ABDELRAHMAN ABBAS
30110374
Issam Akhtar
30131310

CPSC 526
ASSIGNMENT 4

Task 1: Ncat-based Bot

```
def connect_loop(host, port, nick, secret):
    seen_nonces = set()
    command_count = 0

    while True:
        s = None
        try:
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.settimeout(5)
            s.connect((host, port))
            s.settimeout(None)

            print("Connected.")
            join_msg = f"-joined {nick}\n"
            s.sendall(join_msg.encode('utf-8'))

            command_count, should_exit = handle_commands(
                s, nick, secret, seen_nonces, command_count
            )

            if should_exit:
                return

            print("Disconnected.")

        except Exception as e:
            print("Failed to connect.")
            if s:
                s.close()
            time.sleep(5)  # Wait before retrying
```

This function manages the bot's connection to the ncat server. It continuously attempts to connect, sends a join message upon successful connection, and handles incoming commands. If the connection is lost, it retries every 5 seconds.

```
def handle_commands(sock, nick, secret, seen_nonces, command_count):
    while True:
        line = sock.recv(4096)
        if not line:
            return command_count, False

        lines = line.decode('utf-8', errors='ignore').split('\n')
```

```
        for cmdline in lines:
            cmdline = cmdline.strip()
            if not cmdline:
                continue
            parts = cmdline.split()
            if len(parts) < 3:
                continue

            nonce, mac, command = parts[0], parts[1], parts[2]
            args = parts[3:]

            # Authentication
            if nonce in seen_nonces:
                continue
            computed = compute_mac(nonce, secret)
            if computed != mac:
                continue
            seen_nonces.add(nonce)
            command_count += 1

            # Command execution
            if command == "status":
                msg = f"-status {nick} {command_count}\n"
                sock.sendall(msg.encode('utf-8'))
            elif command == "shutdown":
                msg = f"-shutdown {nick}\n"
                sock.sendall(msg.encode('utf-8'))
                return command_count, True
            elif command == "attack":
                target = args[0]
                attack_status = do_attack(target, nick, nonce)
                sock.sendall(attack_status.encode('utf-8'))
            elif command == "move":
                move_target = args[0]
                move_msg = f"-move {nick}\n"
                sock.sendall(move_msg.encode('utf-8'))
                sock.close()
                reconnect_to_new(
                    move_target.split(":")[0],
                    int(move_target.split(":")[1]),
                    nick, secret, seen_nonces, command_count
                )
                return command_count, False
```

This function handles incoming commands from the server. It authenticates each command using a nonce and MAC, then executes the appropriate action (status, shutdown, attack, or move) and sends the response back to the server. This structure will be used throughout the rest of the implementation.

```
def do_attack(target, nick, nonce):
```

```
    """
    Attempt to connect to <hostname>:<port> and send a line:
       <nick> <nonce>
    Then close the connection.
    Return a message string of the form:
       "-attack <nick> OK\n" or
       "-attack <nick> FAIL <error-reason>\n"
    """
    try:
        host, port_str = target.split(":")
        port = int(port_str)
    except:
        return f"-attack {nick} FAIL invalid-target\n"

    # Attempt connection with 3s timeout
    try:
        with socket.create_connection((host, port), timeout=3) as s:
            attack_line = f"{nick} {nonce}\n"
            s.sendall(attack_line.encode('utf-8'))
        # If we reach here, we were successful
        return f"-attack {nick} OK\n"
    except socket.timeout:
        return f"-attack {nick} FAIL timeout\n"
    except Exception as e:
        # Could be ConnectionRefusedError, socket.gaierror, etc.
        reason = str(e).lower().replace(" ", "_")
        return f"-attack {nick} FAIL {reason}\n"
```

The function begins by attempting to parse the target into separate hostname and port components. If this parsing fails it immediately returns a failure message indicating an invalid target. The function then attempts to establish a TCP connection to the specified host and port with a timeout of 3 seconds, ensuring the bot doesn't become stuck waiting indefinitely if the target doesn't respond. If the connection is successful, the bot sends a specially formatted message containing its nickname and the nonce, then closes the connection as per the assignment specification. The return value is a status message formatted as either "-attack <nick> OK\n" if successful, or "-attack <nick> FAIL <error-reason>\n" if unsuccessful, where <error-reason> provides specific details about what went wrong.

