ECE 470 Spring 2025

SMART HOME REMOTE ACCESS SYSTEM

Jose Cardenas C23777927

Dr. Nigel John

System vision and goals

System Vision/Goals

The Smart Home Remote Access System is a client-server application that enables users to control smart home devices remotely over a network using the TCP protocol.

Project Description

The goal of the project is to design and implement a remote-access system that allows a user to control smart home devices such as lights, window blinds, alarms, and electronic locks. The system follows a client-server architecture, where the client application sends commands to a home server, which then processes and executes the requested operations. The server ensures reliable communication using TCP and restricts access through a login authentication mechanism. Users can list devices, check their status, and modify their states as needed.

Requirements

- Users must log in before issuing any commands.
- Ability to retrieve a list of all devices in a home.
- Users can check the current state of a device (e.g., whether a light is on or off).
- Users can change device states (e.g., turning lights on/off, locking/unlocking doors, arming/disarming the alarm).
- The system uses TCP to ensure commands are successfully transmitted.
- Only one user can be logged in at a time.

Architectural solution

The chosen architectural pattern is the **Client-Server Architecture.**

The way it works:

- The system centralizes control by using a dedicated server within the home to manage all smart devices. Instead of each device operating independently, all commands and data pass through the server, ensuring consistency and security in device management.
- The client application acts as the interface through which users interact with their smart home. When a user wants to perform an action—such as turning a light on, unlocking a door, or checking the status of the alarm—the client sends a request to the server. The server processes these requests by retrieving information or executing commands on the appropriate smart device. It maintains an updated status of all connected devices and ensures commands are executed correctly.
- TCP is used as the transport layer protocol to facilitate reliable communication between the client and the server. TCP ensures that data packets are delivered in order and

without loss, which is crucial for executing smart home commands like ensuring that a "lock door" command is not lost due to network issues.

Benefits:

- The server acts as the primary access control point, managing user authentication before allowing any device interaction. This ensures that only authorized users can issue commands to smart home devices, preventing unauthorized access.
- The system is designed to be easily expandable. New smart devices such as a thermostat or security cameras can be integrated into the home without requiring modifications to the client application. The server handles all device management, meaning the client only needs to interact with the server rather than individual devices.
- The server serves as a centralized point of control, simplifying device management and communication. Users don't have to interact with multiple separate devices; instead, they issue commands through the client, and the server ensures that those commands are carried out efficiently. This also enables unified logging and monitoring, allowing users to track changes in the smart home system.

Drawbacks:

- The system relies on an always-on server within the home, meaning the server must remain powered and always connected to the network to function properly. This may increase energy consumption, and if the server experiences a power outage or connection loss, users will be unable to control their smart home devices remotely.
- The server acts as a single point of failure in the system in this scenario. If the server crashes or malfunctions, the entire smart home system becomes inaccessible, preventing users from checking statuses or issuing commands. This could be problematic, especially in scenarios where security devices like alarms and door locks are involved. To mitigate this, redundancy measures or backup solutions such as a secondary server or cloud-based failover would need to be considered.

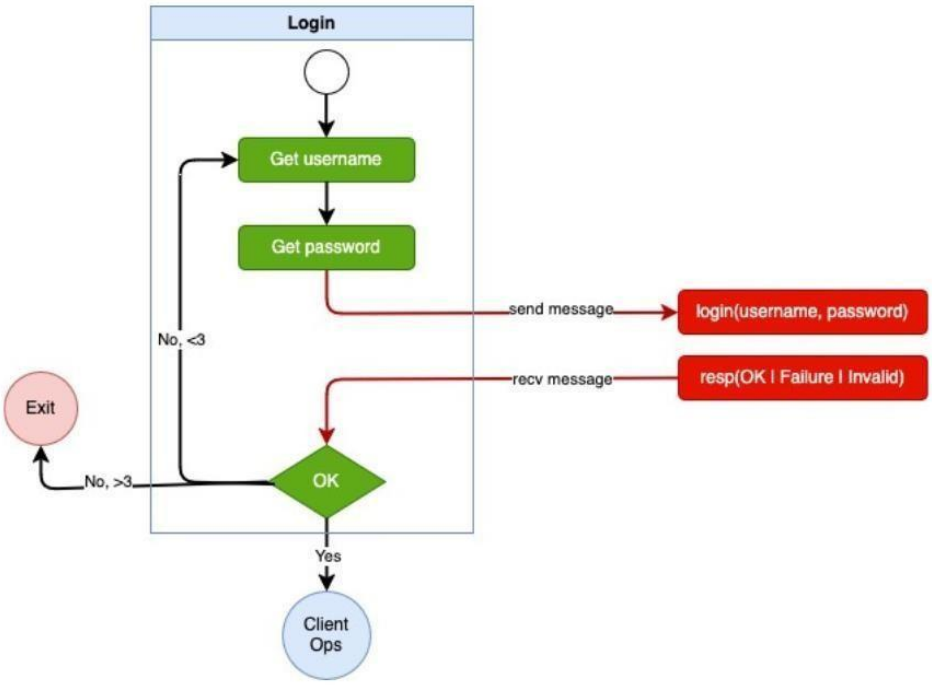
A considered alternative was the Peer-to-Peer Architecture, however; this wouldn't fit properly due to lack of centralized control which would result in devices directly communicating, making user authentication difficult. Moreover, each device would need to maintain its own connection list, leading to high overhead. Additionally, unauthorized users might access devices if not properly secured.

Another alternative would be a Monolithic Architecture. The latter wouldn't work well here since it consists of a singular, massive application handling all operations which would be difficult to modify or scale. Additionally, each device would need to maintain its own

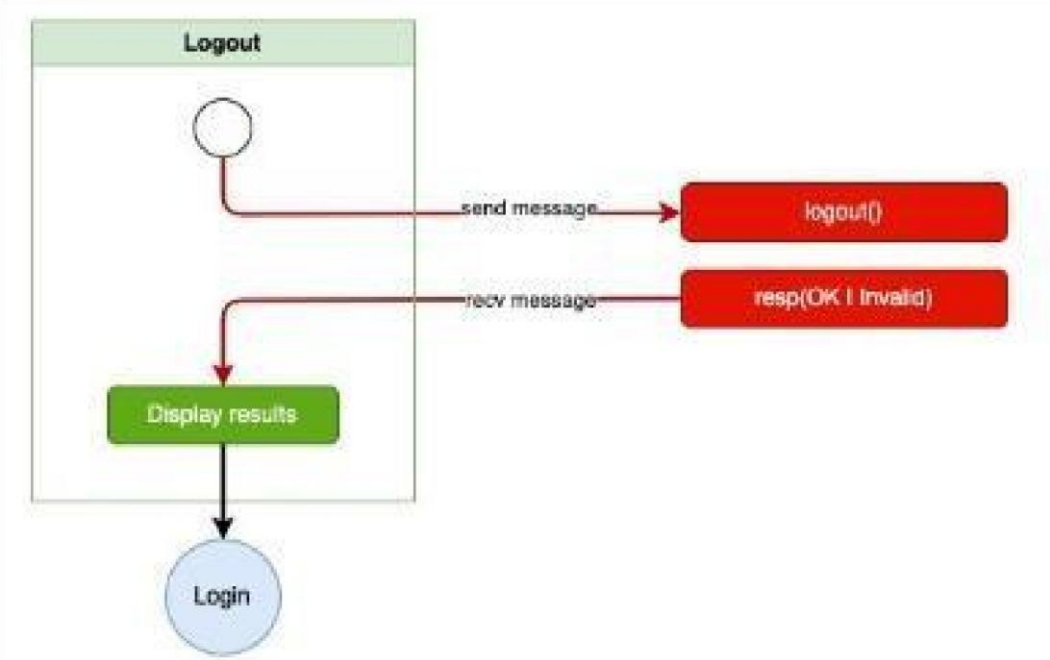
connection list, leading to high overhead. There is also a security risk due to the potential for unauthorized users to have access to devices if not properly secured.

General Design

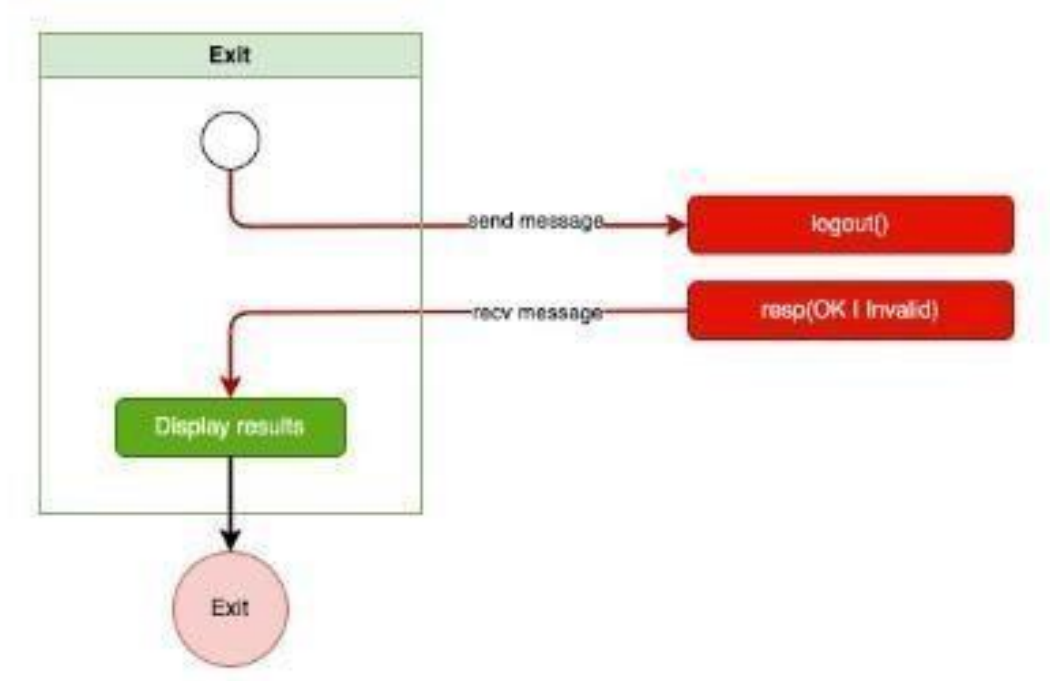
Scenarios/Storyboards/Use Cases



This state chart represents the login process within the smart home remote access system. The user is first prompted to enter their username and password. The entered credentials are then sent as a login(username, password) request to the server. The server responds with either an "OK" message if authentication is successful or a "Failure | Invalid" message if the credentials are incorrect. If the user fails authentication three times, the system exits. If the login is successful, the user is granted access to client operations, allowing them to interact with the system.

