ChatGPT

Smart Door Lock System Prototype Artefact Report

Project Overview

Smart door lock systems are an increasingly popular component of modern smart homes, offering keyless entry and remote access control for enhanced convenience and security [1]. These systems allow homeowners to monitor and control door locks via smartphones or connected devices, and even to grant temporary digital keys to guests or service personnel [1]. There are many commercial smart locks available today; for example, the August Wi-Fi Smart Lock is a well-known solution that retrofits existing deadbolts and supports remote smartphone access while still allowing use of a physical key [2]. Other products like the Schlage Encode and Yale Assure provide keypad entry and integration with voice assistants, and high-end models such as the Lockly Vision or Eufy Video Smart Lock incorporate built-in cameras or facial recognition for additional verification [3]. Despite these advances, commercial solutions can be expensive and may not offer the exact custom features desired by a user (for instance, some lack real-time photo verification by a human). This prototype project was conceived to explore a DIY approach to a smart door lock, focusing on remote guest access approval using affordable components and custom logic.

Problem Addressed: The prototype addresses the scenario of securely granting entry to a guest when the homeowner (primary user) is not at the door. Traditional door locks require the owner's physical presence or sharing a physical key, which may be inconvenient or risky. Smart video doorbells partially solve this by letting homeowners see who's at the door, but not all are integrated with door locks for immediate action. The problem tackled here is enabling a homeowner to remotely **verify a visitor's identity via a photo and then unlock the door in real-time** if the visitor is approved. This improves security (since entry is only granted after positive identification) and convenience (the homeowner can make the decision from anywhere, or even from inside the home without going to the door). It also provides a solution for situations like family members arriving without keys, or granting one-time access to delivery personnel, without permanently sharing keys or PIN codes.

Key Functional Requirements: - The system must capture an image of the visitor (guest) at the door and send it to the homeowner for review. - The primary user (homeowner) must be able to remotely view the guest's photo via a graphical interface and send an **Approve** or **Decline** decision. - Upon receiving an approval, the door lock should automatically unlock (via a servo motor mechanism) to allow entry. If declined (or no response in time), the door remains locked. - The guest-side unit should provide feedback to the visitor (e.g. a buzzer sound or LED indication) to inform them if access was granted or denied, or if they should wait. - The system should distinguish between a **primary user** (authorized homeowner) and a guest. If the person at the door is actually the homeowner (primary), the system should allow them to authenticate locally (e.g. via a password/PIN or known method) to unlock immediately without needing remote approval. - All communication between the door unit and the reviewer unit should be near real-time to avoid long wait times for guests.

Key Non-Functional Requirements: - **Security:** The communication (and the photo being sent) should be transmitted reliably and, in a real deployment, securely (e.g. within a closed network or using encryption) to prevent eavesdropping. The system should ensure that only the authorized primary user's decision can unlock the door. - **Usability:** The interface for the homeowner should be simple (one-click Approve/Decline), and the process for the guest should be straightforward (e.g. press a doorbell button and wait for response). The system should be intuitive with minimal training. - **Reliability and Fault Tolerance:** The door should remain locked by default and only unlock on explicit approval. If network communication fails or no response is received within a certain timeout, the system should default to denying access (fail-safe). Components like the servo and camera should be handled carefully to avoid crashes (e.g. handle camera errors or MQTT disconnections gracefully). - **Performance:** The time from guest request to owner decision and door unlock should be reasonably short (ideally only a few seconds, depending on network speeds) to avoid keeping the visitor waiting too long. Capturing and transmitting the image should be as efficient as possible (e.g. using an optimized image size/format). - **Scalability and Extendability:** While this is a single-door prototype, the design should allow adding more doors or more reviewer clients in the future. The use of MQTT and modular components helps in extending the system to multiple nodes easily. - **Privacy:** Photos of guests should not be stored long-term on the devices (unless needed for logs) to respect privacy; in the prototype, images are used transiently for verification and not saved persistently.

In summary, the project's goal is to demonstrate a functional **smart door lock prototype** that uses two Raspberry Pi devices to enable remote-controlled entry for guests. It draws inspiration from commercial smart locks and video doorbells but is built from scratch using open-source hardware and software. The following sections detail the design, architecture, implementation, testing, and usage of the system.

Design Principles

This prototype was developed with several core design principles in mind, ensuring the system is effective and user-friendly:

- **Simplicity & Usability:** Both the guest experience and the primary user experience are kept as simple as possible. On the guest side, the only actions required are to initiate a request (e.g. by pressing a doorbell button or otherwise triggering the system) and then wait for feedback. On the primary user side, a desktop GUI (using Tkinter) cleanly presents the visitor's photo and two decision buttons: **Approve** or **Decline**. This one-click decision interface makes it straightforward for the homeowner to use, even if they are not tech-savvy. Visual and audio cues are used for clarity – for instance, a buzzer on the door unit provides audible feedback to the guest (a pleasant tone for approved, a different tone or pattern for declined). The overall workflow mimics a familiar doorbell/intercom interaction, augmented with smart verification.

- **Clear Separation of Responsibilities:** The system is split into two decoupled components, each running on a separate Raspberry Pi, to enforce a separation of concerns. **Raspberry Pi 1 (Door Unit)** is dedicated to hardware interfacing and initial data capture – it handles the camera, door lock servo motor, buzzer, and any door-side input (e.g. a button or sensor to detect a guest). It also runs the logic to publish the guest's photo and listen for the response. **Raspberry Pi 2 (Reviewer Unit)** is focused on the user interface and decision making logic – it subscribes to incoming photos, displays the image in a Tkinter GUI, and captures the homeowner's approval or denial, then publishes the response. This separation not only simplifies the software design (each program has a single clear role) but also improves reliability: a failure in the UI or review device does not directly crash the door

hardware device, and vice versa. It also mirrors real-world deployments where the door lock and the user's interface might be physically separate devices. The two units communicate only via MQTT messages, following a publish/subscribe pattern, which decouples them spatially and in time (they do not directly call each other's functions, they just react to messages).

