

```

 01 # Here are our imports, these allow us to use functions from other libraries
 02 import time
 03 import os
 04 import subprocess
 05 import socketserver
 06 import socketserver
 07 import json
 08
 09 def date_time():
10     return json.dumps({"dateTime": time.time()})
11
12 def up_time():
13     uptime = subprocess.run(['uptime', '-a'], stdout=subprocess.PIPE)
14     uptime_output = uptime.stdout.strip().decode('utf-8')
15     up_time = uptime_output.splitlines()
16     return json.dumps([{"upTime": time.time() - int(up_time[1])}))
17
18 def memory_usage():
19     return json.dumps([{"memoryUsage": os.cpu_count()}])
20
21 # Returns network connection fields in the project instructions
22 def network_connections():
23
24     result = subprocess.run(['netstat'], stdout=subprocess.PIPE)
25     jsonToReturn = []
26
27     for line in result.stdout.decode('utf-8').split("\n"):
28         if line != "":
29             protocol, receiveQueue, sendQueue, localAddress, foreignAddress, _, state, _ = line.split()
30             jsonToReturn.append({"proto": protocol, "receiveQueue": receiveQueue, "sendQueue": sendQueue, "localAddress": localAddress, "foreignAddress": foreignAddress, "state": state})
31
32     return json.dumps(jsonToReturn)
33
34
35 # TODO: Check for socket fields in the project instructions
36 def current_users():
37
38     current_users = subprocess.run(['who'], stdout=subprocess.PIPE).stdout.decode('utf-8')
39
40     for user in current_users:
41         users_string = user.split()
42         user_name = users_string[0]
43         host = users_string[1]
44
45     return json.dumps(users)
46
47 # TODO: Check for process fields in the project instructions
48 def running_processes():
49
50     processes = []
51
52     current_processes = subprocess.run(['ps'], stdout=subprocess.PIPE)
53
54     for process in current_processes:
55         processes.append(process)
56
57     return json.dumps(processes)
58
59 # Bundles our response in a readable way
60 def package_response(self, response):
61
62     self.send_header("Content-Type", "application/json")
63
64     self.end_headers()
65
66     package_response(self, response)
67
68     elif path == "/currentUsers":
69         response = current_users()
70
71     elif path == "/networkConnections":
72         response = network_connections()
73
74     elif path == "/memoryUsage":
75         response = memory_usage()
76
77     elif path == "/upTime":
78         response = up_time()
79
80     elif path == "/dateTime":
81         response = date_time()
82
83     elif path == "/":
84         response = self.wfile.read(404)
85
86     else:
87         response = self.wfile.read(404)
88
89     self.end_headers()
90
91     self.wfile.write(response)
92
93
94 # Starts our server
95 with socketserver.TCPServer(("", 3215), ServerHandler) as httpd:
96     print("Serving at port 3215...")
97     httpd.serve_forever()

```

## Iterative Socket Server

Joshua Malgeri, Andreas Ink, Colton

CNT4504 Computer Networks & Distributed Processing

Professor John Kelly

## **Introduction**

CNT4504 Computer Networks & Distributed Processing is a fascinating course that quite literally connects us all. During our class we learned the relationships of servers and clients and how they impact each other. Therefore, we were tasked to apply our learnings and understandings of server/client relationships. We built a server and client that was developed in Python 3.6 where both the server and client had simple tasks to perform. The server takes in several requests from the user and establishes a connection to the server where the requests are processed and distributed back to the user via a printed message. We aimed to examine how the server would be impacted by the number of requests from the client server and the types of requests. Further, to analyze the impact of the processing speed and the number of requests idling. This analysis documents the design process of each server from the client to the request server. We dive into the programming language and the syntax describing the steps of how it iterates and processes the requests, and encapsulates how data is collected and analyzed including at what speed the server is processing requests.