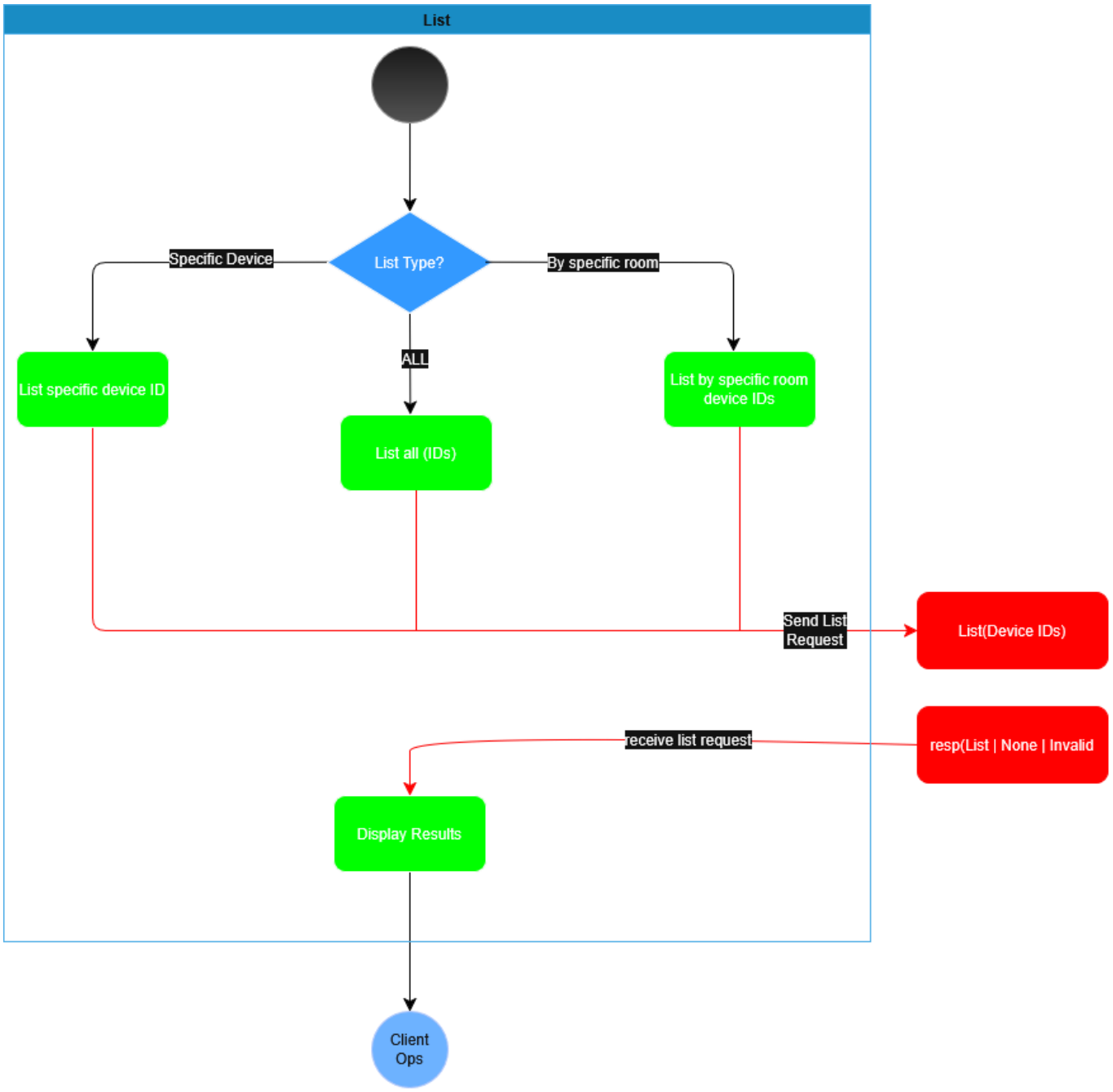


This diagram illustrates the logout functionality of the system. When a user initiates a logout request, the system sends a logout() command to the server. The server then processes this request and responds with either "OK" (successful logout) or "Invalid" (failed logout attempt). Once a valid logout is confirmed, the system displays the results and redirects the user back to the login screen. This ensures that each session is properly terminated, enhancing security by preventing unauthorized access.



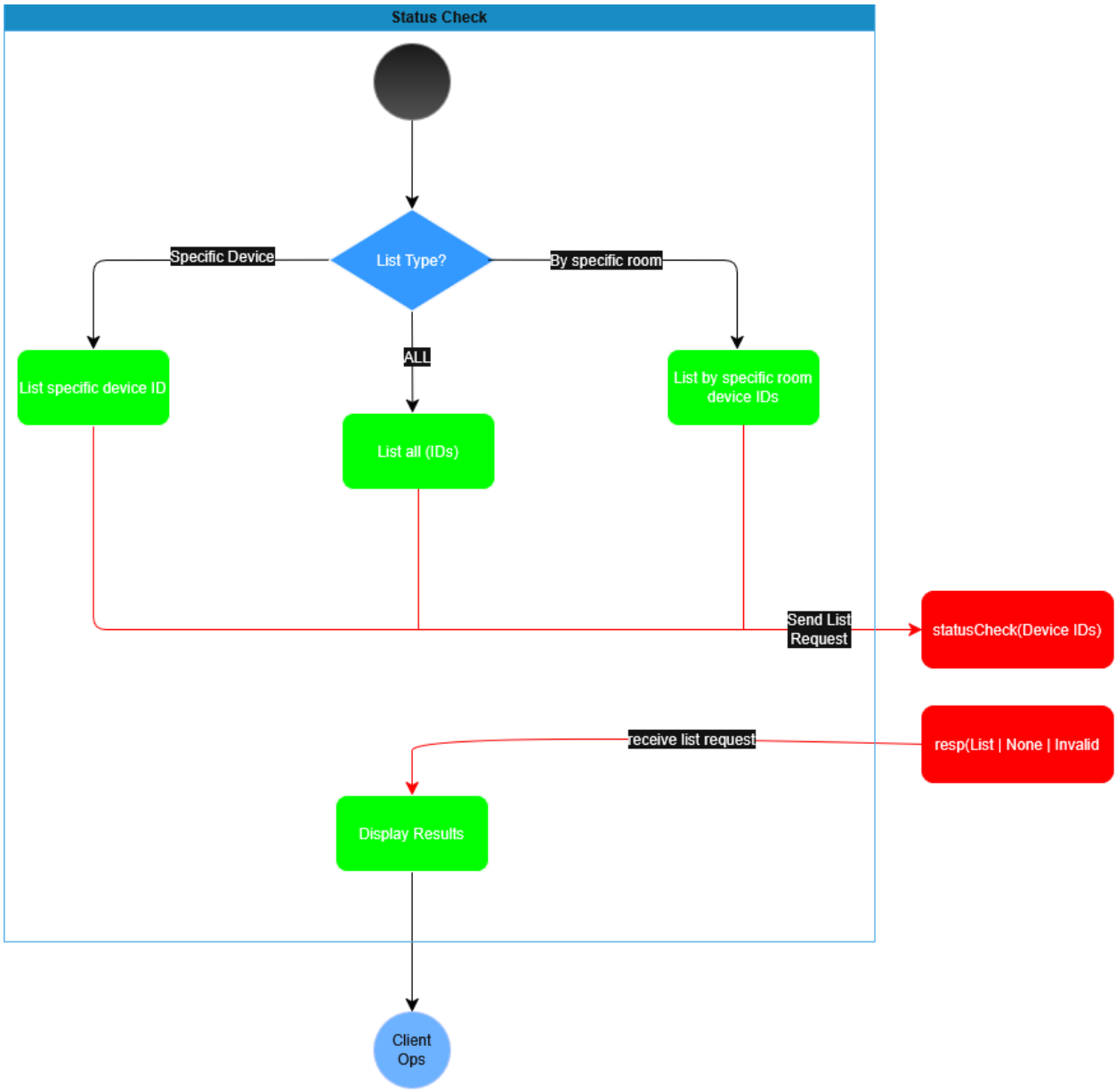
The exit process follows a similar flow to logout but is designed to completely terminate the client session. When a user selects the exit option, a logout() command is sent to the server to properly

end the session. The server then responds with either "OK" (logout successful) or "Invalid" (logout failed). After displaying the logout status, the system shuts down the client application. This ensures that the system properly handles session closures and prevents lingering authenticated sessions that could pose security risks.



