Socket Programming Project 3

Sanuel Parkman

Oregon State University

June 2nd, 2024

Table of Contents

SRS – Hive Member Software Requirements Specification

How to run the app

Open 4 terminals and execute one of the following commands in each

python .\app_main.py -ip 127.0.0.1 -port 54321 -friendly_name LosAngeles -main Yes

python .\app_main.py -ip 127.0.0.1 -port 54322 -friendly_name London

python .\app_main.py -ip 127.0.0.1 -port 54323 -friendly_name Brisbane

python .\app_main.py -ip 127.0.0.1 -port 54324 -friendly_name NewYork

Once the app starts, type the following from London, Brisbane, and NewYork:

connect 127.0.0.1 54321

SRS 1

This requirement was to maintain all existing hive functionality. The initial hive functionality was connections, heartbeat and gossip messages and I made sure to not change any of them. Below is a screenshot of the terminal window during a run and as you can see all three of the features mentioned earlier are seen there(figure 1. If no connection is made to another node then no information will be passed and the node remains unaware of any other nodes that could be running at the same time just as it was in the original code.

Figure 1

Shows initial features are still present in final code

```
PROBLEMS 204   OUTPUT   DEBUG CONSOLE   TERMINAL

[2024-06-01 21:13:13][HeartbeatProtocolCommandManager][INFO   ] Sending heartbeat to NewYork...
[2024-06-01 21:13:23][HeartbeatProtocolCommandManager][INFO   ] Sending heartbeat to NewYork...
[2024-06-01 21:13:27][HiveReceiverService            ][INFO   ] Connection from ('127.0.0.1', 51257)
[2024-06-01 21:13:27][HiveNodeManager                ][INFO   ] Node NewYork already exists in the node list...
[2024-06-01 21:13:27][HiveReceiverService            ][INFO   ] Handled gossip from NewYork
[2024-06-01 21:13:32][HiveReceiverService            ][INFO   ] Connection from ('127.0.0.1', 51259)
[2024-06-01 21:13:32][HiveReceiverService            ][INFO   ] Handled heartbeat from NewYork
[2024-06-01 21:13:33][HeartbeatProtocolCommandManager][INFO   ] Sending heartbeat to NewYork...
[2024-06-01 21:13:33][InboundQueueCommandProcessor   ][INFO   ] Received heartbeat from NewYork...
[2024-06-01 21:13:33][InboundQueueCommandProcessor   ][INFO   ] Found NewYork in the node list...
[2024-06-01 21:13:33][InboundQueueCommandProcessor   ][INFO   ] Updated last heartbeat for NewYork...
[2024-06-01 21:13:36][Service Monitoring             ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-01 21:13:37][HiveReceiverService            ][INFO   ] Connection from ('127.0.0.1', 51262)
[2024-06-01 21:13:37][HiveNodeManager                ][INFO   ] Node NewYork already exists in the node list...
[2024-06-01 21:13:37][HiveReceiverService            ][INFO   ] Handled gossip from NewYork
[2024-06-01 21:13:42][HiveReceiverService            ][INFO   ] Connection from ('127.0.0.1', 51264)
[2024-06-01 21:13:42][HiveReceiverService            ][INFO   ] Handled heartbeat from NewYork
[2024-06-01 21:13:43][HeartbeatProtocolCommandManager][INFO   ] Sending heartbeat to NewYork...
[2024-06-01 21:13:43][InboundQueueCommandProcessor   ][INFO   ] Received heartbeat from NewYork...
[2024-06-01 21:13:43][InboundQueueCommandProcessor   ][INFO   ] Found NewYork in the node list...
[2024-06-01 21:13:43][InboundQueueCommandProcessor   ][INFO   ] Updated last heartbeat for NewYork...
[2024-06-01 21:13:47][HiveReceiverService            ][INFO   ] Connection from ('127.0.0.1', 51266)
[2024-06-01 21:13:47][HiveNodeManager                ][INFO   ] Node NewYork already exists in the node list...
[2024-06-01 21:13:47][HiveReceiverService            ][INFO   ] Handled gossip from NewYork
[2024-06-01 21:13:52][HiveReceiverService            ][INFO   ] Connection from ('127.0.0.1', 51268)
[2024-06-01 21:13:52][HiveReceiverService            ][INFO   ] Handled heartbeat from NewYork
[2024-06-01 21:13:53][HeartbeatProtocolCommandManager][INFO   ] Sending heartbeat to NewYork...
[2024-06-01 21:13:53][InboundQueueCommandProcessor   ][INFO   ] Received heartbeat from NewYork...
[2024-06-01 21:13:53][InboundQueueCommandProcessor   ][INFO   ] Found NewYork in the node list...
[2024-06-01 21:13:53][InboundQueueCommandProcessor   ][INFO   ] Updated last heartbeat for NewYork...
```