## Client-Server Setup and Configuration

```

1 import requests
2 import time
3 from statistics import mean
4
5 # This file is a client application that makes use of error handling and various components
6 # such as requests and subprocesses to handle different types of requests.
7
8 requestErrorDescription = "Request failed, please try again."
9 commands = ["d = date time", "u = up time", "n = memory usage", "n = network connections", "p = running processes", "cu = current users"]
10 iterationInstructions = "How many times should the command run?\n"
11
12 iterations = 1
13
14 serverAddress = input("IP Address of Server (http://localhost): ")
15 serverPort = input("Port of Server (8080): ")
16 baseURL = f'{serverAddress}:{serverPort}/'
17
18 inputInstructions = input(inputInstructions)
19 iterations = int(input(iterationInstructions))
20
21 averageTimes = []
22
23 # This function makes a request to the specified endpoint
24 def fetch(endpoint):
25     try:
26         return requests.get(baseURL + endpoint)
27     except:
28         print(requestErrorDescription)
29         return requestErrorDescription
30
31 # This function gets a specific endpoint and returns a number
32 def getBasicEndpoint(endpoint):
33     response = fetch(endpoint)
34     if response == requestErrorDescription:
35         return requestErrorDescription
36     try:
37         json = response.json()
38         return json[endpoint]
39     except:
40         print(requestErrorDescription)
41         return
42
43 # This function gets a complex endpoint that contains a few components
44 def getComplexEndpoint(endpoint):
45     response = fetch(endpoint)
46     if response == requestErrorDescription:
47         return requestErrorDescription
48     try:
49         json = response.json()
50         return json
51     except:
52         print(requestErrorDescription)
53
54 # Given columns, this method prints the column and value
55 def printJSON(columns):
56     for column in columns:
57         print(column + ": " + json[column])
58
59 # Start listening for new input
60 while inputKey != "q":
61     for iteration in range(0, iterations):
62         iterationStartDate = datetime.datetime.now().timestamp()
63         if inputKey == "b":
64             print(getBasicEndpoint("dateTime"))
65         if inputKey == "u":
66             print(getBasicEndpoint("upTime"))
67         if inputKey == "n":
68             print(getBasicEndpoint("memoryUsage"))
69         if inputKey == "r":
70             networkConnections = getComplexEndpoint("networkConnections")
71             printJSON(["receiveQueue", "sendQueue", "localAddress", "foreignAddress", "state"], networkConnections)
72         if inputKey == "c":
73             runningProcesses = getComplexEndpoint("runningProcesses")
74             printJSON(["pid", "cmdLine"], runningProcesses)
75         if inputKey == "t":
76             runningProcesses = getComplexEndpoint("currentUsers")
77             printJSON(["uid", "cmdLine"], runningProcesses)
78         if inputKey == "h":
79             runningProcesses = getComplexEndpoint("netstatConnections")
80             printJSON(["srcPort", "dstPort", "receiveQueue", "sendQueue"], runningProcesses)
81         if inputKey == "l":
82             printJSON(["pid"], runningProcesses)
83         if inputKey == "a":
84             averageTimes.append(datetime.datetime.now().timestamp() - iterationStartDate)
85
86     inputKey = input()
87
88     print(f"Request over time: {averageTimes}")
89     print(f"Average time per request: {mean(averageTimes)}")

```

The client is composed of modular architecture to ensure maintainability and readability throughout the programming process. For example, we created methods such as `getBasicEndpoint` and `getComplexEndpoint`. These methods were used to abstract certain reusable components such as retrieving a basic endpoint that returns a double or float, and then a more complex method that returns a json. Moreover, it utilizes other helper methods such as `fetch`, which wraps the `requests` library in python and handles exceptions by printing an error message that is defined as a constant at the top of the Python script.

Similarly, the server utilizes modularity and abstraction to write cleaner code. We've abstracted each endpoint into a helper method that executes the Linux command and parses it as needed so it's digestible on the client side. For example, '`network_connections`' utilizes subprocesses to run '`netstat`', which is then split into each line and extracted to be dumped in a json object. However there is one discrepancy in code uniformity with the use of camel case and snake case between the client and server, ideally we'd maintain the same coding conventions across the two scripts.