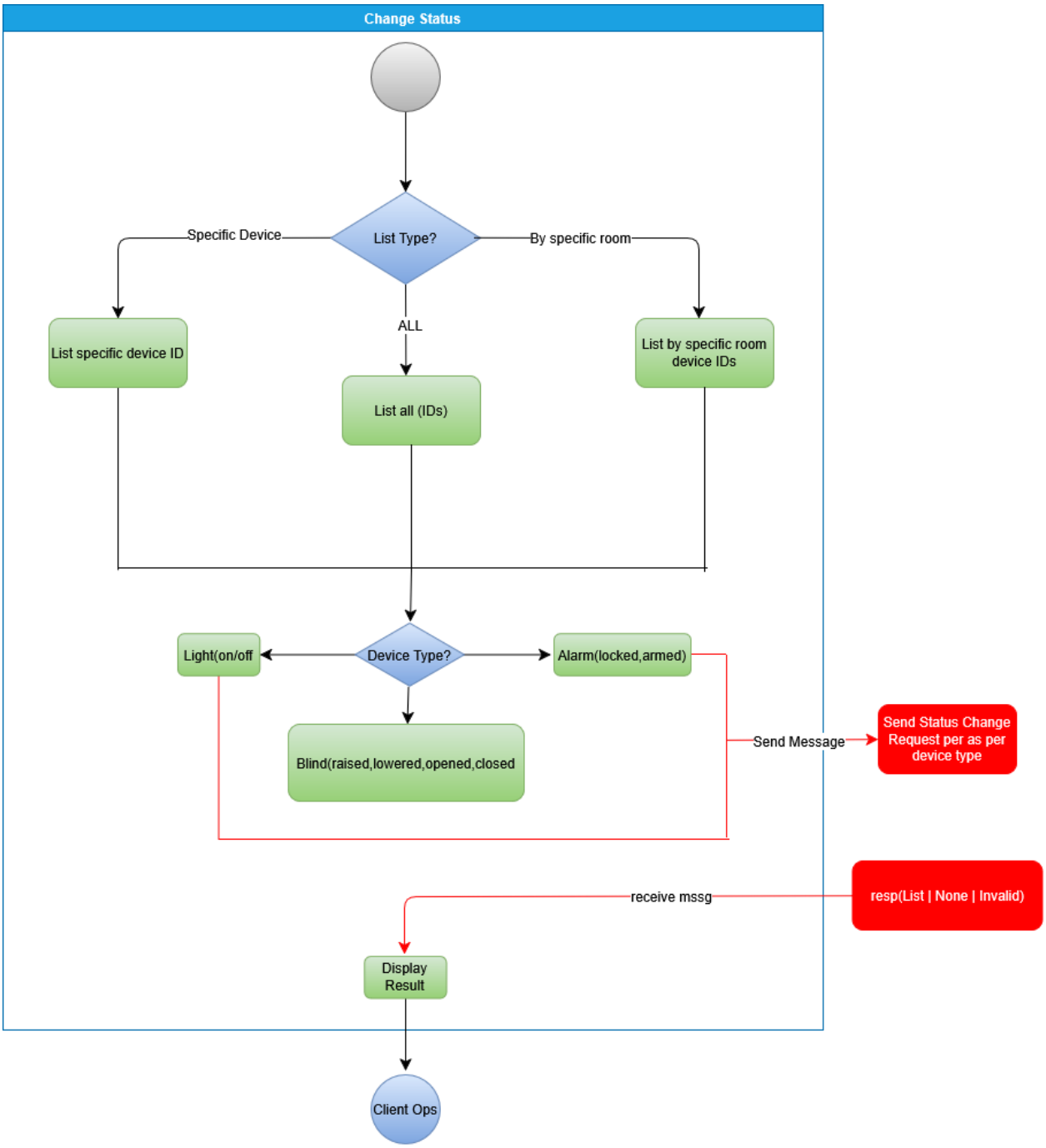
The List Devices process in the Smart Home Remote Access System allows users to retrieve information about connected devices based on different criteria. When initiated, the system

prompts the user to choose between listing a specific device by its ID, retrieving all devices within a specific room, or obtaining a complete list of all smart home devices. Depending on the selection, the system sends a list request to the server, which processes the query and returns an appropriate response. If the request is valid, the server responds with the relevant device IDs; otherwise, it may return None if no devices are found or an Invalid response if the request is improperly formatted or unauthorized. Upon receiving the response, the system processes and displays the results, allowing the user to make informed decisions about their smart home devices. This structured approach ensures flexibility in device retrieval, scalability for managing multiple devices across different rooms and homes, and security by enforcing controlled access to smart home data.



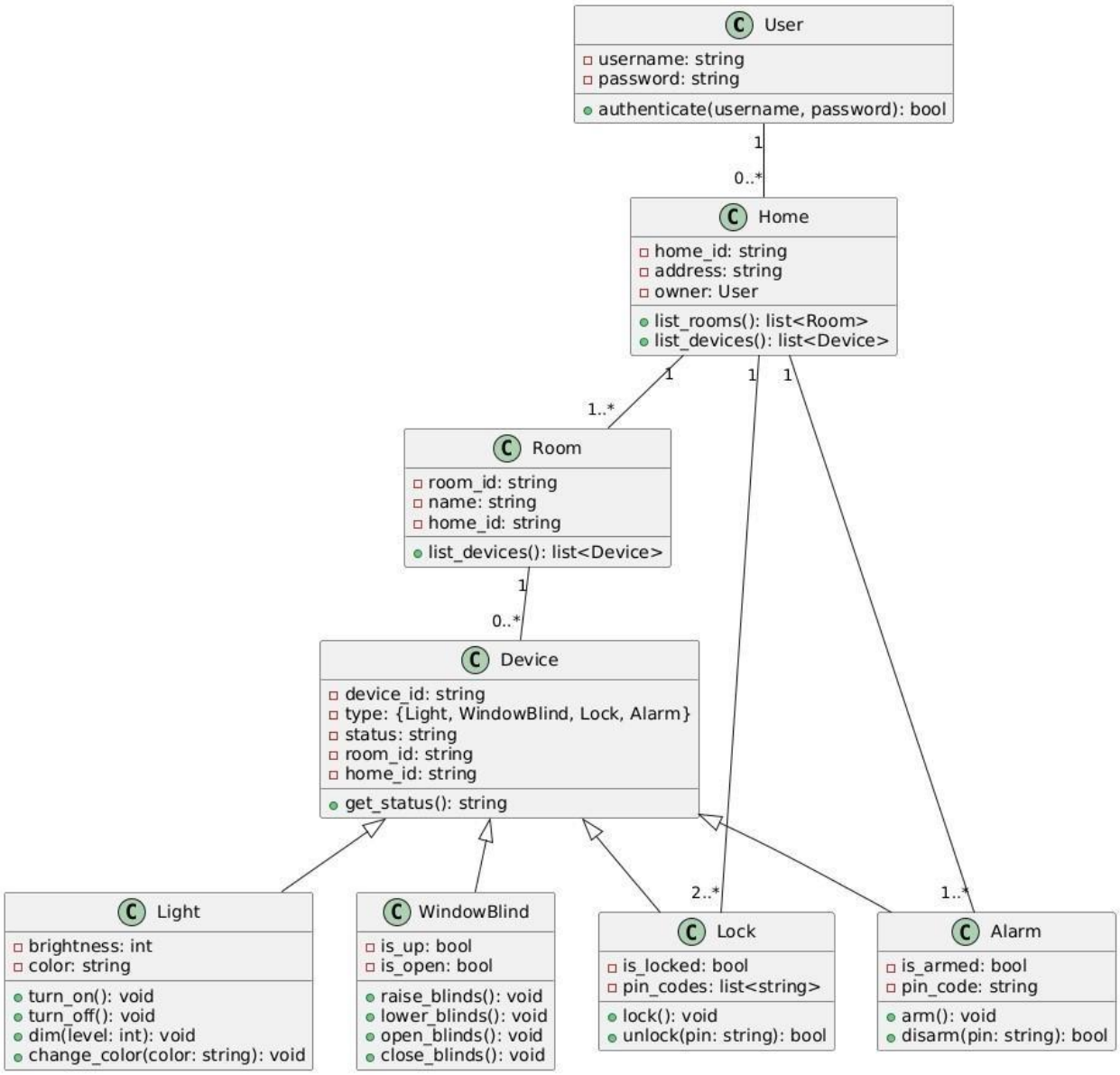
The Status Check process in the Smart Home Remote Access System enables users to verify the current state of smart devices within a home. The system first prompts the user to specify the type of status check they wish to perform—whether for a specific device, for all devices in

a particular room, or for all devices in the home. Once the user makes a selection, the system sends a statusCheck request to the server, which processes the request and retrieves the relevant device status data. The server responds with a list of statuses corresponding to the requested devices, or it may return None if no devices match the query, or Invalid if the request is not properly formatted or lacks authorization. Once the response is received, the system processes and displays the results to the user, ensuring they have up-to-date information about their smart home devices. This structured approach enhances usability by allowing users to efficiently check device statuses across multiple levels while maintaining security and reliability.



The Change Status flowchart illustrates the process of modifying the state of various smart home devices. The process begins with selecting a list type, where the user can choose to modify a specific device, devices in a specific room, or all devices within a home. Once the relevant devices are identified, the system determines the device type, categorizing them into lights, blinds, locks, or alarms. Each device type supports specific status changes, such as turning lights on/off or adjusting brightness, raising/lowering blinds, locking/unlocking doors, or arming/disarming alarms. A request is then sent to the server to execute the status change based on the device type. The server processes this request and returns a response indicating whether the status update was successful, invalid, or no change was made. The final status is then displayed to the user, ensuring that the operation is confirmed before proceeding with

further client actions. This structured approach ensures efficient device management while maintaining security constraints, particularly for locks and alarms requiring PIN authentication. Data Model



Initial Data Model

The Smart Home Remote Access System employs a structured, object-oriented data model to facilitate secure and efficient management of multiple homes, rooms, and smart devices. The design is centered around user authentication, hierarchical device organization, and secure access control, ensuring scalability and maintainability.