SRS 2

This requirement states that each hive node must have the complete configuration for service monitoring for the whole hive. This was implemented by adding the parameter “main” to the way a hive node is started. By making this parameter true it tells the hive node to grab the initial configuration. If its set to false then does when the node is created. That is until it connects to a node that already has the configuration. Once the connection request is sent and the other hive node receives it, Within the inbound queue it creates a HiveMessage with the current configuration it has and sends it back(figure 2). The node that sent the connection request then receives the configuration and updates its currently empty configuration with the new one (figure 3).

Figure 2

On connect request node sends current configuration back

```python
    def process_command_connect(self, hive_message: HiveMessage) -> None:
        """
        Processes a 'connect' command, adding the new node to the node manager.

        Parameters:
        ----------
        hive_message : HiveMessage
            The message containing the 'connect' command.
        """
        self.logger.info("InboundQueueCommandProcessor", f"Received connection request from {hive_message.message.sender.friendly_name}...")
        new_hive_node = HiveNode(hive_message.message.sender.friendly_name, hive_message.message.sender.ip_address, int(hive_message.message.sender.port_number))
        self.hive_node_manager.add_node(new_hive_node)
        self.logger.info("InboundQueueCommandProcessor", f"Added {new_hive_node.friendly_name} to the node list...")

        # Send current monitoring config back to requesting node
        config_message: ConfigMessage = ConfigMessage(
            sender=self.hive_node_manager.local_node,
            recipient=new_hive_node,
            message=str(self.app_main.configuration_dict)
        )
        new_hive_message: HiveMessage = HiveMessage(config_message)
        self.outbound_message_queue.enqueue(new_hive_message)
```

Figure 3

Connecting node receives new configuration and updates its current copy to the new one.

```python
    def process_command_config(self, hive_message: HiveMessage):
        parsed_dict = self.str_to_dict(hive_message.message.message)
        services = self.str_to_dict(parsed_dict['message'])


        # No previous configurations
        if  self.app_main.configuration_dict == {}:
            self.app_main.configuration_dict = services

        else:
            current_version = self.app_main.configuration_dict["config"]["version"]
            new_version = services["config"]["version"]

            if current_version < new_version:
                self.app_main.configuration_dict = services

            elif current_version == new_version:

                timestamp1 = self.app_main.configuration_dict["config"]["datetime"]
                timestamp2 = services["config"]["datetime"]

                if timestamp1 < timestamp2:
                    self.app_main.configuration_dict = services



        self.logger.info('InboundQueueCommandProcessor', f' Updating Configuration to: {self.app_main.configuration_dict}')
```

SRS 3

This requirement is requires that network service monitoring configuration will contain a node name that references a list of services for that node to monitor. This was accomplished by using a dictionary as seen in figure 4. This is the initial configuration found in the file configuration. Json. Each nodes friendly name has an entry in the dictionary and each values is another dictionary of the services and their parameters.

