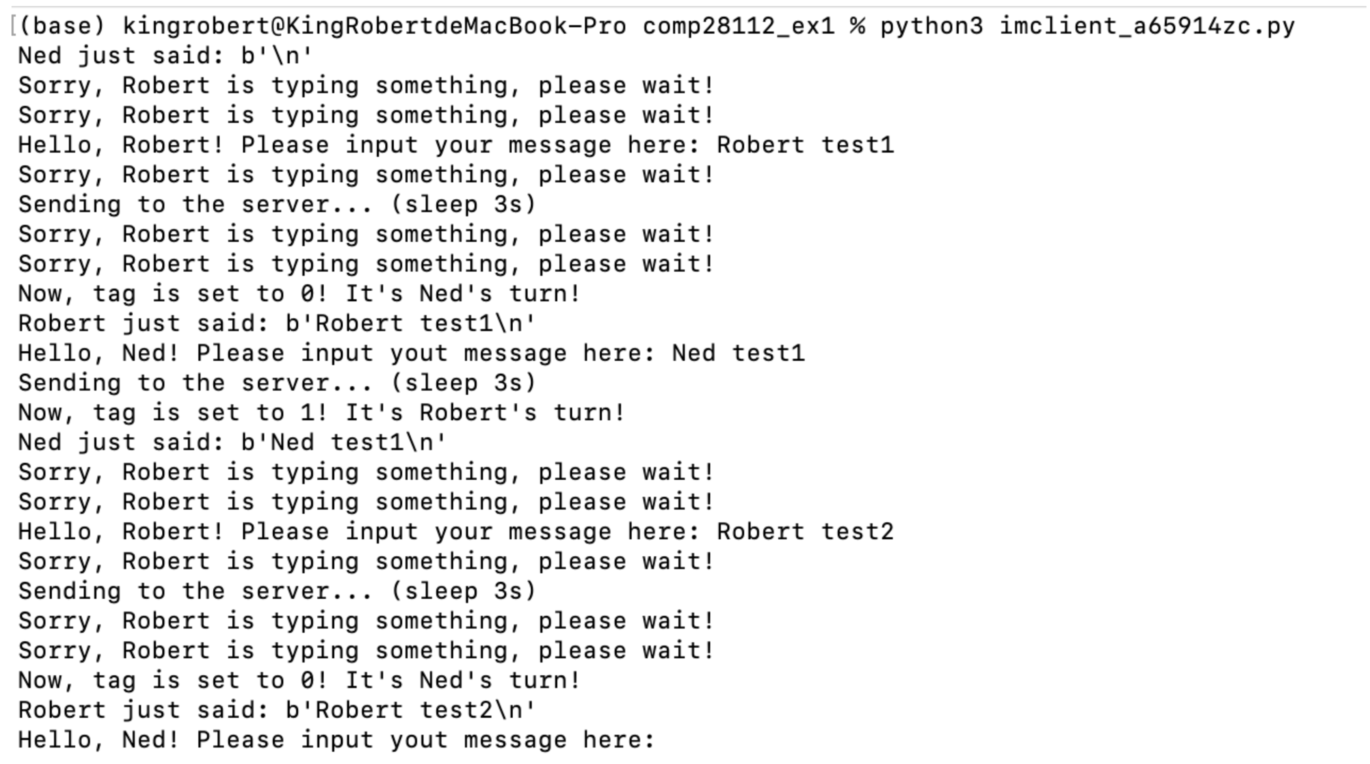
To test the reliability of this protocol, a simple malicious test was designed: five Ned() functions would be deliberately called when Robert() was executing (when server_tag=1) and observe results of terminal. (The python file in the zip commented these five Ned() function.)

```
def Robert():
    global server_tag, Robert_time, Ned_time
    if (server_tag == 1):
        print("Ned just said: " + str(server["Ned" + str(Ned_time)]) )
        Ned()  # TEST!
        Robert_time += 1
        Ned()  # TEST!
        server["Robert" + str(Robert_time)] = input("Hello, Robert! Please input your message
here: ")
        Ned()  # TEST!
        print("Sending to the server... (sleep 3s)")
        Ned()  # TEST!
        time.sleep(3)
        Ned()  # TEST!
        server_tag = 0
        print("Now, tag is set to 0! It's Ned's turn! ")
        Ned()  # GIVE UP RESOURCE
    else:
        print("Sorry, Ned is typing something, please wait! ")  # NOT ALLOWED
```