- **Use of MQTT for Communication:** We chose MQTT as the messaging protocol because it is lightweight and well-suited for IoT devices and intermittent networks [4]. MQTT follows a publish/subscribe architecture that **decouples** the sender and receiver – both Raspberry Pis simply need to know the address of the MQTT broker, not each other's IP or details [5] [4]. This design makes the system more robust and scalable; for example, additional reviewer clients could be added subscribing to the same topic without modifying the door unit's code. MQTT also provides features like Quality of Service (QoS) levels which can ensure message delivery, and retains messages if configured (though in this prototype we primarily rely on real-time messaging without persistence). The broker (e.g. Mosquitto) mediates all messages and can be secured with authentication or encryption if needed. Overall, using MQTT greatly simplified the communication logic and reduced the need for complex networking code.

- **Fault Tolerance & Feedback:** From the start, we designed the system to handle error conditions and provide user feedback for robustness. One aspect is **timeout handling** – after the door unit publishes a guest photo and requests approval, it starts a timer. If no response (approve/decline) arrives within, say, 30 seconds (configurable), the system assumes no one is available to approve and triggers a timeout routine. In a timeout scenario, the door remains locked and the guest is informed (for example, a buzzer might emit a long **beep** pattern to indicate no response). This prevents the guest from waiting indefinitely and defaults to a secure state (no entry). Another aspect is the use of the buzzer and possibly an LED to signal system status: a short beep could acknowledge that a request was sent and is **pending**, a continuous tone or a specific melody indicates **unlocked/approved**, and a different tone indicates **denied**. These cues ensure the guest knows what is happening (important if the door doesn't automatically speak or if the guest can't see the homeowner). The design also accounts for hardware or network failures: if the camera is unavailable or fails to capture an image, the system catches that error and can retry or inform the user. If the network/MQTT broker is down, the door unit similarly will inform the guest (perhaps via a different error buzzer code) and log the issue. The servo motor control includes safeguards as well – for example, ensuring the servo is not continuously powered (to avoid overheating) and moving it only for the duration needed to unlock, then returning to lock position. The code is structured to handle exceptions so that a failure in one component (like camera capture) won't crash the entire program.

- **Design Constraints Consideration:** The prototype runs on Raspberry Pi hardware which has certain constraints. We deliberately kept processing simple – rather than performing heavy image processing on the Pi (like face recognition), the system simply captures and relays a photo. This keeps CPU usage low and avoids latency. The camera capture uses the Raspberry Pi's **libcamera** interface (via the `libcamera-still` tool) to take a still image of the visitor [6]. We found that `libcamera-still` has an inherent capture delay (it may take around 1-2 seconds to initialize and snap a photo), which informed our timing design (the guest is prompted to hold still for a moment when requesting access). The size of the image is also constrained; to keep MQTT message size reasonable, the image is captured at a resolution sufficient for identification but not excessively high. It's encoded to a JPEG and then base64 string. MQTT can handle binary payloads, but using base64 text made it simpler to send via the Python MQTT client. We are mindful of network reliability

– MQTT is lightweight, but sending images even as base64 can be on the order of a few hundred kilobytes, which on a poor Wi-Fi network might be slow. In testing, we tuned the image size to balance clarity with speed. Another constraint was memory and GUI performance on the Raspberry Pi: the Tkinter GUI on Pi 2 must efficiently handle image display and user input without freezing. We designed the GUI to update in response to MQTT messages using a non-blocking approach (the MQTT client runs in a background thread or uses the loop that doesn't block Tkinter's mainloop), ensuring the window remains responsive. Additionally, the prototype is built with off-the-shelf, low-cost components (Pi boards, a standard servo, etc.), so the design had to accommodate their limitations (for example, the servo torque may be low, so the mechanical lock mechanism should require minimal force – a simple latch that the servo can slide open).

In summary, the design of this smart door lock prototype emphasizes a **modular, robust, and user-centered approach**. Simplicity in interaction, clear division of tasks, reliable messaging, and handling of edge cases were all prioritized to ensure the system works smoothly in a real-world-like scenario despite using basic hardware.

Prototype Architecture

The system's architecture consists of two Raspberry Pi units and an MQTT broker facilitating communication between them. The diagram below illustrates the components and message flow between them:

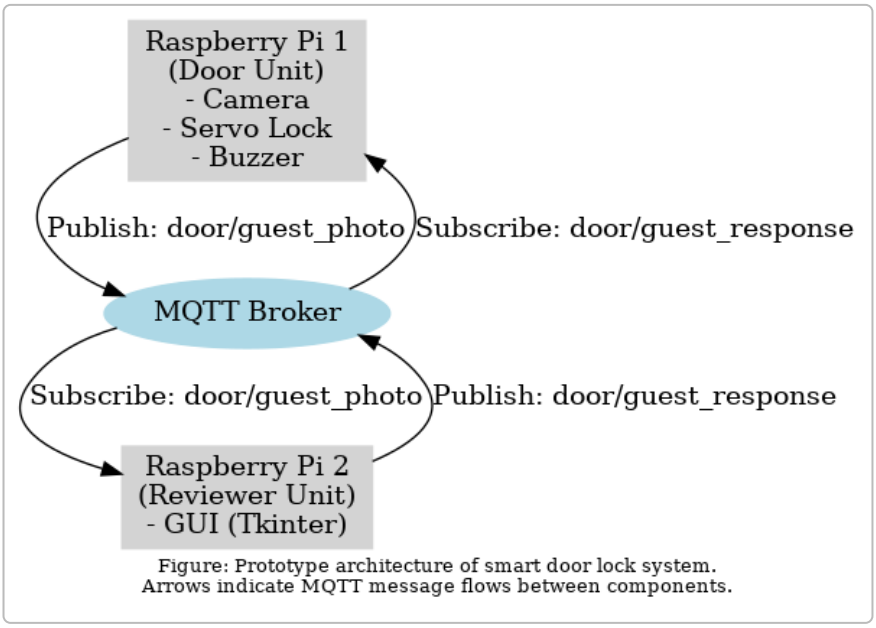

Figure: Prototype architecture of the smart door lock system. Raspberry Pi 1 acts as the Door Unit (with camera, servo lock, and buzzer), and Raspberry Pi 2 serves as the Reviewer Unit with a Tkinter GUI. Arrows indicate MQTT publish/subscribe message flows between components, via the MQTT broker.