Figure 4

Initial JSON configuration for service monitoring services

```
1  {
2      "config": {
3        "version": 1,
4        "datetime": "Initial",
5        "Services": {
6          "LosAngeles": {
7            "DNS": {
8              "target": "108.170.228.102",
9              "queries":
10               {
11                 "domain": "mlssoccer.com",
12                 "type": "A"
13               },
14             "interval": 250
15           },
16           "UDP": {
17             "target": "209.18.36.55",
18             "port": 443,
19             "interval": 80
20           },
21           "NTP": {
22             "target": "a",
23             "interval": 120
24           },
25           "HTTPS": {
26             "target": "wsu.edu",
27             "port": 80,
28             "interval": 40
29           }
30         },
31         "NewYork": {
32           "ICMP": {
33             "target": "13.107.42.14",
34             "interval": 60
35           },
36           "NTP": {
37             "target": "time3.google.com",
38             "interval": 120
39           },
40           "HTTP": {
41             "target": "34.223.124.45",
42             "port": 80,
43             "interval": 2
44           },
45           "TCP": {
46             "target": "23.36.9.243",
47             "port": 443,
48             "interval": 60
49           }
50         },
```

SRS 4

This requirement states that upon receiving a new configuration the node will update the current monitoring checks to match the new configuration. The functionality to add (figure 5), edit(figure 6), and delete (figure 7) were added as command line inputs. Once the user does one of these three actions the updated dictionary is then sent to the function send_updated_config (figure 8) where its version and timestamp are updated. This function checks to see all the lives nodes its aware of and sends the new configuration to each of the nodes. The nodes that receive the updated configuration compare the version number and timestamp to the one they currently have. If its newer then they update their configuration to the latest (figure 9). While this is happening in the ServiceMontiorong.run function is checking every 10 seconds to see if any updates to the dictionary have happened (figure 10). If there is an update then it passes True in for the parameter update in the thread handling function (figure 11). There it stops the current threads by setting the stop event and then starting the new updated config file monitoring services.

Figure 5

Function to add a service

```python
    def add_service(self, parts):
        """add cityName ServiceName Target port interval"""
        try:
            config = self.app_main.configuration_dict
            city = parts[1]
            service_type = parts[2].upper()
            service_entry = {"target": parts[3].upper()}

            if service_type in ["HTTP", "HTTPS", "TCP", "UDP"]:
                service_entry["port"] = int(parts[4])
                service_entry["interval"] = int(parts[5])
            elif service_type == "DNS":
                service_entry["queries"] = {}
                service_entry["queries"]["domain"] = parts[4].lower()
                service_entry["queries"]["type"] = parts[5].lower()
                service_entry["interval"] = int(parts[6])

            elif service_type in ["NTP", "ICMP"]:
                service_entry["interval"] = int(parts[4])

            # Update Entry and version
            timestamp = datetime.datetime.now().strftime(AppSettings.TIMESTAMP_FORMAT)
            config["config"]["Services"][city][service_type] = service_entry
            config["config"]["version"] += 1
            config["config"]["datetime"] = timestamp

            #Send updated config to other nodes
            self.send_updated_config(config)
        except Exception as e:
             self.logger.info("CliCommandProcessor", f"Error due to {e}")
```

Figure 6

Function to add to edit a service

```python
    def edit_service(self, parts):
        """edit cityName ServiceName Target editParts"""
        edit_dict = {}
        try:
            # Finds out which elements are to be edited
            for index, edit in enumerate(parts):
                if '=' in edit:
                    split_edits = parts[index].split('=')
                    edit_dict[split_edits[0].lower()] = split_edits[1].lower()

            if edit_dict:
                config = self.app_main.configuration_dict
                city = parts[1]

                service_type = parts[2].upper()

                target = parts[3]

                config_current = config['config']['Services'][city]

                for service in config_current:
                    if config_current[service]['target'] == target:


                        for edits, value in edit_dict.items():
                            if edits in ["domain", "type"]:
                                del config_current[service_type]["queries"][edits]
                                config_current[service_type]["queries"][edits] = value

                            else:
                                del config_current[service_type][edits]
                                config_current[service_type][edits] = value


                #Updates entry in configurations
                self.logger.info("CliCommandProcessor", f"Edited config {config}")

                #Send updated config to other nodes
                self.send_updated_config(config)

            else:
                self.logger.info("CliCommandProcessor", "No edits entered")

        except Exception as e:
            self.logger.info("CliCommandProcessor", f"Error due to {e}")