At the core of the system, the USER entity manages authentication and ownership of multiple HOME instances, allowing users to control various smart homes within a single interface. Each HOME contains at least one ROOM, organizing devices based on physical

locations. The DEVICE superclass serves as an abstraction for all smart home components, including LIGHTS, WINDOW BLINDS, LOCKS, and ALARMS, ensuring consistency in device management while allowing extensibility for future integrations.

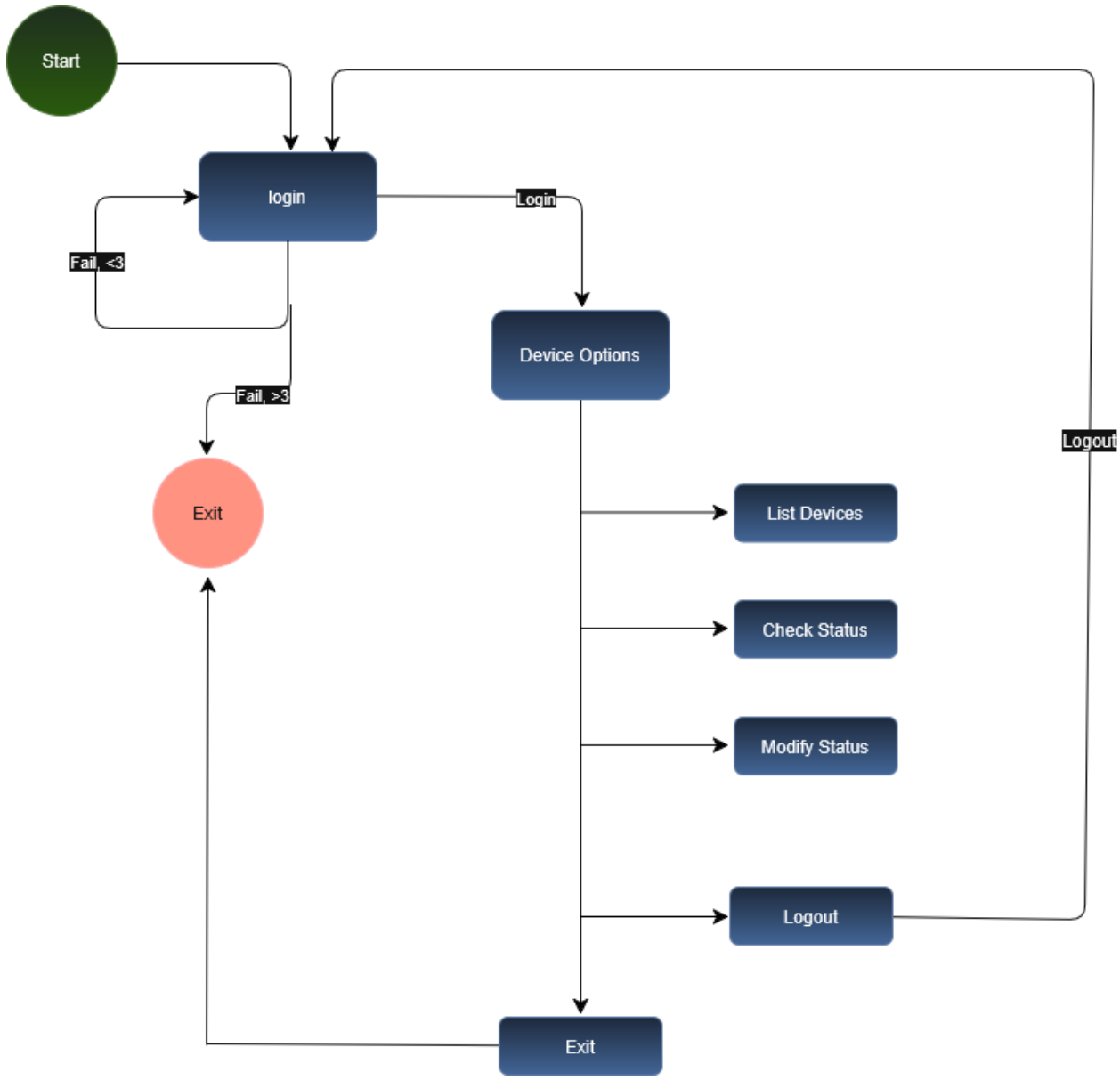
LIGHTS support multiple states such as on/off, brightness adjustment, and color changes, while WINDOW BLINDS track their open/closed and up/down status, enhancing convenience and usability. Security devices, including LOCKS and ALARMS, require authentication via PIN codes, ensuring that only authorized users can unlock doors or disable security measures. Each LOCK maintains a list of PIN codes for controlled access, while the ALARM system enforces a strict security mechanism for arming and disarming. The system ensures proper device-scoping by associating every device with both a HOME ID and a ROOM ID, preventing conflicts across multiple smart homes while maintaining logical separation within each household. To enforce security best practices, LOCKS require at least two instances per home, and each home must have at least one ALARM, aligning with realworld safety standards.

By implementing a hierarchical organization of homes, rooms, and devices, the system achieves a high level of modularity and scalability, allowing future expansion to accommodate new smart devices. The use of a generic DEVICE superclass ensures code reusability, simplifying the integration of additional smart home components while maintaining a unified interface for device control.

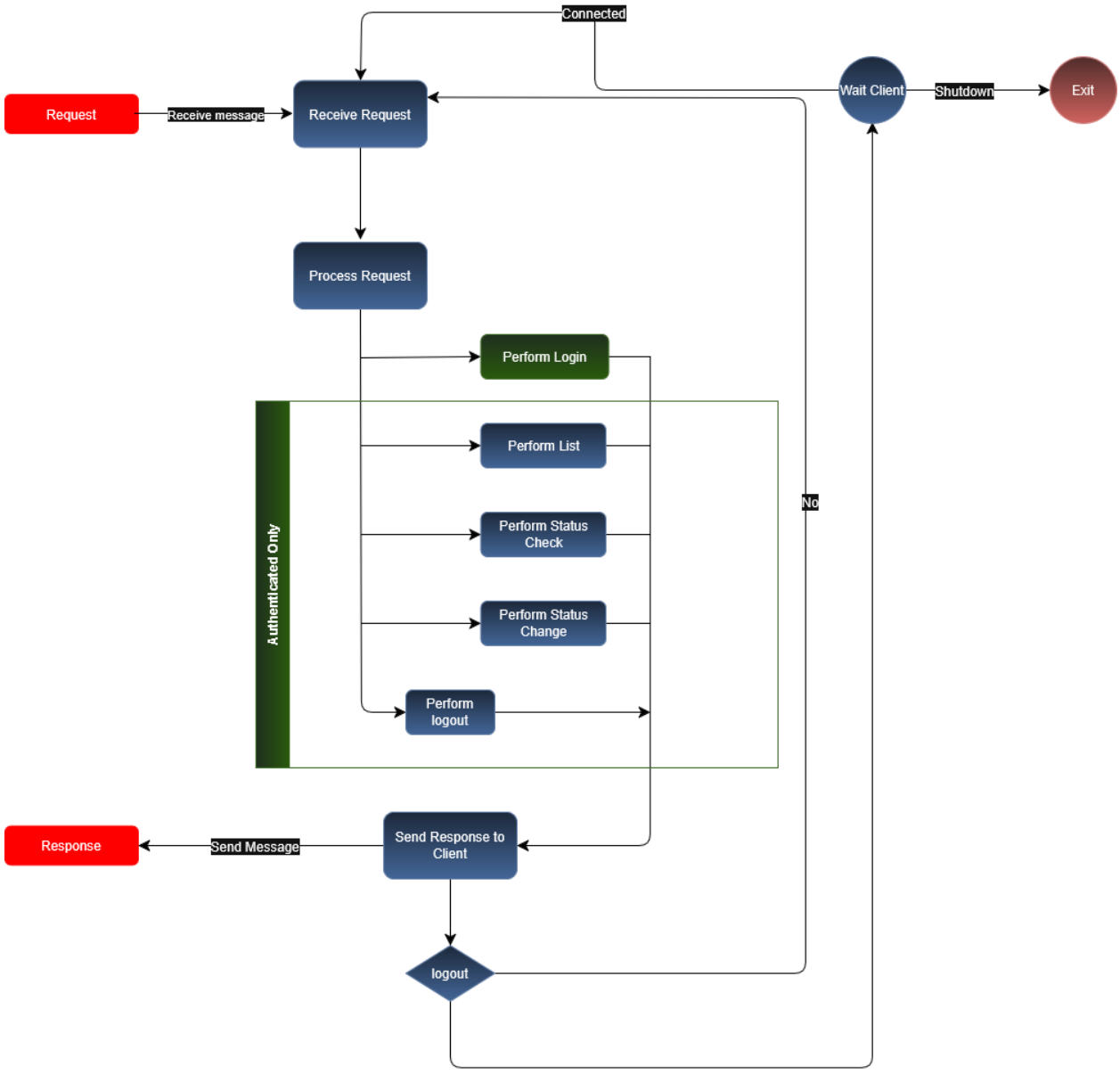
This design effectively balances security, scalability, and usability, making it a robust foundation for managing smart home environments remotely while ensuring seamless user interaction and secure access control.