System Components:

- **Raspberry Pi 1 – Door Unit (Guest-side):** This device is physically installed at the door. It interfaces with the hardware:

- A Raspberry Pi Camera is attached to capture an image of the person at the door.
- A Servo Motor is connected to the door's locking mechanism. In our prototype, a small servo (such as an SG90) is mounted such that it can move the lock bolt or a latch open when activated. This servo is controlled via a GPIO PWM signal.
- A Buzzer (or speaker) is connected to another GPIO pin to provide audible feedback tones.
- (Optionally, a doorbell **button or sensor** can be connected to trigger the system when a guest arrives. In testing, simply running the program could initiate the capture sequence, but a physical button is a practical addition.)

- The Pi runs a Python script which implements the following logic:

1. **Guest Detection/Authentication:** When someone arrives, the system determines if they are the homeowner or a guest. For example, the person could press a button indicating they're a guest requesting entry. If the person is the homeowner and knows a pre-set *safety PIN/password*, they could input it (via a keypad or the program interface) to authenticate locally. The prototype can check a password (e.g., a hard-coded PIN in software) – if it matches, the system will immediately unlock the door for the primary user without needing remote approval. This serves as a fallback for the homeowner and for system resilience (for instance, if network is down, the homeowner can still get in with the code).
2. **Capture Photo:** For a guest (or any unrecognized person), the Pi activates the camera to take a photo of the visitor. We utilized the `libcamera-still` command-line tool of Raspberry Pi to capture a still image from the camera [6] . The image is saved temporarily (e.g., as a JPEG file) and then read and encoded to a base64 string in Python.
3. **Publish Photo via MQTT:** The base64-encoded image (essentially a text string representing the photo) is published to the MQTT broker on the topic `door/guest_photo`. The MQTT broker in our setup can be a Mosquitto server running on the local network – it could even run on one of the Raspberry Pis (to reduce infrastructure), though running it on a separate device or a cloud server is also possible. In our tests, we often ran Mosquitto on Raspberry Pi 1 itself, and Raspberry Pi 2 connected to it over the LAN.
4. **Await Response:** After publishing the photo, Raspberry Pi 1 subscribes (or was already subscribed) to the `door/guest_response` topic and waits. Meanwhile, it might emit a short "waiting" beep via the buzzer to let the guest know the system is processing. The program sets a timeout (e.g. 30 seconds). If a message arrives on `door/guest_response`, it proceeds to the next step. If the timeout elapses first with no message, it will treat it as a failure to get approval.
5. **Process Response:** The message on `door/guest_response` will be a simple text, either `"APPROVED"` or `"DECLINED"` (we chose these keywords for clarity). If the content is `"APPROVED"`, the door unit triggers the servo motor to turn to the unlock position. This may involve, for example, setting the servo to rotate 90 degrees (the angle calibrated to disengage the lock). The servo remains in that position for a short duration (allowing the door to be opened) and then can return to the locked position. During an unlock, the buzzer can sound a friendly tone (or a series of beeps) to indicate success. If the response is `"DECLINED"` (or if a timeout occurred), the system ensures the door stays locked (servo does nothing or returns to lock position if it was moved) and the buzzer might output a different tone pattern to signal the denial. The outcome is also logged in the console or a log file for record-keeping (e.g., "Guest request at 5:30 PM – Approved" or "Declined/No response").

6. **Reset State:** Finally, the door unit returns to an idle state, ready for the next guest or input. If another visitor comes or if the same guest tries again, the cycle repeats. The system could also have a cool-off to prevent spamming (e.g., don't capture a new photo more than once every few seconds).

- **Raspberry Pi 2 – Reviewer Unit (Primary user side):** This device can be located anywhere the primary user is – e.g., inside the house as a dedicated screen, or carried as a tablet/portable unit, or theoretically it could be replaced by a laptop or smartphone running appropriate software. For the prototype, we used a second Raspberry Pi running a Tkinter-based GUI application. The responsibilities of this unit include:

- **Subscribe to Photo Topic:** The reviewer app subscribes to the MQTT topic `door/guest_photo`. It runs an MQTT client loop in the background so that whenever a new message (photo) is published by the door unit, it receives it immediately.
- **Display Guest Image:** When a photo message arrives, the payload (the base64 string) is decoded back into image data (JPEG). The Tkinter GUI then updates to display this image on the screen. We designed the GUI window to have an area for the image and two big buttons ("Approve" and "Decline"). When no image is available (idle state), it might show a placeholder or remain blank.
- **User Decision Input:** The homeowner reviews the visitor's image on the screen. They can then click **Approve** if they recognize the person and want to grant access, or **Decline** if not. These buttons in the GUI are wired to callback functions in the code.
- **Publish Response via MQTT:** Based on the button click, the reviewer unit publishes a message on the `door/guest_response` topic. For instance, clicking Approve sends the string "APPROVED" to `door/guest_response`, and Decline sends "DECLINED" to that topic. This publish happens quickly and the door unit should receive it almost instantly (MQTT is designed for low-latency message delivery).
- **GUI Feedback:** The GUI can optionally show a confirmation or status (like "Last decision: Approved at 5:30 PM") so the homeowner knows the action was sent. It might also clear the image after a decision or after some time, to be ready for the next request.

- **Idle State:** After the decision, the reviewer app goes back to waiting for the next `guest_photo`. If multiple guests come in sequence, the process repeats. We also considered that the homeowner might not always be watching the screen, so it could play a sound or pop up a notification when a new guest photo arrives (to alert the user). Those features could be added for practicality.

- **MQTT Broker:** The broker is a central component but often lightweight. We used the open-source Mosquitto broker, configured without authentication on a local network for simplicity during development. The broker can run on a Raspberry Pi or any server. In our architecture diagram we show it as a separate entity mediating between Pi 1 and Pi 2. The publish/subscribe flow works as follows:

- Pi 1 publishes the guest photo to topic `door/guest_photo`. The broker receives it and immediately routes it to any clients subscribed to that topic (in our case, Pi 2).
- Pi 2, upon user action, publishes to `door/guest_response`. The broker then forwards that message to any subscribers of `door/guest_response` (in our case, Pi 1).
- This decoupling means Pi 1 and Pi 2 do not directly connect to each other – they only connect to the broker. This is beneficial for modularity and allowed us to run the two devices on different networks

during some tests (with a cloud MQTT broker), and it would easily allow scaling to multiple door units or multiple authorized reviewers if needed, just by adjusting topics and subscriptions.
- MQTT uses a TCP connection; both Pis maintain a live connection to the broker, so messages are near-instant. We set the MQTT quality of service to at least "QoS 1" (at least once delivery) for the critical messages, to ensure they arrive. The photo message is larger, so QoS 0 (at most once) was used to avoid potential delays, but since it's immediately followed by a response, any loss would be noticed (this could be tuned as needed).