```

Figure 7

Function to delete a service

```python
    def delete_service(self, parts):
        """delete cityName ServiceName Target"""
        city = parts[1]
        service = parts[2].upper()
        target = parts[3]
        config = self.app_main.configuration_dict
        try:
            if config["config"]["Services"][city][service]["target"] == target:
                del config["config"]["Services"][city][service]
            else:
                self.logger.info("CliCommandProcessor", f"No such entry exist for {city}")
        except Exception as e:
            self.logger.info("CliCommandProcessor", f"Error invalid input{e}")

        # Sends updated configuration to other hive nodes
        self.send_updated_config(config)
```

Figure 8

Function to send updated configuration to all known nodes

```python
    def send_updated_config(self, config: Dict):
        """Sends updated configuration to all current lives nodes"""

        # Update Entry and version
        timestamp = datetime.datetime.now().strftime(AppSettings.TIMESTAMP_FORMAT)
        config["config"]["version"] += 1
        config["config"]["datetime"] = timestamp

        updated_configuration = json.dumps(config)
        updated_configuration = updated_configuration.replace("'", '"')

        for random_node in self.hive_node_manager.get_all_live_nodes():
            if random_node.friendly_name != self.app_main.name:
                Config_message: ConfigMessage = ConfigMessage(
                            sender=self.hive_node_manager.local_node,
                            recipient=random_node,
                            message=updated_configuration,
                            )
                new_hive_message: HiveMessage = HiveMessage(Config_message)
                self.outbound_message_queue.enqueue(new_hive_message)

                # edit NewYork NTP time3.google.com  target=time5.google.com interval=60
```

Figure 9

Function compares new configuration and keeps the latest copy

```python
    def process_command_config(self, hive_message: HiveMessage):
        parsed_dict = self.str_to_dict(hive_message.message.message)
        services = self.str_to_dict(parsed_dict['message'])


        # No previous configurations
        if  self.app_main.configuration_dict == {}:
            self.app_main.configuration_dict = services

        else:
            current_version = self.app_main.configuration_dict["config"]["version"]
            new_version = services["config"]["version"]

            if current_version < new_version:
                self.app_main.configuration_dict = services

            elif current_version == new_version:

                timestamp1 = self.app_main.configuration_dict["config"]["datetime"]
                timestamp2 = services["config"]["datetime"]

                if timestamp1 < timestamp2:
                    self.app_main.configuration_dict = services
```

Figure 10

Function within ServiceMonitoirng class that checks to see if there is an update to the configuration

```python
    def run(self) -> None:
        """
        Starts the HiveReceiverService, listening for incoming connections and handling them in separate threads.
        """
        global current_config
        while True:

            if self.appMain.configuration_dict == {}:
                self.logger.info("Service Monitoring", "Waiting for configuration file")
            else:

                if current_config is None:
                    print('First time through with actual dictionary')
                    self.create_results_dict()
                    current_config = self.appMain.configuration_dict
                    self.thread_handling(self.appMain.configuration_dict)

                # Only updates service monitors if this cities entries were updated
                elif current_config["config"]["Services"][self.appMain.name] != self.appMain.configuration_dict["config"]["Services"][self.appMain.name]:
                    print('Update happened to config and need to update')
                    self.create_results_dict()
                    current_config = self.appMain.configuration_dict
                    self.thread_handling(self.appMain.configuration_dict, True)

            time.sleep(10)
```