Task 2: Ncat-based Controller

```
def main():
    while True:
        cmdline = input("cmd> ").strip()
        if not cmdline:
            continue

        parts = cmdline.split()
```

```
        command = parts[0].lower()

        if command == "quit":
            s.close()
            sys.exit(0)

        elif command == "status":
            nonce = generate_nonce()
            mac = compute_mac(nonce, secret)
            msg = f"{nonce} {mac} status\n"
            s.sendall(msg.encode('utf-8'))
            lines = collect_responses(s, timeout=5)
            st = parse_status_responses(lines)
            # ... formatting and output ...

        elif command == "shutdown":
            nonce = generate_nonce()
            mac = compute_mac(nonce, secret)
            msg = f"{nonce} {mac} shutdown\n"
            s.sendall(msg.encode('utf-8'))
            lines = collect_responses(s, timeout=5)
            nicks = parse_shutdown_responses(lines)
            # ... formatting and output ...

        elif command == "attack":
            target = parts[1]
            nonce = generate_nonce()
            mac = compute_mac(nonce, secret)
            msg = f"{nonce} {mac} attack {target}\n"
            s.sendall(msg.encode('utf-8'))
            lines = collect_responses(s, timeout=5)
            succ, fail = parse_attack_responses(lines)
            # ... formatting and output ...

        elif command == "move":
            new_target = parts[1]
            nonce = generate_nonce()
            mac = compute_mac(nonce, secret)
            msg = f"{nonce} {mac} move {new_target}\n"
            s.sendall(msg.encode('utf-8'))
            lines = collect_responses(s, timeout=5)
            mvnicks = parse_move_responses(lines)
            # ... formatting and output ...
```

The main function implements the command loop for the botnet controller. It operates within an infinite loop, continuously prompting the user for input commands. For each command entered, it splits the input command into parts and identifies the command type and any arguments provided. We have support for the following commands:

quit: This command closes the network connection and exits the program, terminating the controller.

status: Generates a unique nonce and corresponding MAC (Message Authentication Code) using a secret key, then sends a status inquiry command to all bots in the network. After sending the command, it collects responses from bots within a 5-second window and parses these responses to extract status information.

shutdown: Similar to status, this command generates a nonce and MAC, then sends a shutdown command to all bots. It collects and parses responses to determine which bots have successfully shut down.

attack: Accepts a target specification, generates authentication credentials, and sends an attack command directing bots to perform a network attack on the specified target. It then collects and parses responses to determine the success or failure of each bot's attack attempt.

move: Takes a new target location, generates authentication credentials, and sends a move command instructing bots to disconnect from their current server and connect to a new one. It collects and parses responses to identify which bots successfully moved to the new location.

Task 3: IRC-based Bot (ircbot.py)

```
class IRCBot:
    def __init__(self, host: str, port: int, channel: str, secret: str):
        self.host = host
        self.port = port
        self.channel = channel
        self.secret = secret
        self.nick = generate_random_nick()
        self.seen_nonces = set()
        self.command_count = 0
        self.connected = False
        self.socket = None

def generate_random_nick() -> str:
    prefix = "bot_"
    suffix = ''.join(random.choices(string.ascii_lowercase + string.digits,
k=6))
    return prefix + suffix
```

Each IRC bot generates a unique nickname when it connects to the IRC server. The nickname follows the format "bot_" followed by a random 6-character string.

```
def connect(self) -> bool:
    try:
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.connect((self.host, self.port))

        self.send(f"NICK {self.nick}")
        self.send(f"USER {self.nick} 0 * :{self.nick}")

        # ... handle server welcome and join channel ...

        return True
    except Exception as e:
        print(f"Failed to connect: {e}")
        return False
```