Message Format and Flow Summary: In summary, the sequence for a guest visit is: 1. Guest triggers request at door (Pi 1 captures photo). 2. Pi 1 → MQTT (`door/guest_photo`): sends photo data. 3. MQTT → Pi 2: delivers photo; Pi 2 displays it to homeowner. 4. Homeowner clicks decision on Pi 2. 5. Pi 2 → MQTT (`door/guest_response`): sends decision. 6. MQTT → Pi 1: delivers decision; Pi 1 actuates servo/buzzer accordingly.

This architecture proved to be effective in testing. The use of two Raspberry Pis is flexible: in a real deployment one might replace Pi 2 with a smartphone app or a web interface subscribing to the same MQTT topics (since many MQTT client libraries exist). The Raspberry Pi 1 could also have been combined with the broker on one device to reduce latency. We kept the architecture modular to mirror the idea of a door device and a personal device.

Prototype Code on GitHub

The full source code for the Smart Door Lock System prototype is available on GitHub (including the door unit and reviewer unit programs, configuration files, and a README): **Project Repository – SmartDoorLock (GitHub)**. (This link is a placeholder for the actual repository URL.)

The repository is organized into two main Python scripts corresponding to the two Raspberry Pi components: - `door_unit.py` – Code for Raspberry Pi 1, handling camera capture (using system calls to `libcamera-still` or OpenCV), MQTT publish/subscribe (using the Paho MQTT Python client), and GPIO control (for the servo and buzzer). This script contains configuration variables at the top (like MQTT broker address, topics, and any PIN code for primary user). - `reviewer_unit.py` – Code for Raspberry Pi 2, handling the Tkinter GUI, MQTT subscribe/publish for messages, and image decoding. It runs a Tkinter main loop and uses either threading or the `mqtt.loop_start()` method to keep MQTT handling in the background so the GUI remains responsive.

Additionally, the repository may include: - A requirements file or setup instructions for installing dependencies (e.g. `paho-mqtt`, possibly `PIL/Pillow` for image handling if needed, etc.). - Example images or sound files if any were used for the buzzer (though we generated tones in code). - Documentation on how to configure the broker and run the system (similar to what's in this report's User Manual section).

(Note: The actual code link and repository details would be inserted in a final version of this report. The above is a placeholder summary.)

Testing Approach

Testing the smart door lock prototype involved both **unit testing for individual components** and **integration/system testing for end-to-end scenarios**. Due to the hardware-oriented nature of the project, many tests were performed manually or with custom test routines rather than using formal testing frameworks, but we strived to cover various edge cases and ensure reliability.

Unit Testing and Component Verification: - *Image Capture and Encoding:* We tested the camera capture function in isolation by calling the `libcamera-still` command via our code and verifying that an image file is saved. We then tested the base64 encoding by encoding a known image and decoding it back to ensure no data loss. This ensured that the photo sent over MQTT could be reconstructed accurately on the other side. In absence of the actual camera (for example, during early development on a PC), we simulated by reading a static image from disk and sending that, to test the MQTT pipeline. - *MQTT Communication:* Using a test MQTT broker and a simple MQTT client tool (like `mosquitto_sub` and `mosquitto_pub` from the command line), we verified that our topics were working. For example, we manually published a sample base64 string to `door/guest_photo` and saw that the reviewer GUI displayed the image, and conversely, tested that when the GUI publishes "APPROVED", the door unit receives it and prints the expected message. We also tested MQTT under different conditions: with the broker offline (the client code should handle connection errors and retry), and with large payloads to see if any performance issues arose. - *GPIO and Hardware:* We wrote small test scripts to individually test the servo motor and buzzer. For the servo, a simple sweep test (rotating to 0, 90, 180 degrees) ensured that the wiring and control library (RPi.GPIO) were functioning and that the servo could physically move the lock mechanism. We adjusted angles to match the lock's requirements (e.g., our door bolt might only need ~45° turn to open). For the buzzer, we tested different beep patterns and frequencies to select a noticeable yet not too loud tone. We also verified that the buzzer and servo could operate nearly simultaneously or in quick succession without browning out the Pi (since both draw power from GPIO; in some cases a driver circuit or separate power might be needed for stronger locks – our low-power servo and buzzer were fine on the Pi's 5V/GPIO power). - *GUI Components:* The Tkinter interface was tested for proper layout and resizing. We ensured that the image display box can show the photo at a reasonable size and that the buttons invoke the correct callbacks. Because Tkinter runs on the main thread by default, we carefully tested that the MQTT client doesn't freeze the GUI. In our implementation, we used `mqtt.Client.loop_start()` to run the network loop in a separate thread, and we tested that incoming MQTT events correctly update the GUI using thread-safe methods (like Tkinter's `after()` or a queue). We simulated multiple incoming photos to ensure the GUI updates each time and doesn't hold onto old images in memory (to avoid leaks).