Figure 11

If update is True then all current checks are stopped and the new set are started

```python
    def thread_handling(self, config_dict, update=False) -> None:
        """
        Main function to handle user input and manage threads.
        Uses prompt-toolkit for handling user input with auto-completion and ensures
        the prompt stays at the bottom of the terminal.
        """
        if update:
            self.logger.info("Service Monitoring", "Updated services")
            for event in stop_events:
                event.set()

        city_services = config_dict["config"]["Services"][self.appMain.name]
        # Event to signal the worker thread to stop
        stop_event: threading.Event = threading.Event()
        stop_events.append(stop_event)

        # Create and start the worker thread
        thread = None
        for service in city_services:
            if service == 'ECHO':
                thread: threading.Thread = threading.Thread(target=self.check_echo_server, args=(stop_event, city_services[service]))

            elif service  == 'HTTP' or service  == 'HTTPS' :
                thread: threading.Thread = threading.Thread(target=self.check_http_request, args=(stop_event, city_services[service], service))

            elif service  == "ICMP":
                thread: threading.Thread = threading.Thread(target=self.check_icmp, args=(stop_event, city_services[service]))

            elif service  == "DNS":
                thread: threading.Thread = threading.Thread(target=self.check_dns, args=(stop_event,city_services[service]))

            elif service  == "TCP":
                thread: threading.Thread = threading.Thread(target=self.check_tcp, args=(stop_event, city_services[service]))

            elif service  == 'UDP':
                thread: threading.Thread = threading.Thread(target=self.check_udp, args=(stop_event, city_services[service]))

            elif service  == 'NTP':
                thread: threading.Thread = threading.Thread(target=self.check_ntp, args=(stop_event, city_services[service]))
            thread.start()
```

SRS 5

This requirement states that the service monitoring checks must be printed to the terminal as well as logged in a file. The starting hive functionality already included the ability to log information in a file name app.friendlyName.log as well as print it to the terminal. So the functions self.logger.info() were used to maintain this functionality. Figure 12 below shows the service monitoring checks were printed to the terminal and figure 13 shows them in the log file.

In both cases the blue and red colored boxes show the same monitoring check in the log and the terminal to show it being present in both cases.