The IRC bot connects to an IRC server, registers with its generated nickname, and joins the specified channel. All communication happens within this channel. This is implementing the ‘logging into an IRC server’’ as described in the resources provided in the assignment.

```
def handle_command(self, message: str) -> None:
    parts = message.split()
    if len(parts) < 3:
        return

    nonce, mac, command = parts[0], parts[1], parts[2]
    args = parts[3:]

    if nonce in self.seen_nonces:
        return

    computed_mac = compute_mac(nonce, self.secret)
    if computed_mac != mac:
        return

    self.seen_nonces.add(nonce)
    self.command_count += 1
```

```
    if command == "status":
        self._handle_status()
    elif command == "shutdown":
        self._handle_shutdown()
    elif command == "attack" and args:
        self._handle_attack(args[0], nonce)
    elif command == "move" and args:
        self._handle_move(args[0])
```

This method processes commands received via IRC messages, authenticates them by seeing if the nonce has been used before, and executes the appropriate action with simple control logic. The bot responds to commands by sending messages back to the channel that include its nickname for identification.

Task 4: IRC-based Controller (irccontroller.py)

```
class IRCController:
    def __init__(self, host: str, port: int, channel: str, secret: str):
        self.host = host
        self.port = port
        self.channel = channel
        self.secret = secret
        self.nick = f"ctrl_{random.randint(1000, 9999)}"
        self.socket = None
        self.connected = False
        self.responses = []
        self.recv_buffer = ""
```

For this implementation we defined a controller class, similar to how we have our bot class. We name the control class nickname by giving it a random integer from 1000 to 9999, prepended by the string ‘ctrl_’

```
def connect(self) -> bool:
        try:
            self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.socket.connect((self.host, self.port))

            self.send(f"NICK {self.nick}")
```

```
            self.send(f"USER {self.nick} 0 * :Bot Controller")

            waiting_for_welcome = True
            while waiting_for_welcome:
                data = self.socket.recv(4096).decode('utf-8',
errors='ignore')
                if not data:
                    return False

                lines = data.split('\n')
                for line in lines:
                    line = line.strip()
                    if not line:
                        continue

                    if line.startswith("PING"):
                        pong = line.replace("PING", "PONG", 1)
                        self.send(pong)

                    if " 001 " in line:
                        self.send(f"JOIN #{self.channel}")
                        self.connected = True
                        waiting_for_welcome = False
                        break

            print(f"Connected to {self.host}:{self.port} and joined
#{self.channel}.")
            return True

        except Exception as e:
            print(f"Failed to connect: {e}")
            if self.socket:
                self.socket.close()
                self.socket = None
            self.connected = False
            return False
```

The IRC controller logs in to an IRC server with the double send, registers with a generated nickname (format "ctrl_XXXX"), and joins the specified channel after receiving the welcome message. This establishes a persistent connection through which commands can be sent to the bots.

```
def send_message(self, message: str) -> None:
    self.send(f"PRIVMSG #{self.channel} :{message}")
```

The controller sends commands to the channel using the IRC PRIVMSG command. This function constructs messages that are sent to all members of the channel, including the bots.

```
def run(self) -> None:
    if not self.connect():
        return

    while self.connected:
        try:
            print("cmd> ", end='', flush=True)
            cmd = input().strip()

            if not cmd:
                continue

            parts = cmd.split()
            command = parts[0].lower()

            if command == "quit":
                print("Disconnected.")
                if self.socket:
                    self.socket.close()
                break

            elif command in ["status", "shutdown", "attack", "move"]:
                nonce = generate_nonce()
                mac = compute_mac(nonce, self.secret)

                if command in ["attack", "move"]:
                    if len(parts) < 2:
                        print(f"Error: {command} requires
<hostname>:<port>")
                        continue

                    target = parts[1]
                    message = f"{nonce} {mac} {command} {target}"
                else:
                    message = f"{nonce} {mac} {command}"
```

```
                self.send_message(message)

                print("  Waiting 5s to gather replies.")
                if not self.collect_responses(RESPONSE_TIMEOUT):
                    print("  Disconnected while waiting for responses.")
                    break

                if command == "status":
                    self._handle_status_responses()
                elif command == "shutdown":
                    self._handle_shutdown_responses()
                elif command == "attack":
                    self._handle_attack_responses()
                elif command == "move":
                    self._handle_move_responses()

            else:
                print("Unknown command. Valid commands: status, shutdown,
attack <h:p>, move <h:p>, quit.")

        except KeyboardInterrupt:
            print("\nKeyboard interrupt. Exiting.")
            if self.socket:
                self.socket.close()
            break

        except Exception as e:
            print(f"Error: {e}")
            if self.socket:
                try:
                    self.socket.close()
                except:
                    pass
                self.socket = None
            break
```