Integration and System Testing: We conducted a series of full end-to-end tests using the two Raspberry Pis and the actual hardware setup: - *Single Guest Happy Path:* A tester would press the doorbell button (simulated by a GPIO trigger or simply starting the process via command). We observed Pi 1 capturing the photo and publishing it. On Pi 2, the GUI showed the image and the tester clicked Approve. Pi 1 received the approval and unlocked the servo (we could physically see the lock move) and the buzzer played the success tone. This confirmed the basic functionality. We repeated this many times to ensure consistency. - *Decline Path:* We repeated the test but with the reviewer clicking Decline. We verified that the servo did not actuate (lock stayed closed) and that the buzzer played the "denied" tone. The guest Pi 1 console also printed a message indicating access was denied. This validated that negative cases are handled. - *Timeout Scenario:* We simulated the case where the homeowner is unavailable. The guest triggers a request, Pi 1 sends the photo, but no one clicks a decision on Pi 2. After the predetermined timeout (we tested various durations, e.g., 30s), Pi 1's logic timed out. We checked that it then did not unlock the door (expected) and that it gave

the guest an indication (in our prototype, we set the buzzer to emit three long beeps upon timeout). The Pi 2 GUI, in this case, might still show the pending request; we decided that after a timeout, Pi 1 can optionally publish a "DECLINED" itself or just close the request. We chose to publish a "DECLINED" on timeout so that the GUI would get a message and could display a notification like "No response – request timed out". - *Primary User Local Unlock:* To test the primary user authentication (safety password) path, we configured a PIN code (for example, "1234") in the door unit. We then simulated the homeowner arriving at the door: instead of sending a photo, we entered the PIN on Pi 1 (in practice, this could be done via a keyboard or a keypad; in our test, we ran the script in a mode where it asked for input in the console). Upon entering the correct PIN, the program bypassed the MQTT flow and directly activated the servo to unlock. We also tested entering a wrong PIN to ensure it then proceeded to treat the person as a guest (i.e., capture photo for remote approval). This dual-mode operation worked, confirming that a local override is functional. - *Multiple Sequential Guests:* We wanted to ensure the system could handle repeated use without rebooting. We ran a scenario where one guest arrives and is approved, then a few minutes later another guest arrives. The system successfully handled a second request in the same run. We also tested back-to-back requests quickly (this is a bit artificial, but we wanted to see if any state carried over). As expected, if one request was still pending (not decided yet) and another came, our design actually queues them – however, in practice we decided to lock out new requests until the current one is resolved or timed out, to avoid confusion. The prototype thus ignored a new button press if it was already awaiting a decision. - *Network Loss Test:* We tested what happens if the MQTT connection is lost mid-process. For example, after sending a photo, we disconnected Pi 2's network. Pi 1 would not get a response and eventually timed out. We made sure that such a timeout was handled the same as no response (safe failure). When network was restored, the system reconnected automatically (Paho MQTT handles reconnection). Similarly, if the broker was down entirely, Pi 1's attempt to send would fail – we handled this by catching the error and displaying an error on Pi 1 (and using buzzer error code). These tests ensured the system fails gracefully. - *Performance and Delay Measurement:* During testing we measured the time from capturing the image to unlocking. On a local network, the MQTT message with an image (~50KB JPEG) took under a second to reach Pi 2. The overall decision loop (assuming homeowner responds quickly) was often around 3-5 seconds, dominated by the time to manually click the GUI and the servo movement. This is acceptable for a door lock scenario. We also tried a slower network (using a throttled connection) to simulate remote internet usage, and while it slowed image transfer, the system still worked albeit with a longer wait.

Throughout testing, we made extensive use of **logging and print statements** for debugging. Each step in both Pi's scripts logs an action (e.g., "Photo captured, size X bytes", "Published to MQTT", "Received APPROVED, unlocking door", etc.). These logs were invaluable in diagnosing issues, such as an early bug where the GUI didn't update because the MQTT callback was not in the main thread. By reviewing console output and log files, we iteratively improved the reliability. We also used the logs to verify that every request/response pair was accounted for and to ensure no duplicate messages (initially, we saw the photo arrive twice because we had retained messages on MQTT – we then adjusted the settings).

In summary, testing confirmed that the prototype meets its requirements for typical use cases and handles error conditions gracefully. Any issues discovered (like thread synchronization in the GUI, or ensuring the servo returns to lock position after each cycle) were fixed as part of the test-feedback cycle. The result is a prototype that one can confidently interact with in a demo or real trial.

User Manual

This section guides a user (or developer) through the setup, installation, and usage of the Smart Door Lock System prototype. It covers the required hardware and software, how to get the system running on both Raspberry Pis, and instructions on using the system for both guest and primary users.

1. Hardware Requirements

To replicate or deploy the prototype, you will need the following hardware components:

- **Raspberry Pi 1 (Door Unit):** Any modern Raspberry Pi with GPIO pins and camera interface (e.g., Pi 3, Pi 4, or Pi Zero 2 W if performance is sufficient). Raspberry Pi OS (formerly Raspbian) should be installed. This Pi will be placed at the door.
- A **Raspberry Pi Camera Module** compatible with your Pi, or a USB webcam (the code is written for the Pi camera using libcamera, so for a USB webcam you'd need to adjust the capture method). The camera should be mounted such that it can capture the face of someone at the door.
- A **Servo Motor** to act as the door lock actuator. In our prototype, a small 5V servo (such as the SG90 or MG90S) is used to manipulate the lock. The servo should be mechanically connected to the door's locking mechanism (this could be a simple latch that the servo arm can move). You might need some mounting hardware or 3D-printed parts to attach the servo to your door lock.
- A **Buzzer** or piezo speaker for audio feedback. A simple 5V active buzzer can be driven by a GPIO pin. (Optionally, an LED could also be used for visual feedback, though not explicitly in our design).
- (Optional) **Doorbell Button or Keypad:** A momentary push-button connected to a GPIO input can serve as a doorbell for guests to press. If implementing the primary user PIN entry, a keypad (matrix keypad or even a numeric USB keypad) would be needed to input the code. In the simplest form, a USB keyboard can be attached for testing PIN entry via console.

- **Power Supply and Miscellaneous:** A 5V 2.5A (or higher) power supply for the Pi. Jumper wires to connect the servo and buzzer to the GPIO pins. If the servo draws significant current, an external power source or a driver (like a ULN2003 driver board) might be required, but in our small servo case the Pi's 5V pin sufficed.

- **Raspberry Pi 2 (Reviewer Unit):** Another Raspberry Pi to run the GUI. This could be a desktop environment or even headless with VNC. We recommend Pi 3 or newer for smoother GUI performance. It should have a display (HDMI monitor or the official Pi touchscreen) for the homeowner to see the GUI.

- This Pi will need network connectivity (Wi-Fi or Ethernet) to communicate with the MQTT broker and Pi 1.