· Test results:

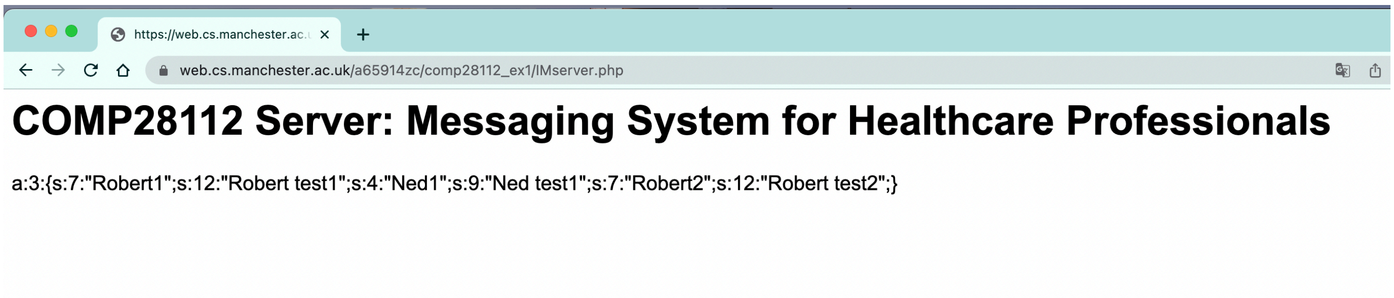
As we could observe: "Sorry, Robert is typing something, please wait! " happened five times, which mean when the Ned() asked for the server resource, the system rejected all these unappropriate requests using the mutex lock with the warning and Ned was allowed to hold the resource until server_tag was changed to 0. Finally, the server still could show expected results.

· Terminal:



```
[(base) kingrobert@KingRobertdeMacBook-Pro comp28112_ex1 % python3 imclient_a65914zc.py
Ned just said: b'\n'
Sorry, Robert is typing something, please wait!
Sorry, Robert is typing something, please wait!
Hello, Robert! Please input your message here: Robert test1
Sorry, Robert is typing something, please wait!
Sending to the server... (sleep 3s)
Sorry, Robert is typing something, please wait!
Sorry, Robert is typing something, please wait!
Now, tag is set to 0! It's Ned's turn!
Robert just said: b'Robert test1\n'
Hello, Ned! Please input yout message here: Ned test1
Sending to the server... (sleep 3s)
Now, tag is set to 1! It's Robert's turn!
Ned just said: b'Ned test1\n'
Sorry, Robert is typing something, please wait!
Sorry, Robert is typing something, please wait!
Hello, Robert! Please input your message here: Robert test2
Sorry, Robert is typing something, please wait!
Sending to the server... (sleep 3s)
Sorry, Robert is typing something, please wait!
Sorry, Robert is typing something, please wait!
Now, tag is set to 0! It's Ned's turn!
Robert just said: b'Robert test2\n'
Hello, Ned! Please input yout message here:
```

· Server:


https://web.cs.manchester.ac.u
web.cs.manchester.ac.uk/a65914zc/comp28112_ex1/IMserver.php

COMP28112 Server: Messaging System for Healthcare Professionals

```
a:3:{s:7:"Robert1";s:12:"Robert test1";s:4:"Ned1";s:9:"Ned test1";s:7:"Robert2";s:12:"Robert test2";}
```

In conclusion, it could be argued that this protocol using the "server_tag" viewed as the mutex lock and alternative to each other derived a communication situation avoiding the deadlock.