This is the main control logic after the connection is initialized. The controller accepts commands from the user, generates appropriate nonces and MACs, and sends them as private messages to the channel. After sending a command, it waits for responses from the bots, collects them, and formats the results for display. This works similarly as before within the IRC bot class.

```
def collect_responses(self, timeout: float = RESPONSE_TIMEOUT) -> bool:
    self.responses = []
    start_time = time.time()

    while time.time() - start_time < timeout:
        if not self.socket:
            return False

        rlist, _, _ = select.select([self.socket], [], [], 0.1)
        if not rlist:
            continue

        try:
            data = self.socket.recv(4096).decode('utf-8', errors='ignore')
            if not data:
                print("Disconnected from server.")
                self.connected = False
                return False

            self.recv_buffer += data
            lines = self.recv_buffer.split('\n')
            self.recv_buffer = lines.pop() if lines else ""

            for line in lines:
                line = line.strip()
                if not line:
                    continue

                if line.startswith("PING"):
                    pong = line.replace("PING", "PONG", 1)
                    self.send(pong)
                    continue

                if "PRIVMSG" in line and f"#{self.channel}" in line:
                    try:
                        sender = line.split('!')[0][1:] if '!' in line else
"unknown"
                        message_start = line.find(" :",
line.find("PRIVMSG")) + 2

                        if message_start != -1:
                            message = line[message_start:]
                            if message.startswith('-'):
```

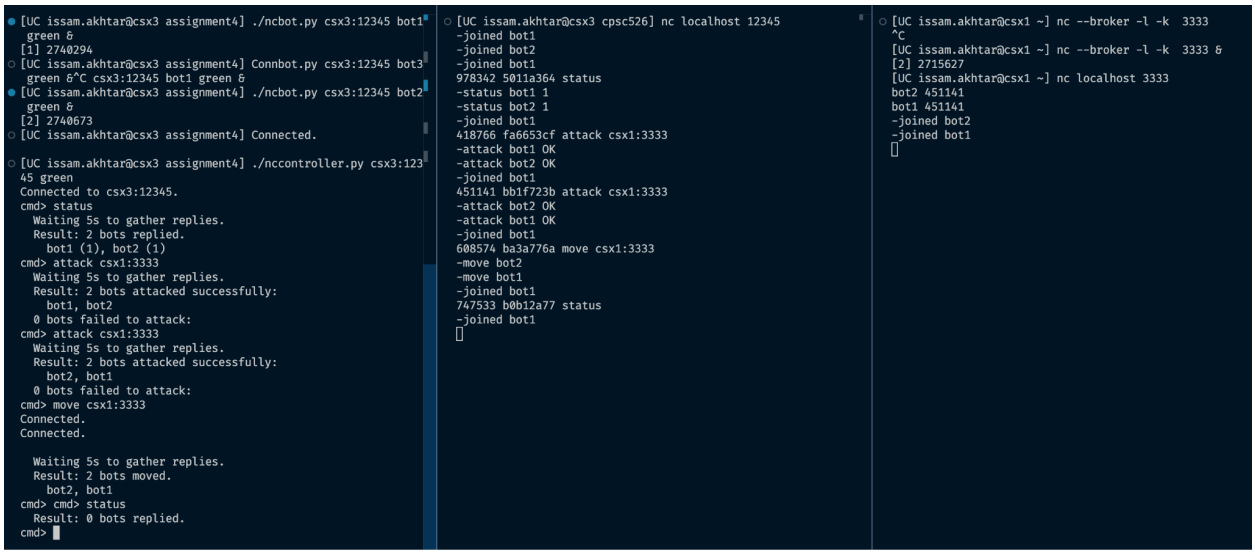
```
                                self.responses.append(message)
                    except IndexError:
                        pass
            except Exception as e:
                print(f"Error receiving data: {e}")
                self.connected = False
                return False

        return True
```