State Charts

Client State Chart



Server State Chart



Application Protocol

<Initial design of the application protocol layer and message structure> Middle-Ware Application Protocol

<from Part1, design of the Client<->Middle-Ware Application Protocol>

Operations – P2, P3

<Definitions of operations for the systems> Client

Operations – P2

<Definition of client operations>

Middle-Ware Server Operations – P2

<Definition of server operations>

Example Scenarios

<Show an example of different scenarios using the proposed data objects, operations, and application messages. These are the test cases to be used in the Testing section.>

--- Test Message ---
Marshaled: type=REQS.TURN_ON&device=light&room=Kitchen
Unmarshaled: {'type': 'REQS.TURN_ON', 'device': 'light', 'room': 'Kitchen'}
[Server Response] Kitchen light status: on

--- Test Message ---
Marshaled: type=REQS.UNLOCK&device=lock&room=Living Room&pin=1234
Unmarshaled: {'type': 'REQS.UNLOCK', 'device': 'lock', 'room': 'Living Room', 'pin': '1234'}
[Server Response] Living Room lock: Unlocked

--- Test Message ---
Marshaled: type=REQS.UNLOCK&device=lock&room=Living Room&pin=0000
Unmarshaled: {'type': 'REQS.UNLOCK', 'device': 'lock', 'room': 'Living Room', 'pin': '0000'}
[Server Response] Lock status: Locked (Invalid PIN)

--- Test Message ---
Marshaled: type=REQS.UNLOCK&device=alarm&room=Bedroom&pin=9999
Unmarshaled: {'type': 'REQS.UNLOCK', 'device': 'alarm', 'room': 'Bedroom', 'pin': '9999'}
[Server Response] Alarm status: Armed (Invalid PIN)

--- Test Message ---
Marshaled: type=REQS.TURN_ON&device=fan&room=Living Room
Unmarshaled: {'type': 'REQS.TURN_ON', 'device': 'fan', 'room': 'Living Room'}
[Server Response] Error: Invalid device type 'fan'

--- Test Message ---
Marshaled: type=REQS.CHECK_STATUS

```
Unmarshaled: {'type': 'REQS.CHECK_STATUS'} [Server
Response] Error: Empty message received.

--- Test Message ---
Marshaled: type=REQS.TURN_OFF&device=light
Unmarshaled: {'type': 'REQS.TURN_OFF', 'device': 'light'}
[Server Response] Error: Missing 'room' parameter in message.

--- Test Message ---
Marshaled: type=REQS.TURN_OFF&device=windowblind&room=Living Room
Unmarshaled: {'type': 'REQS.TURN_OFF', 'device': 'windowblind', 'room': 'Living Room'}
[Server Response] Blinds in Living Room: Down

--- Test Message ---
Marshaled: type=REQS.UNLOCK&device=alarm&room=Bedroom&pin=4321
Unmarshaled: {'type': 'REQS.UNLOCK', 'device': 'alarm', 'room': 'Bedroom', 'pin': '4321'}
[Server Response] Alarm in Bedroom: Disarmed

===== FINAL DEVICE STATES =====
Room: Living Room
- Light (light_lr): off
- WindowBlind (blind_lr): off
- Lock (lock_lr): off
Room: Bedroom
- Alarm (alarm_bd): off
Room: Kitchen
- Light (light_kt): on
```

Implementation – P2, P3, P4

Application Protocol

Middle-Ware Application Protocol – P2, P3

The implementation of the Client-Middleware Application Protocol is central to ensuring seamless communication between the smart home client and server components. This protocol is responsible for structuring messages, handling different request types, and ensuring the correct exchange of data between the client application and the middleware, which subsequently interacts with the smart home system.
The communication protocol follows a structured request-response model, where the client sends requests formatted using a custom-defined messaging format, and the

server processes these requests and returns the appropriate responses. Each message adheres to a key-value structure, where the request type is specified using a predefined integer code, and any additional parameters required for processing are included as key-value pairs.

To facilitate this structured exchange, a CSmessage class was implemented to encapsulate message formatting, serialization, and deserialization. Each message consists of a type field that defines the operation being requested, such as login (LGIN), logout (LOUT), listing available devices (LIST), and changing the status of a device (CHG_STATUS). The marshal method converts the internal message representation into a string format suitable for network transmission, while the unmarshal method reconstructs a message from a received string.

The middleware layer processes these messages using a SmartHomePDU (Protocol Data Unit) class, which is responsible for sending and receiving structured data over a TCP connection. The SmartHomePDU ensures that messages are transmitted in their entirety by implementing methods to handle the proper segmentation and reassembly of data. This guarantees that all bytes of a message are received correctly before processing. Upon receiving a request from the client, the middleware extracts the message type and determines the appropriate course of action. If the request is for authentication, it verifies the provided credentials before responding with either a success or failure message. If the request involves listing available devices, the server responds with a list of registered smart home devices. For change status requests, the middleware updates the status of the specified device and acknowledges the modification. Each response message mirrors the request structure and includes relevant data indicating the outcome of the requested operation.

To verify the correct implementation of this communication protocol, extensive testing was conducted using Wireshark. By capturing network packets exchanged between the client and server, the integrity of the message format, correct serialization and deserialization, and adherence to the expected request-response structure were validated. The captured data confirmed that messages were properly transmitted and processed, with responses accurately reflecting the requested operations.
The middleware protocol implementation successfully enables structured and efficient communication between the client and server, ensuring that smart home operations can be performed reliably. The modular design allows for future expansions, where additional request types can be integrated without major modifications to the existing system.

Operations

<Implementation details of the application operations>

The implementation of operations within the smart home system involves defining the functionality of both the client and middleware server. These operations include user authentication, device listing, status modification, and session termination, ensuring seamless communication between the client application and the middleware.

Client Ops – P3

<Implementation of the Client operations>

The client operations handle the construction and transmission of requests to the server. Each request follows a structured format using the CSmessage class, which assigns a request type and attaches relevant parameters. The client first sends a login request, providing a username and password. If authentication is successful, it proceeds to request a list of available devices, modify a device's status, and finally, send a logout request. The client utilizes the SmartHomePDU class to handle message transmission over a TCP connection, ensuring complete message delivery and response handling. Upon receiving a response, the client interprets the message and displays the status accordingly.

Middle-Ware Server Ops – P3

<Implementation of the Middle-Ware server operations>

The middleware server operations process incoming client requests and generate appropriate responses. The server listens for connections and, upon receiving a message, extracts the request type and associated parameters. Authentication requests verify credentials before granting access. Device listing requests return a list of registered devices. Change status requests modify the state of a specified device, ensuring that updates are processed correctly. Logout requests terminate the client session. The server constructs response messages using CSmessage, serializes them, and transmits them back to the client. Proper error handling is implemented to ensure that invalid or malformed requests are managed without causing system failures.

Applications – P4

<Implementation details of the application protocols>

Client Application – P4

<Implementation of the Client Application>

Middle-Ware Server – P4

<Implementation of the Middle-Ware Server>

Testing – P2, P3, P4

<Testing of the client and servers. Include sections as necessary to match the General Design architecture of the system. Make sure you include screenshots of any tests.>

Application Protocol – P2, P3, P4

<Testing details of the application protocols>

Middle-Ware Application Protocol – P2, P3

<Testing of the Client<->Middle-Ware Application Protocol. Use your scenarios to show specific messages>

A series of tests were conducted to validate the functionality of the Smart Home Remote Access System, ensuring proper authentication, device management, and request handling. The login system was successfully tested, allowing a valid user to log in while preventing duplicate logins and rejecting incorrect credentials. The system correctly listed available devices, returning their statuses and attributes in the expected format. Device control requests were processed accurately, enabling updates to device states such as brightness and color. Additionally, the logout functionality worked as intended, ensuring that a user could not perform actions after logging out. The system also appropriately handled unknown request types, rejecting invalid commands. All messages were properly marshaled and unmarshaled, with no errors in communication between the client and server. The results confirm that the system meets its intended functionality, with stable request handling and secure user authentication

Client Output:

===== Running Full Test Suite =====

[TEST] Successful Login

[CLIENT] ---- Sending Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.LGIN: 100>, 'username': 'admin', 'password': 'pass123'}
Marshaled: type=100&username=admin&password=pass123

[CLIENT] ---- Received Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.LGIN: 100>, 'status': 'Login successful'}
Marshaled: type=100&status=Login successful

[TEST] Unsuccessful Login (Invalid Password)

[CLIENT] ---- Sending Request ----

Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.LGIN: 100>, 'username': 'admin', 'password': 'wrongpass'}
Marshaled: type=100&username=admin&password=wrongpass

[CLIENT] ---- Received Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.LGIN: 100>, 'status': 'Already logged in'}
Marshaled: type=100&status=Already logged in

[TEST] Login Attempt When Already Logged In

[CLIENT] ---- Sending Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.LGIN: 100>, 'username': 'admin', 'password': 'pass123'}
Marshaled: type=100&username=admin&password=pass123

[CLIENT] ---- Received Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.LGIN: 100>, 'status': 'Already logged in'}
Marshaled: type=100&status=Already logged in

[TEST] Listing Devices

[CLIENT] ---- Sending Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.LIST: 102>}
Marshaled: type=102

[CLIENT] ---- Received Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.LIST: 102>, 'status': 'OK', 'devices': 'light,status=off,brightness=100,color=255,255,255|lock,status=unlocked|alarm,status=disarmed,code=1234'}
Marshaled: type=102&status=OK&devices=light,status=off,brightness=100,color=255,255,255|lock,status=unlocked|alarm,status=disarmed,code=1234

[TEST] Changing Device Status (Turning Light ON)

[CLIENT] ---- Sending Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.CHG_STATUS: 103>, 'device': 'light', 'status': 'on'}
Marshaled: type=103&device=light&status=on

[CLIENT] ---- Received Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.CHG_STATUS: 103>, 'status': 'light updated successfully'}
Marshaled: type=103&status=light updated successfully

[TEST] Setting Device Attributes (Brightness and Color)