Overall, we decided to approach the code in this way to improve maintainability and readability which helped with the efficiency of development and pressing .

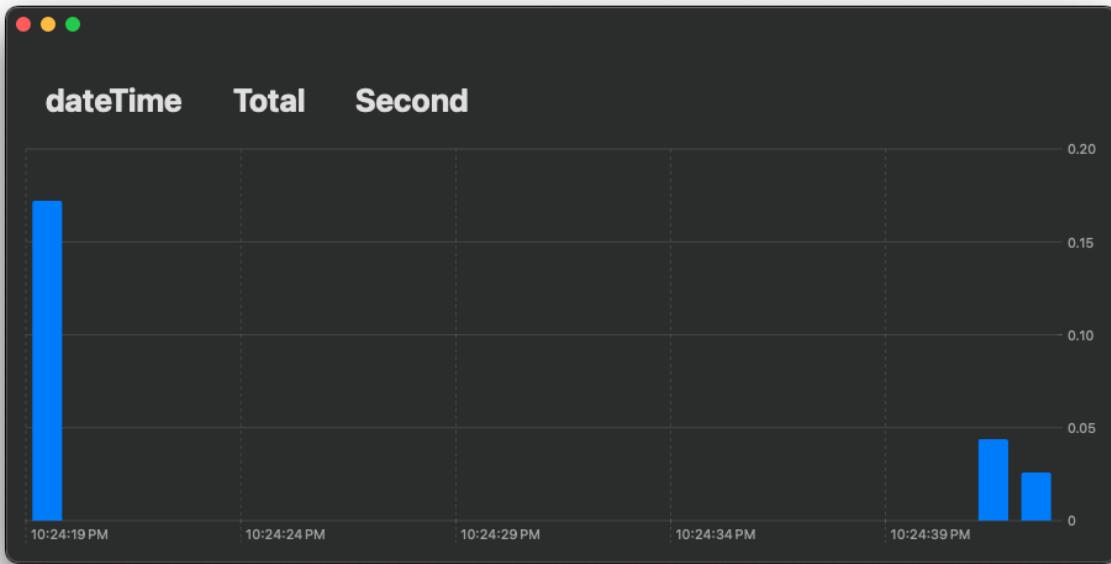
```

01 # Here are our imports, these allow us to use functions from other libraries
02 import os
03 import subprocess
04 import socketserver
05 import socketserver
06 import http.server
07 import socket
08
09 def date_time():
10     return json.dumps({"dateTime": time.time()})
11
12 def up_time():
13     up_time = subprocess.run(['uptime', '-s'], stdout=subprocess.PIPE)
14     up_time_output = up_time.stdout.decode('utf-8')
15     up_time_fields = up_time_output.split("\n")
16     return json.dumps({"upTime": time.time() - int(up_time_fields[1])})
17
18 def memory_usage():
19     return json.dumps({"memoryUsage": os.cpu_count()})
20
21 # Returns network connection
22 # TODO: Check for correct fields in the project instructions
23 def network_connections():
24
25     result = subprocess.run(['netstat'], stdout=subprocess.PIPE)
26     jsonToReturn = []
27
28     # We have to do some parsing to return a json
29     for line in result.stdout.decode("utf-8").split("\n"):
30         if len(line.split()) == 8:
31             proto, receiveQueue, sendQueue, localAddress, foreignAddress, state, = line.split()
32             jsonToReturn.append({
33                 "proto": proto,
34                 "receiveQueue": receiveQueue,
35                 "sendQueue": sendQueue,
36                 "localAddress": localAddress,
37                 "foreignAddress": foreignAddress,
38                 "state": state
39             })
40
41     return json.dumps(jsonToReturn)
42
43 # TODO: Check for correct fields in the project instructions
44 def current_users():
45
46     users = []
47
48     current_user = subprocess.run(['who'], stdout=subprocess.PIPE).stdout.decode("utf-8")
49
50     for user in current_user:
51         users.append({
52             "name": user.name,
53             "host": user.host
54         })
55
56     return json.dumps(users)
57
58 # TODO: Check for correct fields in the project instructions
59 def running_processes():
60
61     current_processes = subprocess.run(['ps'], stdout=subprocess.PIPE)
62
63     for process in current_processes:
64         processes.append(process)
65
66     return json.dumps(processes)
67
68
69 class ServerHandler(http.server.BaseHTTPRequestHandler):
70
71     # Calls when we do a get request
72     def do_GET(self):
73         path = self.path
74
75         if path == "/runningProcesses":
76             response = running_processes()
77             package_response(self, response)
78
79         elif path == "/currentUsers":
80             response = current_users()
81             package_response(self, response)
82
83         elif path == "/networkConnections":
84             response = network_connections()
85             package_response(self, response)
86
87         elif path == "/memoryUsage":
88             response = memory_usage()
89             package_response(self, response)
90
91         elif path == "/uptime":
92             response = up_time()
93             package_response(self, response)
94
95         elif path == "/dateTime":
96             response = date_time()
97             package_response(self, response)
98
99         else:
100             # Return a 404 response for routes unknown
101             self.send_response(404)
102             self.end_headers()
103             self.wfile.write("Not Found".encode())
104
105
106 # Starts our server
107 with socketserver.TCPServer(("0.0.0.0", 3213), ServerHandler) as httpd:
108     print("Server is running")
109     httpd.serve_forever()