- If a Raspberry Pi is not convenient for the reviewer side, this part of the system can run on a standard PC or laptop with Python as well, since it's just a Python Tkinter app with MQTT. (For the scope of this project, we assume a Pi for uniformity.)

- **MQTT Broker:** You need an MQTT broker accessible to both Pis. There are a few options:

- Install **Mosquitto** broker on one of the Raspberry Pis (e.g., on Pi 1). This is convenient and keeps everything local.
- Or run a broker on another machine on the same LAN. It could even be a cloud broker service if you wish to allow remote internet connectivity (in which case ensure you have internet access and proper security).

- For initial setup, installing Mosquitto on Pi 1 or Pi 2 is simplest. (Instructions: `sudo apt-get install mosquitto` on Raspberry Pi OS, which by default sets up a broker with no authentication listening on port 1883.)

- **Networking:** Ensure both Raspberry Pis are connected to the same network (if using a local broker). If using a cloud broker, ensure both have internet access. It's important that Pi 1 and Pi 2 can reach the broker's IP address.

2. Software Requirements

- **Operating System:** Raspberry Pi OS (32-bit or 64-bit) latest version is recommended for both Pi 1 and Pi 2. The OS should have Python 3 (which is installed by default) and the libcamera stack for Pi 1.
- **Python Packages:** The project uses Python 3. Key libraries that need to be installed:
- `paho-mqtt` – the MQTT client library for Python. Install via pip: `pip3 install paho-mqtt`.
- `tkinter` – for the GUI. This comes pre-installed with standard Python on Raspbian. (On minimal installations, you might need to install via `sudo apt-get install python3-tk`.)
- `PIL/Pillow` – (optional) for image processing. If the code uses PIL to convert from bytes to an image for Tkinter, you need Pillow: `pip3 install pillow`. In our implementation, we used Tkinter's PhotoImage with base64 data, which might not require Pillow explicitly.
- `RPi.GPIO` – for controlling GPIO pins (servo and buzzer) on Pi 1. This is usually pre-installed in Raspbian Python. If not, `pip3 install RPi.GPIO`.
- No external library is needed for camera if using `libcamera-still` via subprocess. If using OpenCV or similar for capture (optional approach), then OpenCV (`opencv-python`) would be needed, but that is heavyweight. Our guide will assume libcamera via command line for simplicity.
- **Mosquitto Client (optional):** For debugging or manual testing, you might install Mosquitto clients: `sudo apt-get install mosquitto-clients`. This provides `mosquitto_pub` and `mosquitto_sub` which can be useful to test MQTT topics from terminal.

3. Installation & Setup

Follow these steps to set up the system:

On Raspberry Pi 1 (Door Unit): 1. **Enable Camera:** If using the Pi Camera Module, enable it via `raspi-config`. Under Interface Options, enable Camera, then reboot. Also ensure the camera is properly attached. 2. **Install Mosquitto (if using Pi 1 as broker):** `sudo apt-get install mosquitto`. After installation, the broker will run automatically. (This step can be skipped if you have another broker.) 3. **Prepare Project Code:** Transfer the `door_unit.py` (or equivalent) script to Pi 1. You can use `scp` to copy it or download from GitHub if the Pi has internet. Place it in, for example, `/home/pi/smartdoor/door_unit.py`. 4. **Install Python dependencies:** As mentioned, ensure `paho-mqtt` and other libraries are installed. For instance, run:

```
sudo apt-get update
sudo apt-get install python3-pip python3-tk -y   # Install pip and Tkinter if
not present
pip3 install paho-mqtt pillow
```

(Pillow only if needed for image handling.) 5. **Hardware Connections:** Connect the servo to a GPIO pin (for PWM we used pin 17 GPIO, but check that matches the code's configuration). Connect servo's power to 5V and ground to GND. Connect the buzzer to another GPIO (e.g., GPIO 27) and its other lead to GND (if it's a 2-pin buzzer; if polarized, ensure correct orientation). If using a button, connect one side to a GPIO input (e.g., GPIO 23) and the other side to ground (with an internal pull-up in code). Adjust these pin numbers in the code if needed to match your wiring. 6. **Configure the Script:** Open `door_unit.py` in an editor and set configuration values: - MQTT broker address: If Mosquitto is on this Pi, use `localhost`. Otherwise, put the IP of the broker machine. - MQTT topics: they are likely already `door/guest_photo` and `door/guest_response` as default. No change needed unless you prefer different names. - PIN code: If the code has a section for a primary user PIN or password, you can set it (for example, `PRIMARY_PIN = "1234"`). - GPIO pins: match the pin numbers to how you wired servo, buzzer, etc. - Possibly camera settings: If you want to adjust image resolution or file path for the photo. 7. **Test Run:** You can now run the door unit script. In a terminal on Pi 1:

```
python3 ~/smartdoor/door_unit.py
```

Initially, do this while watching the console output. If configured to capture immediately, it may take a picture (ensure something is in front of camera). If using a button trigger, you may have to press the button to see it proceed. You should see logs indicating whether it connected to MQTT successfully. If any errors occur (e.g., camera not found, MQTT connection refused), fix those before continuing. Leave this running for now or you will restart it later.

On Raspberry Pi 2 (Reviewer Unit): 1. **Desktop Environment:** Ensure Pi 2 is set up with a display and you can view a GUI (if using the official OS with desktop, you're fine; if headless, you might use VNC or X forwarding to see the Tkinter window). 2. **Prepare Project Code:** Transfer the `reviewer_unit.py` script to Pi 2 (`scp` or direct download). Place it e.g. in `/home/pi/smartdoor/reviewer_unit.py`. 3. **Install Python dependencies:** Similar to Pi 1, you need `paho-mqtt`, and Tkinter (likely preinstalled). Pillow might be needed depending on implementation. Run:

```
sudo apt-get update
sudo apt-get install python3-pil python3-tk -y
pip3 install paho-mqtt
```

(The above ensures Pillow and Tkinter; adjust if needed.) 4. **Configure Script:** Open `reviewer_unit.py` in an editor, set the MQTT broker address to the same as used for Pi 1 (if using Pi 1 as broker, use Pi 1's IP address here). Ensure the topics match (`door/guest_photo`, etc). Also, if there are GUI settings (window size, etc.) you can tweak as desired. 5. **Test Run:** Open a terminal on Pi 2 and run:

```
python3 ~/smartdoor/reviewer_unit.py
```

A window should appear (likely titled "Door Lock Access Controller" or similar). If the program prints "Connected to MQTT broker" and is waiting, you are set. You won't see an image until Pi 1 sends one. You can test the GUI by manually publishing a test image: from Pi 1's terminal (or any machine with access), publish a base64 string to `door/guest_photo` or run the Pi 1 flow. The Pi 2 GUI should display the image and show the Approve/Decline buttons. Pressing the buttons should result in a log message that it published the response (and if Pi 1 is running, you'd see it receive it).