[CLIENT] ---- Sending Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.CHG_STATUS: 103>, 'device': 'light', 'brightness': '75', 'color': '255,0,0'}
Marshaled: type=103&device=light&brightness=75&color=255,0,0

[CLIENT] ---- Received Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.CHG_STATUS: 103>, 'status': 'light updated successfully'}
Marshaled: type=103&status=light updated successfully

[TEST] Attempting to Change Status Without Logging In

[CLIENT] ---- Sending Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.CHG_STATUS: 103>, 'device': 'lock', 'status': 'locked'}
Marshaled: type=103&device=lock&status=locked

[CLIENT] ---- Received Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.CHG_STATUS: 103>, 'status': 'lock updated successfully'}
Marshaled: type=103&status=lock updated successfully

[TEST] Logout

[CLIENT] ---- Sending Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.LOUT: 101>}
Marshaled: type=101

[CLIENT] ---- Received Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.LOUT: 101>, 'status': 'Logged out successfully'}
Marshaled: type=101&status=Logged out successfully

[TEST] Sending an Unknown Request Type

[CLIENT] ---- Sending Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': 999, 'param': 'unknown'}
Marshaled: type=999&param=unknown

Server Output:

[SERVER] Server started on 127.0.0.1:5000

[SERVER] ---- New Connection ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53490)

[SERVER] ---- Received Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.LGIN: 100>, 'username': 'admin', 'password': 'pass123'}
Marshaled: type=100&username=admin&password=pass123

[SERVER] ---- Sending Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.LGIN: 100>, 'status': 'Login successful'}
Marshaled: type=100&status=Login successful

[SERVER] ---- Connection Closed ----
Time: 2025-02-28 15:14:22

Client Address: ('127.0.0.1', 53490)

[SERVER] ---- New Connection ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53491)

[SERVER] ---- Received Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.LGIN: 100>, 'username': 'admin', 'password': 'wrongpass'}
Marshaled: type=100&username=admin&password=wrongpass

[SERVER] ---- Sending Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.LGIN: 100>, 'status': 'Already logged in'}
Marshaled: type=100&status=Already logged in

[SERVER] ---- Connection Closed ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53491)

[SERVER] ---- New Connection ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53492)

[SERVER] ---- Received Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.LGIN: 100>, 'username': 'admin', 'password': 'pass123'}
Marshaled: type=100&username=admin&password=pass123

[SERVER] ---- Sending Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.LGIN: 100>, 'status': 'Already logged in'}
Marshaled: type=100&status=Already logged in

[SERVER] ---- Connection Closed ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53492)

[SERVER] ---- New Connection ----
Time: 2025-02-28 15:14:22

Client Address: ('127.0.0.1', 53493)

[SERVER] ---- Received Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.LIST: 102>}
Marshaled: type=102

[SERVER] ---- Sending Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.LIST: 102>, 'status': 'OK', 'devices': 'light,status=off,brightness=100,color=255,255,255|lock,status=unlocked|alarm,status=disarmed,code=1234'}
Marshaled: type=102&status=OK&devices=light,status=off,brightness=100,color=255,255,255|lock,status=unlocked|alarm,status=disarmed,code=1234

[SERVER] ---- Connection Closed ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53493)

[SERVER] ---- New Connection ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53494)

[SERVER] ---- Received Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.CHG_STATUS: 103>, 'device': 'light', 'status': 'on'}
Marshaled: type=103&device=light&status=on

[SERVER] ---- Sending Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.CHG_STATUS: 103>, 'status': 'light updated successfully'}
Marshaled: type=103&status=light updated successfully

[SERVER] ---- Connection Closed ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53494)

[SERVER] ---- New Connection ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53495)

[SERVER] ---- Received Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.CHG_STATUS: 103>, 'device': 'light', 'brightness': '75', 'color': '255,0,0'}
Marshaled: type=103&device=light&brightness=75&color=255,0,0

[SERVER] ---- Sending Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.CHG_STATUS: 103>, 'status': 'light updated successfully'}
Marshaled: type=103&status=light updated successfully

[SERVER] ---- Connection Closed ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53495)

[SERVER] ---- New Connection ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53496)

[SERVER] ---- Received Request ----
Time: 2025-02-28 15:14:22
Raw Message Data: {'type': <REQS.CHG_STATUS: 103>, 'device': 'lock', 'status': 'locked'}
Marshaled: type=103&device=lock&status=locked

[SERVER] ---- Sending Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.CHG_STATUS: 103>, 'status': 'lock updated successfully'}
Marshaled: type=103&status=lock updated successfully

[SERVER] ---- Connection Closed ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53496)

[SERVER] ---- New Connection ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53497)

[SERVER] ---- Received Request ----
Time: 2025-02-28 15:14:22

Raw Message Data: {'type': <REQS.LOUT: 101>}
Marshaled: type=101

[SERVER] ---- Sending Response ----
Time: 2025-02-28 15:14:22
Raw Response Data: {'type': <REQS.LOUT: 101>, 'status': 'Logged out successfully'}
Marshaled: type=101&status=Logged out successfully

[SERVER] ---- Connection Closed ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53497)

[SERVER] ---- New Connection ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53498)

[SERVER] ---- Error ----
Time: 2025-02-28 15:14:22
Message: Error receiving message

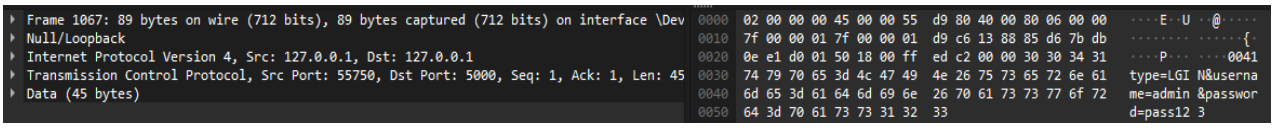
[SERVER] ---- Connection Closed ----
Time: 2025-02-28 15:14:22
Client Address: ('127.0.0.1', 53498)

Operations – P3

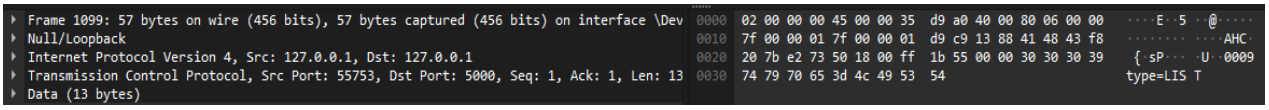
Below are some of the packets transferred between client and server:

Client Ops – P3

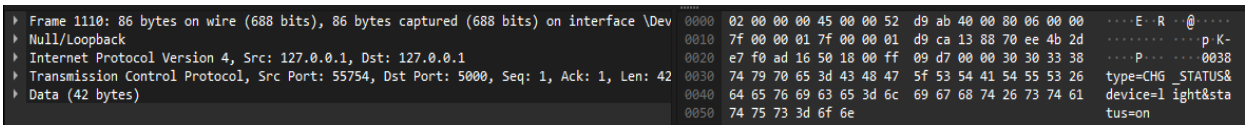
Login Req



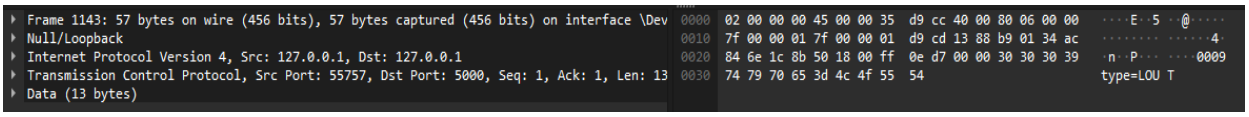
List Req



Change Status Req:

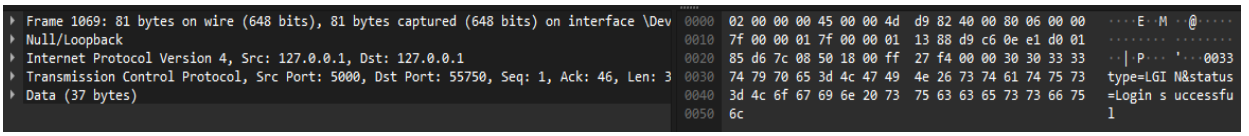


Logout Req:

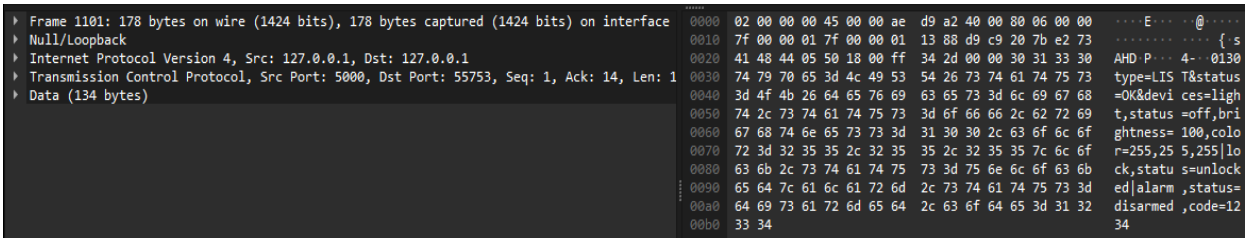


Middle-Ware Server Ops – P3

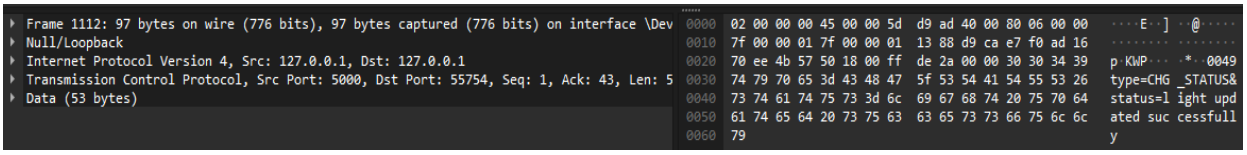
Login Resp:



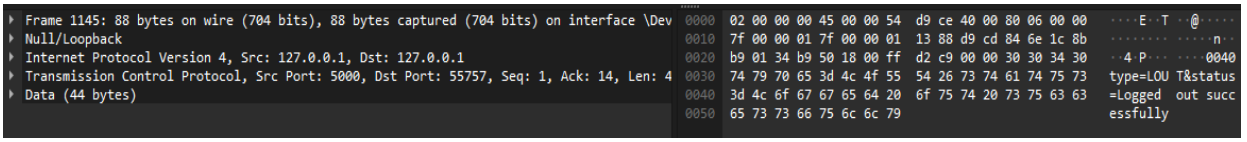
List Resp:



Change Status Resp:



L Out Resp:



Applications – P4

<Testing details of the application>

Client Application – P4

<Testing of the Client Application>

Middle-Ware Server – P4

Conclusion – P2, P3, P4

The design and implementation of the Smart Home Remote Access System involved several key decisions to ensure a robust and scalable architecture. A critical part of this process was defining the data model and communication protocol between the client and server. This required careful consideration of how device states would be stored, how authentication would be managed, and how different types of client requests would be processed efficiently.
Throughout development, several challenges arose, particularly in designing the data model and drawing the necessary system architecture diagrams. Structuring the relationships between different components while ensuring clarity in the diagrams proved to be more difficult than expected. Capturing the exact flow of requests, responses, and authentication states in a way that was both accurate and easy to interpret required multiple iterations.
One of the most persistent technical issues was ensuring that requests were correctly authenticated and handled by the right user. This was especially evident in the early implementation phases when we initially tracked logged-in users based on their connection addresses. This approach proved unreliable due to the transient nature of client-server connections. The issue was eventually mitigated by simplifying the model to assume a single active user at a time, which allowed for a more straightforward validation process.
Another major challenge was defining the edge cases for each request type. Handling invalid or unexpected inputs, such as modifying a non-existent device or trying to execute actions while unauthorized, required significant debugging and refinement. Ensuring proper error handling and response messages was an iterative process that revealed gaps in our initial logic.
Additionally, testing with Wireshark presented another layer of difficulty. Initially, no packets were being captured due to incorrect filter configurations and potential local loopback interface restrictions. Adjusting the capture settings and verifying that the traffic was properly routed helped resolve this, but it underscored the difficulty of real-time packet inspection for debugging. Additionally, testing with Wireshark presented another layer of difficulty. Initially, no packets were being captured due to incorrect filter configurations and potential local loopback interface restrictions. Adjusting the capture settings and verifying that the traffic was properly routed helped resolve this, but it underscored the difficulty of real-time packet inspection for debugging.

Appendix – P4

* code was separated by folders and files for better project structure

- **messaging/**
  - **csmessage.py:**

```
from enum import Enum
class REQS(Enum):

  TURN_ON = 200
  TURN_OFF = 201
  LOCK = 202
  UNLOCK = 203
  CHECK_STATUS = 204
class CSmessage:

  PJOIN = '&'
  VJOIN = '{}={}'
  VJOIN1 = '='
  def __init__(self):

    self._data = {}
    self._data['type'] = REQS.CHECK_STATUS
  def setType(self, t):

    self._data['type'] = t
  def getType(self):

    return self._data['type']
  def addValue(self, key, value):

    self._data[key] = value
  def getValue(self, key):

    return self._data.get(key)
```

```
    def marshal(self):

      pairs = [CSmessage.VJOIN.format(k, v) for (k, v) in self._data.items()]
      return CSmessage.PJOIN.join(pairs)
    def unmarshal(self, data):

      self._data = {}
      if data:
        params = data.split(CSmessage.PJOIN)
        for p in params:
          k, v = p.split(CSmessage.VJOIN1)
          self._data[k] = v

```

- **cspdu.py:**

```
import socket
from messaging.csmessage import CSmessage
 class SmartHomePDU:

    """
    Handles sending and receiving messages over a TCP connection for the Smart Home system.
    """
     def __init__(self, comm: socket):

      """
      Initializes the PDU with a socket connection.
      """
      self._sock = comm
     def _loop_recv(self, size: int):

      """
      Ensures we receive exactly `size` bytes of data.
      """
      data = bytearray(b" " * size)
      mv = memoryview(data)
```

```
            while size:
                rsize = self._sock.recv_into(mv, size)
                mv = mv[rsize:]
                size -= rsize
            return data
        def send_message(self, message: CSmessage):

            """
            Marshals and sends a message over the socket.
            """
            mdata = message.marshal()
            size = len(mdata)
            sdata = '{:04}{}'.format(size, mdata)
            self._sock.sendall(sdata.encode('utf-8'))
        def receive_message(self) -> CSmessage:

            """
            Receives a message, unmarshals it, and returns a SmartHomeMessage object.
            """
            try:
                m = CSmessage()
                size = int(self._loop_recv(4).decode('utf-8'))  # Read the first 4 bytes (size)
                params = self._loop_recv(size).decode('utf-8')  # Read the message body
                m.unmarshal(params)
            except Exception:
                raise Exception('Error receiving message')
            else:
                return m
        def close(self):

            """
            Closes the socket connection.
            """
            self._sock.close()

```

- **models/** ○

alarm.py:

```
from models.device import Device

 class Alarm(Device):


   def __init__(self, device_id, room_id, home_id, pin_code: str):
     super().__init__(device_id, "Alarm", room_id, home_id)
     self.is_armed = False
     self.pin_code = pin_code
      def
arm(self):
     self.is_armed = True
      def disarm(self, pin:
str):
     if pin == self.pin_code:
       self.is_armed = False
       return True
     return False
```

○ **device.py:**

```
class Device:
   def __init__(self, device_id: str, device_type: str, room_id: str, home_id: str):
     self.device_id = device_id
     self.device_type = device_type
     self.status = "off"
     self.room_id = room_id
     self.home_id = home_id
      def
get_status(self):
     return self.status
```

- **home.py:**

```
from models.room import Room

class Home:


    def __init__(self, home_id: str, address: str, owner):
        self.home_id = home_id
        self.address = address
        self.owner = owner
        self.rooms = []

    def list_rooms(self):
        return self.rooms
    def list_devices(self):
        devices = []
        for room in self.rooms:
            devices.extend(room.devices)
        return devices
```

- **light.py:**

```
from models.device import Device

class Light(Device):


    def __init__(self, device_id, room_id, home_id, brightness=100, color="white"):
        super().__init__(device_id, "Light", room_id, home_id)
        self.brightness = brightness
        self.color = color
    def turn_on(self):
        self.status = "on"
    def turn_off(self):
        self.status = "off"
    def dim(self, level: int):

```

```
        self.brightness = level
    def change_color(self, color: str):

        self.color = color
```

- **lock.py:**

```
from models.device import Device
class Lock(Device):

    def __init__(self, device_id, room_id, home_id, pin_codes):
        super().__init__(device_id, "Lock", room_id, home_id)
        self.is_locked = True
        self.pin_codes = pin_codes
    def lock(self):

        self.is_locked = True
    def unlock(self, pin: str):

        if pin in self.pin_codes:
            self.is_locked = False
            return True
        return False
```

- **room.py:**

```
from models.device import Device
class Room:

    def __init__(self, room_id: str, name: str, home_id: str):
        self.room_id = room_id
        self.name = name
        self.home_id = home_id
        self.devices = []
    def list_devices(self):

        return self.devices
```

- **user.py:**

```
class User:
    def __init__(self, username: str, password: str):
        self.username = username
        self.password = password
        def authenticate(self, username: str, password: str) -> bool:

        return self.username == username and self.password == password
```

- **window_blind.py:**

```
from models.device import Device
class WindowBlind(Device):

    def __init__(self, device_id, room_id, home_id):
        super().__init__(device_id, "WindowBlind", room_id, home_id)
        self.is_up = True
        self.is_open = True
        def raise_blinds(self):
        self.is_up = True
        def lower_blinds(self):
        self.is_up = False
        def open_blinds(self):
        self.is_open = True
        def close_blinds(self):
        self.is_open = False
```