Figure 12

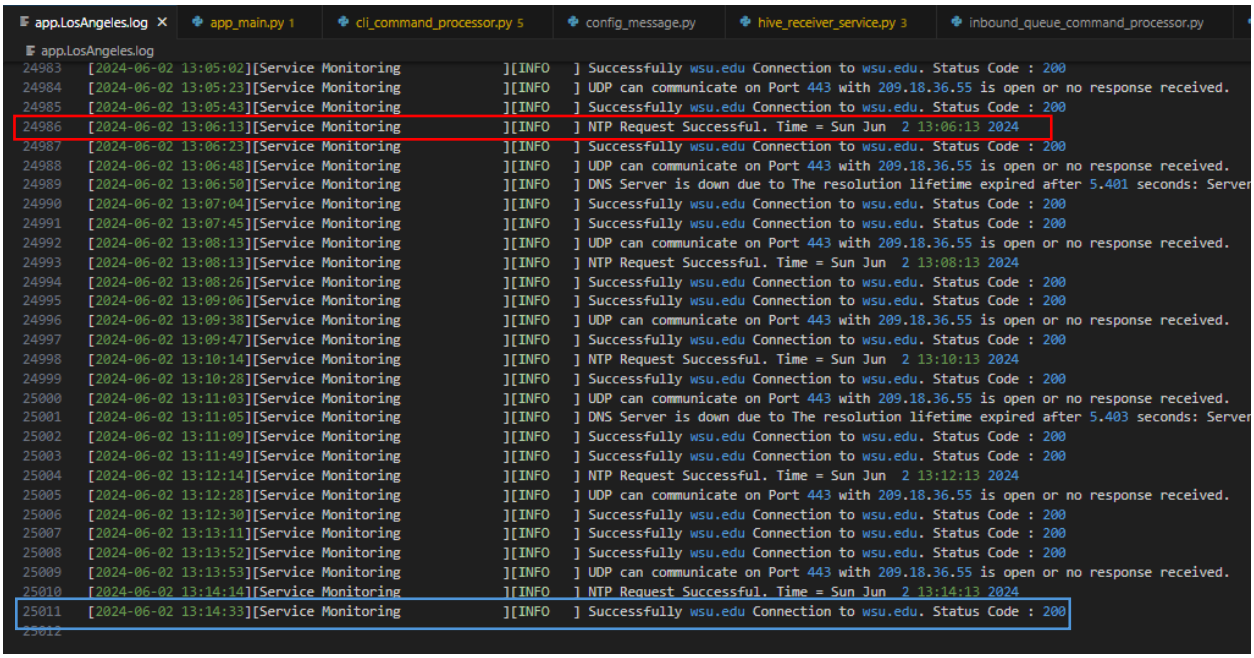
Monitor checks are printed to the terminal window

```
PROBLEMS 199   OUTPUT   DEBUG CONSOLE   TERMINAL

[2024-06-02 13:05:02][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:05:23][Service Monitoring            ][INFO   ] UDP can communicate on Port 443 with 209.18.36.55 is open or no response received.
[2024-06-02 13:05:43][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:06:13][Service Monitoring            ][INFO   ] NTP Request Successful. Time = Sun Jun  2 13:06:13 2024
[2024-06-02 13:06:23][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:06:48][Service Monitoring            ][INFO   ] UDP can communicate on Port 443 with 209.18.36.55 is open or no response received.
[2024-06-02 13:06:50][Service Monitoring            ][INFO   ] DNS Server is down due to The resolution lifetime expired after 5.401 seconds: Server
ered The DNS operation timed out.; Server Do53:108.170.228.102@53 answered The DNS operation timed out.
[2024-06-02 13:07:04][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:07:45][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:08:13][Service Monitoring            ][INFO   ] UDP can communicate on Port 443 with 209.18.36.55 is open or no response received.
[2024-06-02 13:08:13][Service Monitoring            ][INFO   ] NTP Request Successful. Time = Sun Jun  2 13:08:13 2024
[2024-06-02 13:08:26][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:09:06][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:09:38][Service Monitoring            ][INFO   ] UDP can communicate on Port 443 with 209.18.36.55 is open or no response received.
[2024-06-02 13:09:47][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:10:14][Service Monitoring            ][INFO   ] NTP Request Successful. Time = Sun Jun  2 13:10:13 2024
[2024-06-02 13:10:28][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:11:03][Service Monitoring            ][INFO   ] UDP can communicate on Port 443 with 209.18.36.55 is open or no response received.
[2024-06-02 13:11:05][Service Monitoring            ][INFO   ] DNS Server is down due to The resolution lifetime expired after 5.403 seconds: Server
ered The DNS operation timed out.; Server Do53:108.170.228.102@53 answered The DNS operation timed out.
[2024-06-02 13:11:09][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:11:49][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:12:14][Service Monitoring            ][INFO   ] NTP Request Successful. Time = Sun Jun  2 13:12:13 2024
[2024-06-02 13:12:28][Service Monitoring            ][INFO   ] UDP can communicate on Port 443 with 209.18.36.55 is open or no response received.
[2024-06-02 13:12:30][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:13:11][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:13:52][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 13:13:53][Service Monitoring            ][INFO   ] UDP can communicate on Port 443 with 209.18.36.55 is open or no response received.
[2024-06-02 13:14:14][Service Monitoring            ][INFO   ] NTP Request Successful. Time = Sun Jun  2 13:14:13 2024
[2024-06-02 13:14:33][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
LosAngeles> config_print
Node      | Service    | Parameters
PS C:\Users\swpar\Desktop\OSU\CS372\Hive.v01>
```

Figure 13

Monitor checks are logged in the appropriate log file



SRS 6

This requirement was to display a tabular view of the current network monitoring configuration for the entire hive. This is achieved by typing “config_print” into the command line as seen in figure 14. It not only shows the services for each but all the parameters associated with each of them.