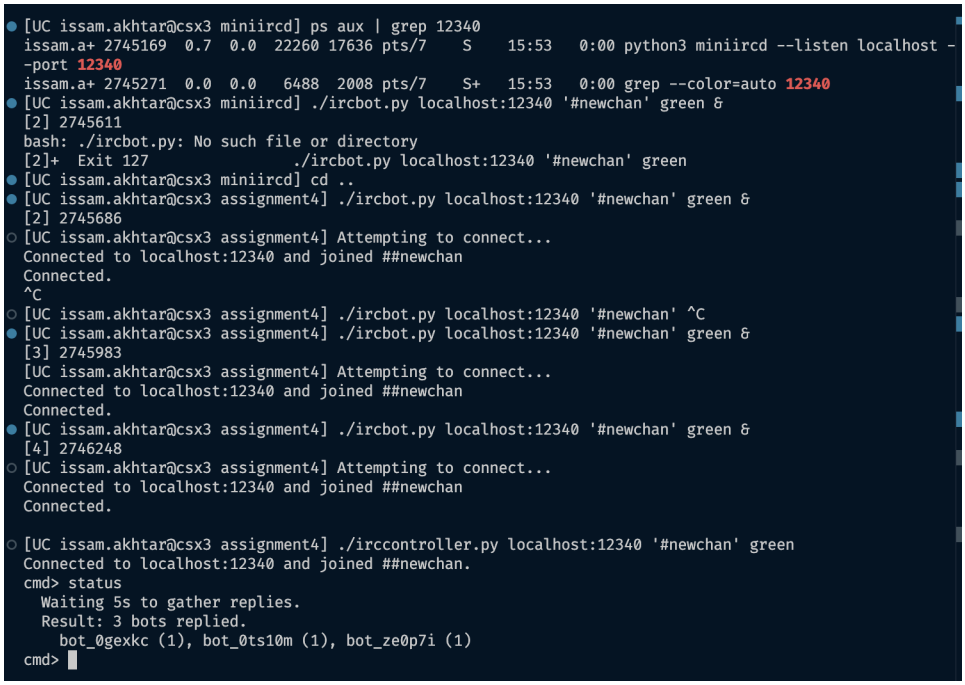
The collect_responses method is responsible for gathering responses from bots within an IRC channel over a specified timeout period. It initializes an empty list to store incoming responses and records the current time to track how long it has been collecting responses. Using a while loop, it continuously checks if the timeout has been reached. Inside the loop, it employs the select module to monitor the socket for incoming data without blocking indefinitely. When data is detected, it receives the data using socket.recv() and decodes it into a string. This data is accumulated in a receive buffer, which is then split into individual lines for processing. For each line, the method checks if it's a PING request from the IRC server, responding with a PONG message to maintain the connection. If the line is a PRIVMSG (private message) directed to the channel the bot is monitoring, it extracts the message content by finding the position of the message text and checks if it starts with '-', which indicates it's a response from a bot. These responses are collected in a list for further processing after the timeout period has expired. The method handles potential exceptions during data reception and ensures the bot remains connected to the server throughout the response collection process.