- **tests/**

- **extended_tester.py:**

```
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from models.home import Home
from models.room import Room
from models.light import Light
from models.window_blind import WindowBlind
from models.lock import Lock
from models.alarm import Alarm
from models.user import User

from messaging.csmessage import CSmessage, REQS
from messaging.cspdu import SmartHomePDU

# ====================
# SETUP TEST ENVIRONMENT
# ====================
user = User("admin", "pass123")
home = Home("home1", "123 Main St", user)

# Rooms
living_room = Room("room1", "Living Room", home.home_id)
bedroom = Room("room2", "Bedroom", home.home_id)
kitchen = Room("room3", "Kitchen", home.home_id)

# Devices
light_lr = Light("light_lr", living_room.room_id, home.home_id)
blind_lr = WindowBlind("blind_lr", living_room.room_id, home.home_id)
lock_lr = Lock("lock_lr", living_room.room_id, home.home_id, ["1234", "5678"])
alarm_bd = Alarm("alarm_bd", bedroom.room_id, home.home_id, "4321")
light_kt = Light("light_kt", kitchen.room_id, home.home_id)
```

```

# Add devices
living_room.devices.extend([light_lr, blind_lr, lock_lr])
bedroom.devices.append(alarm_bd)
kitchen.devices.append(light_kt)
home.rooms.extend([living_room, bedroom, kitchen])


# ====================
# TEST CASES
# ====================
def print_status():

    print("\n===== FINAL DEVICE STATES =====")
```

```
    for room in home.list_rooms():
        print(f"Room: {room.name}")
        for device in room.list_devices():
            print(f" - {device.device_type} ({device.device_id}): {device.get_status()}")
def test_message(message):

    print("\n--- Test Message ---")
    print(f"Marshaled: {message.marshal()}")
    received = CSmessage()
    received.unmarshal(message.marshal())
    print(f"Unmarshaled: {received._data}")
    return received


# Normal case: Turn on kitchen light
msg1 = CSmessage()
msg1.setType(REQS.TURN_ON)
msg1.addValue("device", "light")
msg1.addValue("room", "Kitchen")
received = test_message(msg1)
if received.getValue("room") == "Kitchen":
    light_kt.turn_on()
```

```
    print(f"[Server Response] Kitchen light status: {light_kt.get_status()}")

# Normal case: Unlock Living Room door with correct PIN
msg2 = CSmessage()
msg2.setType(REQS.UNLOCK)
msg2.addValue("device", "lock")
msg2.addValue("room", "Living Room")
msg2.addValue("pin", "1234")
received = test_message(msg2)
if lock_lr.unlock(received.getValue("pin")):
    print(f"[Server Response] Living Room lock: Unlocked")
else:
    print(f"[Server Response] Living Room lock: Locked (Invalid PIN)")

# Edge case: Invalid PIN for Living Room lock
msg3 = CSmessage()
msg3.setType(REQS.UNLOCK)
msg3.addValue("device", "lock")
msg3.addValue("room", "Living Room")
msg3.addValue("pin", "0000")
received = test_message(msg3)
if not lock_lr.unlock(received.getValue("pin")):
    print(f"[Server Response] Lock status: Locked (Invalid PIN)")

# Edge case: Attempt to disarm Bedroom alarm with wrong PIN
msg4 = CSmessage()
msg4.setType(REQS.UNLOCK)
msg4.addValue("device", "alarm")
msg4.addValue("room", "Bedroom")
msg4.addValue("pin", "9999")
received = test_message(msg4)
if not alarm_bd.disarm(received.getValue("pin")):
    print(f"[Server Response] Alarm status: Armed (Invalid PIN)")
```

```
    # Edge case: Invalid device type
    msg5 = CSmessage()
    msg5.setType(REQS.TURN_ON)
    msg5.addValue("device", "fan")  # Invalid device
    msg5.addValue("room", "Living Room")
    received = test_message(msg5)
    if received.getValue("device") not in ["light", "lock", "alarm", "windowblind"]:
        print(f"[Server Response] Error: Invalid device type '{received.getValue('device')}'")

    # Edge case: Empty message without using reset()
    msg6 = CSmessage()  # No values added, simulating an empty message
    received = test_message(msg6)
    if len(received._data) <= 1:
        print("[Server Response] Error: Empty message received.")

    # Edge case: Malformed message (missing room key)
    msg7 = CSmessage()
    msg7.setType(REQS.TURN_OFF)
    msg7.addValue("device", "light")  # No 'room' key
    received = test_message(msg7)
    if "room" not in received._data:
        print("[Server Response] Error: Missing 'room' parameter in message.")

    # Normal case: Lower Living Room blinds
    msg8 = CSmessage()
    msg8.setType(REQS.TURN_OFF)
    msg8.addValue("device", "windowblind")
    msg8.addValue("room", "Living Room")
    received = test_message(msg8)
    if received.getValue("device") == "windowblind" and received.getValue("room") == "Living Room":
        blind_lr.lower_blinds()
        print(f"[Server Response] Blinds in Living Room: {'Up' if blind_lr.is_up else 'Down'}")

    #  Normal case: Disarm Bedroom alarm with correct PIN
```

```
msg9 = CSmessage()
msg9.setType(REQS.UNLOCK)
msg9.addValue("device", "alarm")
msg9.addValue("room", "Bedroom")
msg9.addValue("pin", "4321")
received = test_message(msg9)
if alarm_bd.disarm(received.getValue("pin")):
    print(f"[Server Response] Alarm in Bedroom: Disarmed")


# ====================
# FINAL DEVICE STATES
# ====================
print_status()

```

- **networking:**
  - **client.py**

```
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

import socket
from messaging.csmessage import CSmessage, REQS
from messaging.cspdu import SmartHomePDU as CSpdu
import time

HOST = "127.0.0.1"
PORT = 5000

def send_request(request_type, values={}):
    """Send a request to the server and print detailed response logs."""
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client:
        client.connect((HOST, PORT))
        pdu = CSpdu(client)

        message = CSmessage()
        message.setType(request_type)

        for key, value in values.items():
            message.addValue(key, value)
```

```
        print(f"\n[CLIENT] ---- Sending Request ----")
        print(f"Time: {time.strftime('%Y-%m-%d %H:%M:%S')}")
        print(f"Raw Message Data: {message._data}")
        print(f"Marshaled: {message.marshal()}\n")

        pdu.send_message(message)

        response = pdu.receive_message()
        print(f"[CLIENT] ---- Received Response ----")
        print(f"Time: {time.strftime('%Y-%m-%d %H:%M:%S')}")
        print(f"Raw Response Data: {response._data}")
        print(f"Marshaled: {response.marshal()}\n")

if __name__ == "__main__":
    print("\n===== Running Full Test Suite =====")

    print("\n[TEST] Successful Login")
    send_request(REQS.LGIN, {"username": "admin", "password": "pass123"})

    print("\n[TEST] Unsuccessful Login (Invalid Password)")
    send_request(REQS.LGIN, {"username": "admin", "password": "wrongpass"})

    print("\n[TEST] Login Attempt When Already Logged In")
    send_request(REQS.LGIN, {"username": "admin", "password": "pass123"})

    print("\n[TEST] Listing Devices")
    send_request(REQS.LIST)

    print("\n[TEST] Changing Device Status (Turning Light ON)")
    send_request(REQS.CHG_STATUS, {"device": "light", "status": "on"})

    print("\n[TEST] Setting Device Attributes (Brightness and Color)")
    send_request(REQS.CHG_STATUS, {"device": "light", "brightness": "75", "color": "255,0,0"})

    print("\n[TEST] Attempting to Change Status Without Logging In")
    send_request(REQS.CHG_STATUS, {"device": "lock", "status": "locked"})

    print("\n[TEST] Logout")
    send_request(REQS.LOUT)

    print("\n[TEST] Sending an Unknown Request Type")
    send_request(999, {"param": "unknown"})

    print("\n[TEST] Sending a Request With Missing Parameters")
    send_request(REQS.CHG_STATUS, {})
```

```
    print("\n[TEST] Handling Large Message Payloads")
    send_request(REQS.CHG_STATUS, {"device": "light", "brightness": "100", "color": "255,255,255", "extra_data": "x" * 500})

    print("\n===== Full Test Suite Completed =====")

```

- **server.py**

```
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

import socket
from datetime import datetime
from messaging.csmessage import CSmessage, REQS
from messaging.cspdu import SmartHomePDU as CSpdu

# Track login status and device states
logged_in = False
devices = {
    "light": {"status": "off", "brightness": 100, "color": "255,255,255"},
    "lock": {"status": "unlocked"},
    "alarm": {"status": "disarmed", "code": "1234"}
}

def log_message(event, details):
    """Logs server events with timestamps."""
    print(f"\n[SERVER] ---- {event} ----")
    print(f"Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    for key, value in details.items():
        print(f"{key}: {value}")

def handle_client(conn, addr):
    """Handles client requests and processes responses."""
    global logged_in
    pdu = CSpdu(conn)

    try:
        log_message("New Connection", {"Client Address": addr})

        message = pdu.receive_message()
        log_message("Received Request", {"Raw Message Data": message._data, "Marshaled": message.marshal()})

        response = CSmessage()
```

```
        response.setType(message.getType())

        if message.getType() == REQS.LGIN:
            username = message.getValue("username")
            password = message.getValue("password")

            if logged_in:
                response.addValue("status", "Already logged in")
            elif username == "admin" and password == "pass123":
                logged_in = True
                response.addValue("status", "Login successful")
            else:
                response.addValue("status", "Invalid credentials")

        elif message.getType() == REQS.LOUT:
            if logged_in:
                logged_in = False
                response.addValue("status", "Logged out successfully")
            else:
                response.addValue("status", "User not logged in")

        elif message.getType() == REQS.LIST:
            if not logged_in:
                response.addValue("status", "Unauthorized request")
            else:
                device_list = "|".join([f"{name},{','.join(f'{k}={v}' for k,v in
info.items())}" for name, info in devices.items()])
                response.addValue("status", "OK")
                response.addValue("devices", device_list)

        elif message.getType() == REQS.CHG_STATUS:
            if not logged_in:
                response.addValue("status", "Unauthorized request")
            else:
                device_name = message.getValue("device")
                if device_name in devices:
                    for key, value in message._data.items():
                        if key not in ["type", "device"]:
                            devices[device_name][key] = value
                    response.addValue("status", f"{device_name} updated successfully")
                else:
                    response.addValue("status", "Invalid device")

        else:
            response.addValue("status", "Unknown request type")
```

```
        log_message("Sending Response", {"Raw Response Data": response._data, "Marshaled":
response.marshal()})
        pdu.send_message(response)

    except Exception as e:
        log_message("Error", {"Message": str(e)})

    finally:
        log_message("Connection Closed", {"Client Address": addr})
        conn.close()


def start_server():
    """Starts the Smart Home Remote Access Server."""
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(("127.0.0.1", 5000))
    server_socket.listen(5)
    print("[SERVER] Server started on 127.0.0.1:5000")

    while True:
        conn, addr = server_socket.accept()
        handle_client(conn, addr)

if __name__ == "__main__":
    start_server()
```