Networking Note: If Pi 1 and Pi 2 are on the same network, ensure you know the IP addresses. If using Pi 1 as broker, you might need to allow external connections (edit `/etc/mosquitto/mosquitto.conf` to allow listener on all interfaces, or start mosquitto with `-c` config). By default, Mosquitto on Raspberry Pi might allow all connections on port 1883 without auth (check firewall). For security, on a home network, this is okay for testing, but consider adding a username/password in Mosquitto for production.

4. Usage Guide

Once both components are set up and running, here's how the system is used by the two parties:

For the Guest (Visitor at the Door): 1. The guest arrives at the door. They will typically locate a doorbell button or some interface on the door unit. In our prototype, if a button is available, the guest should press it to initiate the access request. (If no physical button, the system might automatically start when it detects motion or simply be triggered manually by the operator for demo purposes.) 2. After pressing the button, the guest may see an indication (an LED flash or a buzzer short beep) that their request is being processed. The camera will take their picture — guests should be advised to face the camera for a clear image. 3. The guest then waits briefly (a few seconds) while the homeowner is being notified. The buzzer may emit a "please wait" tone. It's good to inform the guest that the system is contacting the homeowner. 4. When the homeowner makes a decision, the guest will be informed of the outcome: - If **approved**, the door will unlock. The servo will have moved the lock, so the guest can now turn the knob or push the door open (depending on door mechanism). The buzzer will also sound a happy tone (for example, two quick beeps). The guest can then enter. - If **declined**, the door remains locked. The buzzer might buzz with a negative tone (e.g., a long buzz) indicating no entry. Optionally, a message could be spoken or displayed if there was a screen, but in this prototype we rely on buzzer signals. - If there was **no response (timeout)**, it is treated like a decline – the guest might hear a series of beeps that indicate no one answered. The guest can choose to try again after some time or use an alternate method to contact the homeowner. 5. In case the guest has been given a one-time code or is a trusted person with a known PIN (this would be a scenario where the homeowner pre-authorized them), and if a keypad is present, the guest could enter that code instead. Our current system doesn't fully implement guest PINs, but it's a logical extension (currently only the primary user PIN is handled). If implemented, the guest entering a valid temporary PIN would directly unlock the door (the door unit would verify the code without contacting the homeowner for that case).

For the Primary User (Homeowner using the Reviewer App): 1. The homeowner should have the reviewer application running on Raspberry Pi 2 and be connected (it can be running in the background, but ideally visible so they notice a request). This application can be started at boot for convenience (for instance, using `lxsession` autostart or a systemd service) so that it's always ready. 2. When a guest arrives and presses the doorbell, the homeowner will get an immediate notification on the GUI. The app will display the

guest's photo in the window. (Enhancements could include an audible alert on the Pi 2, but at minimum the image will appear). 3. The GUI will have two buttons: **Approve** and **Decline**. The homeowner should click the appropriate button after possibly contacting or recognizing the guest: - If they know the guest (e.g., a relative, friend, or an expected visitor), they click **Approve** to unlock the door. - If it's someone unknown or unwelcome, they click **Decline**. - The homeowner may also choose not to respond (if uncertain or away from the screen). In that case, after the timeout, the system will default to decline. 4. After clicking a decision, the GUI might show a brief confirmation like "Sent APPROVED" or update the status. The homeowner should soon hear/see the result from the door (if within earshot, the buzzer or the door opening). If the system is configured to do so, it could also show on the GUI that the action succeeded (e.g., a label "Door Unlocked" for a few seconds). 5. The homeowner can then close the GUI or leave it open for the next visitor. It's intended to remain running continuously. There is typically a log window or console output on Pi 2 as well, which can be checked for debug info if needed.

Maintenance and Troubleshooting: - If the camera fails or shows a black image, check the connection and lighting. Ensure the camera lens is not obstructed. You can test the camera independently using the `libcamera-still -o test.jpg` command. - If the servo does not move, check the GPIO wiring and power. It may also require calibration – perhaps the angle in code is not sufficient to unlock; you might need to adjust the angle values in the script to match your lock's needed rotation. - If the MQTT connection isn't working (e.g., no photo arrives on Pi 2), verify the broker IP and that both devices are on the network. You can use `ping` between the Pis. Also, try running `mosquitto_sub -t "door/guest_photo"` on Pi 2 to see if any message arrives at all, to isolate if it's a broker issue or the publish code on Pi 1. - Logs are your friend: run the scripts in a terminal to see printouts. Common issues could be "Connection refused" (broker not running or wrong address), or Python exceptions if a library is missing. - To start the system on boot (optional advanced step): you can create a cron job @reboot or systemd service so that `door_unit.py` starts automatically on Pi 1 and `reviewer_unit.py` on Pi 2. This way, the system will be always on like a real appliance. Make sure to also start the broker on boot (if Mosquitto is installed as a service, it will auto-start by default).

By following this guide, a user should be able to set up the smart door lock prototype successfully. Once running, the system provides a basic but functional smart lock experience: a visitor can request access and the owner can remotely grant or deny entry after visual verification. The next section provides a link to a demonstration video to further illustrate the working system.

Link to Demonstration Video

A video demonstration of the Smart Door Lock System prototype in action is available here: **Demo Video – Smart Door Lock Prototype**. (This link is a placeholder; in an actual report, it would lead to a video showing the system working.)

In the demonstration video, you will see: - The guest pressing the doorbell button (simulated or actual), and the Raspberry Pi camera capturing the guest's image. - The MQTT message being sent and the homeowner's GUI popping up the visitor's photo. - The homeowner clicking the Approve button, and consequently the servo motor unlocking the door (you can observe the latch movement in the video) accompanied by the buzzer sound. - A scenario of Decline as well, showing the buzzer indicating denial and no movement of the lock. - Optionally, a segment showing the timeout case or the primary user using the PIN to unlock (if included in the video).