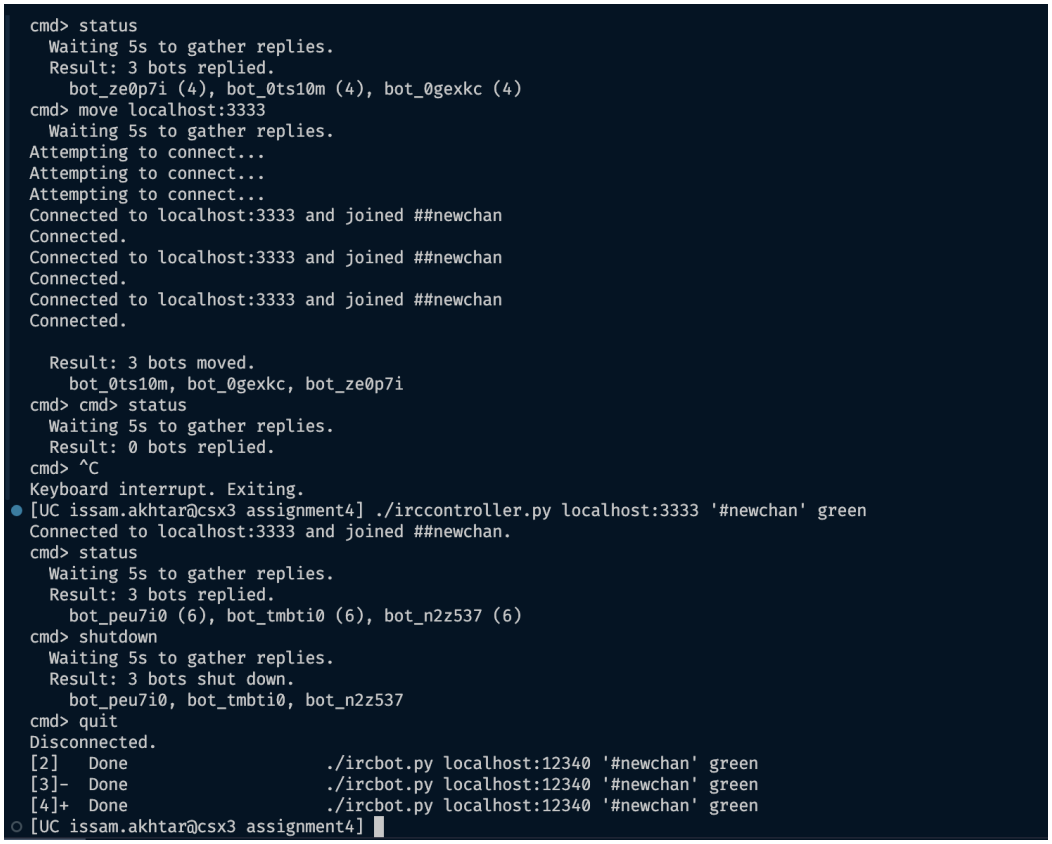
Testing

NCAT:


[UC issam.akhtar@csx3 assignment4] ./ncbot.py csx3:12345 bot1 green &
[1] 2740294
[UC issam.akhtar@csx3 assignment4] Connbot.py csx3:12345 bot3 green &^C csx3:12345 bot1 green &
[UC issam.akhtar@csx3 assignment4] ./ncbot.py csx3:12345 bot2 green &
[2] 2740673
[UC issam.akhtar@csx3 assignment4] Connected.
[UC issam.akhtar@csx3 assignment4] ./nccontroller.py csx3:12345 green
Connected to csx3:12345.
cmd> status
Waiting 5s to gather replies.
Result: 2 bots replied.
bot1 (1), bot2 (1)
cmd> attack csx1:3333
Waiting 5s to gather replies.
Result: 2 bots attacked successfully:
bot1, bot2
0 bots failed to attack:
cmd> attack csx1:3333
Waiting 5s to gather replies.
Result: 2 bots attacked successfully:
bot2, bot1
0 bots failed to attack:
cmd> move csx1:3333
Connected.
Connected.
Waiting 5s to gather replies.
Result: 2 bots moved.
bot2, bot1
cmd> cmd> status
Result: 0 bots replied.
cmd>
[UC issam.akhtar@csx3 cpsc526] nc localhost 12345
-joined bot1
-joined bot2
-joined bot1
978342 5011a364 status
-status bot1 1
-status bot2 1
-joined bot1
418766 fa6653cf attack csx1:3333
-attack bot1 OK
-attack bot2 OK
-joined bot1
451141 bb1f723b attack csx1:3333
-attack bot2 OK
-attack bot1 OK
-joined bot1
608574 ba3a776a move csx1:3333
-move bot2
-move bot1
-joined bot1
747533 b0b12a77 status
-joined bot1
[UC issam.akhtar@csx1 ~] nc --broker -l -k 3333
^C
[UC issam.akhtar@csx1 ~] nc --broker -l -k 3333 &
[2] 2715627
[UC issam.akhtar@csx1 ~] nc localhost 3333
bot2 451141
bot1 451141
-joined bot2
-joined bot1