Figure 14

All hive network monitoring configuration printed to the terminal window

```
PROBLEMS 199   OUTPUT   DEBUG CONSOLE   TERMINAL

LosAngeles> config_print
Node        | Service    | Parameters
-------------------------------------------------------------------------------------------------------------------------
LosAngeles      | DNS      | {'target': '108.170.228.102', 'queries': {'domain': 'mlssoccer.com', 'type': 'A'}, 'interval': 250}
LosAngeles      | UDP      | {'target': '209.18.36.55', 'port': 443, 'interval': 80}
LosAngeles      | NTP      | {'target': 'time3.google.com', 'interval': 120}
LosAngeles      | HTTPS      | {'target': 'wsu.edu', 'port': 80, 'interval': 40}
NewYork      | ICMP      | {'target': '13.107.42.14', 'interval': 60}
NewYork      | NTP      | {'target': 'time3.google.com', 'interval': 120}
NewYork      | HTTP      | {'target': '34.223.124.45', 'port': 80, 'interval': 200}
NewYork      | TCP      | {'target': '23.36.9.243', 'port': 443, 'interval': 60}
```

Figure 15

Command line code for when config_print is entered

```
    def config_print(self):
        """Prints configuration"""
        print("Node      | Service    | Parameters")
        print("-------------------------------------------------------------------------------------------------------------------------")
        for hive_member in self.app_main.configuration_dict["config"]["Services"]:
            for key in self.app_main.configuration_dict["config"]["Services"][hive_member]:
                print(f"{hive_member}      | {key}      | {self.app_main.configuration_dict["config"]["Services"][hive_member][key]}")
```

SRS 7

This requirement was to display a tabular list of the local nodes monitoring status for each service check it was assigned. This can be seen by typing into the command line “status”. This will print each monitoring service, the latest result and the timestamp of when it was taken (figure 16). The code can be seen in figure 17.

Figure 16

Only local monitoring checks status are printed to the terminal

```
PROBLEMS 204   OUTPUT   DEBUG CONSOLE   TERMINAL

0}, 'NTP': {'target': 'time3.google.com', 'interval': 120}, 'HTTPS': {'target': 'wsu.edu', 'port': 80, 'interval': 40}}, 'NewYork': {'
 {'target': '34.223.124.45', 'port': 80, 'interval': 200}, 'TCP': {'target': '23.36.9.243', 'port': 443, 'interval': 60}}, 'Brisbane':
[2024-06-02 14:13:39][Service Monitoring            ][INFO   ] NTP Request Successful. Time = Sun Jun  2 14:13:39 2024
[2024-06-02 14:13:40][Service Monitoring            ][INFO   ] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
[2024-06-02 14:13:44][Service Monitoring            ][INFO   ] UDP can communicate on Port 443 with 209.18.36.55 is open or no respons
[2024-06-02 14:13:44][Service Monitoring            ][INFO   ] DNS Server is down due to The resolution lifetime expired after 5.405 s
ered The DNS operation timed out.; Server Do53:108.170.228.102@53 answered The DNS operation timed out.
LosAngeles> status
Node      | Service    | Current Status
-------------------------------------------------------------------------------------------------------------------
LosAngeles      | DNS      | [2024-06-02 14:13:39] DNS Server connection Unsuccessful
LosAngeles      | UDP      | [2024-06-02 14:13:39] UDP can communicate on Port 443 with 209.18.36.55 is open or no response received.
LosAngeles      | NTP      | [2024-06-02 14:13:39] NTP Request Successful. Time = Sun Jun  2 14:13:39 2024
LosAngeles      | HTTPS      | [2024-06-02 14:13:39] Successfully wsu.edu Connection to wsu.edu. Status Code : 200
LosAngeles> exit
PS C:\Users\swpar\Desktop\OSU\CS372\Hive.v01>
```

Figure 17

Command line code for when Status is entered

```
    def status(self):
        """Prints configuration"""
        print("Node      | Service    | Current Status")
        print("-------------------------------------------------------------------------------------------------------------------")
        for service in self.app_main.results_dict[self.app_main.name]:
            print(f"{self.app_main.name}      | {service}      | {self.app_main.results_dict[self.app_main.name][service]}")
```