```

## Testing and Data Collection

The final step of developing the iterative socket server was by running and testing these servers through trial and error. The project was tested by running two python programs, a server program and a client program on the UNF servers. We also developed a test case app to help us with our development and testing which we will discuss later on. These programs were run by connecting to the server through the use of a VPN and a remote-in app such as Bitvise or Terminus. Once these servers were connected through the UNF server, the code to compile the python programs was executed on both the client and the server. We developed two client servers, one for testing as shown below on the graph and data visualization images. The other client was used in the UNF server as part of the final project of the iterative socket server.



## Data Analysis

Increasing the number of clients does in fact lead to longer turnaround time for individual clients since the servers capacity is limited to support a fixed size of customers. However, effectively managing the requests with prioritization, and the proper allocation of resources can mitigate this problem. Additionally, clear communication with clients about the expected turnaround times and potential delays is essential for managing expectations and maintaining client relationships. Increasing the number of clients can greatly affect the average turnaround time due to the increased workload of a large client base. In order to fix this the same steps can be taken as above. The primary cause of the effect on individual client turn-around time and the average turnaround time is that they both are affected by the workload that is brought onto them by the client. Also the capacity of how many requests the server can hold is a huge factor in the increasing and decreasing of efficiency. Without the changing of capacity, the amount of clients will overwhelm the server with requests and cause it to slow its processing speed. Thus, these factors contribute to the increased average turnaround time and the individual turnaround time.



## Conclusion

Based on the data analysis and from our testing of the client/server, we can strongly conclude that server response time does in fact increase based on an influx of concurrent requests from the client server. We determined that the netstat command produced the slowest return time. Further, we found that the date time command was the fastest to return a response among the commands. We found these results based on multiple testing and data analysis performed by our group. Throughout the project we learned that modularity is important, especially in a group project with multiple teammates to isolate issues and resolve git conflicts and so that the process of the project can run more efficiently. Each team member was assigned their own portion of the project such as development of the server, development of the client, and the writing of the

paper. Ultimately we all pitched in with each other's work load or process of the project. Also making TODO lists which gave us tasks to conquer for the project each week, a github sharing station for the code, and a task list were helpful when working as a team. We originally wrote the client and server in a version incompatible with the UNF server which led us to drastically revamp the server and client in a new way. To resolve this next time, we'd test the imports on the server to ensure we are not using incompatible technology.