IRC:


[UC issam.akhtar@csx3 miniircd] ps aux | grep 12340
issam.a+ 2745169 0.7 0.0 22260 17636 pts/7 S 15:53 0:00 python3 miniircd --listen localhost --port 12340
issam.a+ 2745271 0.0 0.0 6488 2008 pts/7 S+ 15:53 0:00 grep --color=auto 12340
[UC issam.akhtar@csx3 miniircd] ./ircbot.py localhost:12340 '#newchan' green &
[2] 2745611
bash: ./ircbot.py: No such file or directory
[2]+ Exit 127 ./ircbot.py localhost:12340 '#newchan' green
[UC issam.akhtar@csx3 miniircd] cd ..
[UC issam.akhtar@csx3 assignment4] ./ircbot.py localhost:12340 '#newchan' green &
[2] 2745686
[UC issam.akhtar@csx3 assignment4] Attempting to connect...
Connected to localhost:12340 and joined ##newchan
Connected.
^C
[UC issam.akhtar@csx3 assignment4] ./ircbot.py localhost:12340 '#newchan' ^C
[UC issam.akhtar@csx3 assignment4] ./ircbot.py localhost:12340 '#newchan' green &
[3] 2745983
[UC issam.akhtar@csx3 assignment4] Attempting to connect...
Connected to localhost:12340 and joined ##newchan
Connected.
[UC issam.akhtar@csx3 assignment4] ./ircbot.py localhost:12340 '#newchan' green &
[4] 2746248
[UC issam.akhtar@csx3 assignment4] Attempting to connect...
Connected to localhost:12340 and joined ##newchan
Connected.
[UC issam.akhtar@csx3 assignment4] ./irccontroller.py localhost:12340 '#newchan' green
Connected to localhost:12340 and joined ##newchan.
cmd> status
Waiting 5s to gather replies.
Result: 3 bots replied.
bot_0gexkc (1), bot_0ts10m (1), bot_ze0p7i (1)
cmd>


cmd> status
Waiting 5s to gather replies.
Result: 3 bots replied.
bot_ze0p7i (4), bot_0ts10m (4), bot_0gexkc (4)
cmd> move localhost:3333
Waiting 5s to gather replies.
Attempting to connect...
Attempting to connect...
Attempting to connect...
Connected to localhost:3333 and joined ##newchan
Connected.
Connected to localhost:3333 and joined ##newchan
Connected.
Connected to localhost:3333 and joined ##newchan
Connected.
Result: 3 bots moved.
bot_0ts10m, bot_0gexkc, bot_ze0p7i
cmd> cmd> status
Waiting 5s to gather replies.
Result: 0 bots replied.
cmd> ^C
Keyboard interrupt. Exiting.
[UC issam.akhtar@csx3 assignment4] ./irccontroller.py localhost:3333 '#newchan' green
Connected to localhost:3333 and joined ##newchan.
cmd> status
Waiting 5s to gather replies.
Result: 3 bots replied.
bot_peu7i0 (6), bot_tmbti0 (6), bot_n2z537 (6)
cmd> shutdown
Waiting 5s to gather replies.
Result: 3 bots shut down.
bot_peu7i0, bot_tmbti0, bot_n2z537
cmd> quit
Disconnected.
[2] Done ./ircbot.py localhost:12340 '#newchan' green
[3]- Done ./ircbot.py localhost:12340 '#newchan' green
[4]+ Done ./ircbot.py localhost:12340 '#newchan' green
[UC issam.akhtar@csx3 assignment4]