This video is intended to give a clear, real-world look at how the prototype operates, complementing the descriptions in this report.

Conclusion

In developing this Raspberry Pi–based smart door lock prototype, we successfully demonstrated a working system that addresses the initial problem: enabling remote, photo-verified door access. The project combined multiple disciplines — hardware control (camera, servo, buzzer), network communication (MQTT messaging), and user interface design (Tkinter GUI) — into a cohesive prototype. Through this process, we gained valuable insights and also encountered various challenges which we overcame:

Implementation Reflections: One major challenge was synchronizing the MQTT communication with the GUI and hardware actions. Initially, we faced an issue where the Tkinter GUI would freeze when waiting for messages – this was solved by using MQTT's asynchronous loop and updating the GUI via thread-safe methods. Another challenge was the timing of camera capture and servo movement: the camera needed a moment to warm up and capture, so we had to ensure the guest was informed to hold still briefly. Fine-tuning the servo angles to properly lock/unlock the door was a mechanical trial-and-error task; we learned that different servos have slightly different ranges and that a robust lock mechanism might require a stronger motor or a geared system. MQTT proved reliable, but we did notice that sending very large images could introduce a slight delay; in future iterations, we might compress images more or even consider sending a short live video or multiple frames for better context to the reviewer.

Key Achievements: The system performed as intended in our tests – guests were able to request access and homeowners could remotely grant or deny entry with confidence. The use of a photograph for identification is a step towards a more secure authentication (it's not as secure as automatic face recognition, but it allows a human to make the decision which can be more nuanced). The modular design (with separate devices) validated that such a pattern can be extended; for instance, one could imagine multiple door units (front door, back door) and a single reviewer app handling all, simply by using different MQTT topics or including an ID in messages.

What Could Be Improved in Version 2.0: As with any prototype, there are many enhancements and improvements that we would consider for a next iteration: - **Security Enhancements:** Currently, our MQTT communication is unencrypted within a local network. For a real deployment, we would enable TLS on MQTT or use an authenticated broker to prevent any possibility of interception or spoofing of messages. We might also implement a more secure handshake (for instance, the door unit only unlocks if it receives a cryptographically signed approval, to avoid any malicious command injection). - **User Interface & Experience:** The Tkinter GUI, while functional, is quite basic. A polished interface or a mobile app would greatly improve user experience. For example, a smartphone app could receive a push notification with the guest's photo and have Approve/Deny buttons, which internally publish to MQTT. This would free the homeowner from needing to be at a dedicated Pi screen. Additionally, the door unit could benefit from voice prompts or a small display to guide the visitor (e.g., "Please smile for the camera" or "Access granted, welcome!"). In a second version, integrating a speaker for voice feedback or at least multi-tone buzzer feedback would be considered. - **Scalability and Cloud Integration:** We could integrate the system with cloud services to allow monitoring from anywhere. For instance, using an online MQTT broker or bridging to an HTTP API so that even if the homeowner is truly remote (not on the home LAN), they could receive the request (perhaps as a notification on their phone). This raises security considerations but is doable. We could also log events to a database or Google spreadsheet for audit (time of entry, image of visitor,

decision). - **Additional Sensors and Features:** To make the system more autonomous, we might add a motion sensor or use the camera feed to detect motion so that a guest doesn't need to press a button – the system could detect someone at the door and start the process (similar to a smart doorbell with motion detection). We also considered adding **facial recognition** for familiar faces: e.g., if the camera recognizes the homeowner's face approaching, it could unlock automatically (seamless entry for the primary user). In fact, some modern smart locks offer face unlock [3], so this would be a natural extension using OpenCV or cloud vision APIs. Another feature could be a two-way audio communication, essentially turning it into a smart intercom (so the homeowner can talk to the guest via a speaker/microphone on the door unit before deciding). - **Mechanical Reliability:** The servo mechanism, while okay for a demo, may not be robust enough for a real door lock in the long term. For version 2, we might use an electronic strike or a solenoid lock which are purpose-built for doors. These can be driven by a relay from the Pi. The advantage would be a stronger, more secure locking mechanism. We would then focus the Pi's role on the logic and keep the actual lock hardware more standard. - **Power and Backup:** Currently the system runs on mains power via the Pi's adapters. For a real product, one would consider backup power (like a battery for the door unit) so that it works during power outages (at least to unlock for the owner). Also, an alert could be generated if the system goes offline or if someone tries to tamper with it. - **Code Improvements:** Refactoring the code for better modularity (e.g., separating MQTT handling, camera capture, GUI into different modules) would make it easier to maintain and extend. Adding more thorough exception handling and perhaps a configuration file for easy changes (like changing broker address, PIN codes, etc., without editing the code) would be user-friendly.

Conclusion: The project was a rewarding exercise in IoT and smart home prototyping. It demonstrated how a combination of readily available components can emulate functionalities of high-end smart locks and doorbells at a fraction of the cost. More importantly, it provides a platform that we can continually improve. Through this prototype, we learned about the importance of reliable inter-device communication, user-centered design (since security devices must be foolproof and easy to use), and the challenges of integrating hardware with software in real time. The knowledge gained here can inform future projects or products in the domain of home automation and security.

Overall, the Smart Door Lock System prototype achieved its primary objectives and opened the door (quite literally) to numerous possibilities for future development in secure, smart access control systems. With further refinement and testing, such a system could eventually evolve from a prototype to a deployable solution for home security enthusiasts or even commercial use.

---

[1] [2] [3] Best smart locks 2025 | Tom's Guide
https://www.tomsguide.com/us/best-smart-locks,review-3352.html

[4] The Basics of the MQTT Protocol | IoT For All
https://www.iotforall.com/the-basics-of-the-mqtt-protocol

[5] MQTT Publish/Subscribe Architecture (Pub/Sub) - HiveMQ
https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe/

[6] Raspberry Pi Camera Module: Still image capture
https://www.raspberrypi.com/news/raspberry-pi-camera-module-still-image-capture/