

SWE4211 Lab 5: Networked Light Control

1 Introduction

Up to this point, you've debugged C++ code on the Raspberry Pi, controlled LEDs, and even developed a simple game. In this exercise, you'll take the next step by introducing networking capabilities to the Raspberry Pi. You'll be provided with a basic Java program that sends messages over the network using a simple protocol. When it receives these messages, the Raspberry Pi will turn the board's lights on or off and adjust their brightness.

2 Lab Objectives

- Develop embedded code using network sockets.
- Interpret data in a simple communications protocol.
- Construct a synchronized queue to process received messages.
- Control GPIO using the network.
- Control the brightness of a light using software PWM.
- Verify the robustness of your implementation through invalid-input (negative) testing.

3 Laboratory Equipment Needed

1 Raspberry Pi embedded systems board 	1 Raspberry Pi Custom GPIO Hat 
2 Ethernet cables 	1 USB Power Supply 
1 USB to Ethernet adapter 	Java Virtual Machine and Eclipse or IntelliJ IDE 



4 Deliverables / Submission

When finished, commit and push your completed source code to your repository. Comment your code so that Doxygen can generate documentation from it.

You will have two deliverables: a report and a video demonstration.

4.1 Report

Each team will be responsible for submitting one report into Canvas with the following contents:

1. Title Page
 - a. Names
 - b. Assignment
 - c. Date
 - d. Name of your GitHub Classroom repository
 - e. Video link
2. Introduction
 - a. In several grammatically correct sentences of your own, describe what this lab is intended to accomplish.
3. Screen captures and data captures
 - a. Include a screen capture of your software building without warnings. To do this, run “make clean” followed by “make all” in Eclipse or on the console.
 - b. Include an htop capture showing the CPU usage of each thread while the program runs.
 - c. Include an oscilloscope screen capture verifying that the program can generate a correct PWM signal at approximately 20% and 80% duty cycle.
4. Discussion: Answer each question in several grammatically correct sentences.
 - a. What protocol does this system use across the network? Why might it be chosen over the main alternative transport protocol? How does this choice affect the design of the receiving end?
 - b. How did you handle issues of endianness when messages were received over Ethernet? Did you need to do anything?
 - c. What checks did you perform on each message before enqueueing it, and how do they prevent crashes?
 - d. What happens to a blocked `recv()` call when its socket is closed?
 - e. Why does the queue need both a semaphore and a mutex rather than a mutex alone?
 - f. What is the advantage of using the `RunnableClass` and `PeriodicTask` as parent classes in this design and implementation?
5. Things gone right / Things gone wrong
 - a. Discuss what went well and what caused problems during this lab.
 - b. What aspects of C/C++ or networking proved to be the most challenging for this lab?
6. Conclusions
 - a. What did you learn from this lab? How could it be improved in the future?

If you have any questions, consult your instructor.

4.2 Video Demonstration

When finished, create a brief video (30–60 seconds) that demonstrates the following:

1. The Java program controls the lights. Show that each light can be turned on and off, and its PWM duty cycle changed, independently of the other.
2. Holding a pushbutton lights its LED, overriding the network state, even when the Java program has turned the light off.

3. Your implementation does not crash or lock up when it receives messages with invalid checksums or destinations.
4. Your implementation continues to control the lights while invalid messages flood the device (i.e., the MC Test).

5 Hardware Design

This lab uses the same hardware as the previous two labs. The GPIO pin for each LED and pushbutton is printed on the board.

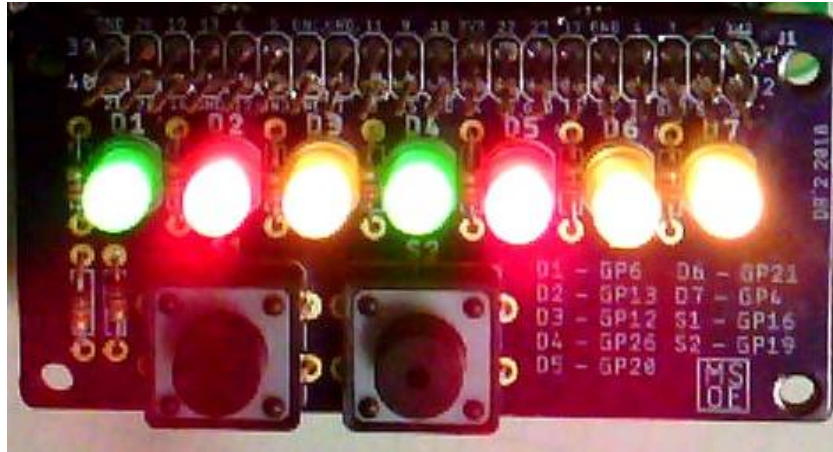


Figure 1: GPIO Board

6 Java Program

To control the lights, you'll use a simple Java program built with Java Swing. The program is intentionally straightforward, though the source code may not be fully commented. It's provided as raw source code and should be compiled and run using your native Windows Java environment for best results. While it's designed to be compatible with most Java versions, Java 8 project settings are recommended if you encounter compatibility issues. A prebuilt JAR file is also available.

You can run the program in either Eclipse or IntelliJ, though for better performance you will likely want to run it outside of your Linux development VM. The user interface is shown in Figure 2. Enter your Raspberry Pi's IP address in the text field at the bottom of the window, then click Connect.¹ You can find this by running the `ip address` command in a Raspberry Pi shell, pinging the Pi by hostname, or observing the console output when the Pi program starts. The `main()` function prints them automatically.

¹ Although the Java program attempts to resolve hostnames, this functionality is unreliable and may only work intermittently.

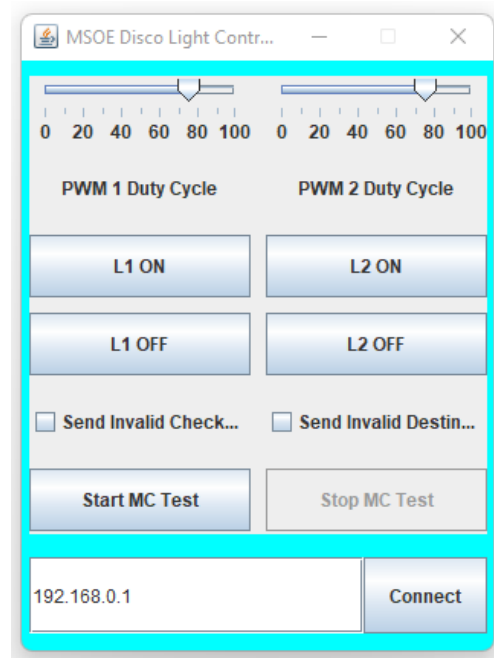


Figure 2: Java UI.

For each light, you can toggle its state over the network using the **On** and **Off** buttons. You can also adjust the brightness by modifying the duty cycle via the corresponding sliders.

At the center of the UI, you'll find several options designed to test the robustness of your network code and its ability to validate incoming messages.

- The **left checkbox** triggers the transmission of messages with invalid checksums. Your implementation should detect and reject these.
- The **right checkbox** sends messages with invalid destination addresses, which should also be properly discarded.

The **MC Test** refers to a Monte Carlo-style stress test. Click Start MC Test to have the Java program continuously send a stream of invalid messages. Your code should reject these malformed messages while still correctly processing valid messages in parallel. This test verifies that your implementation remains responsive and resilient under noisy network conditions.

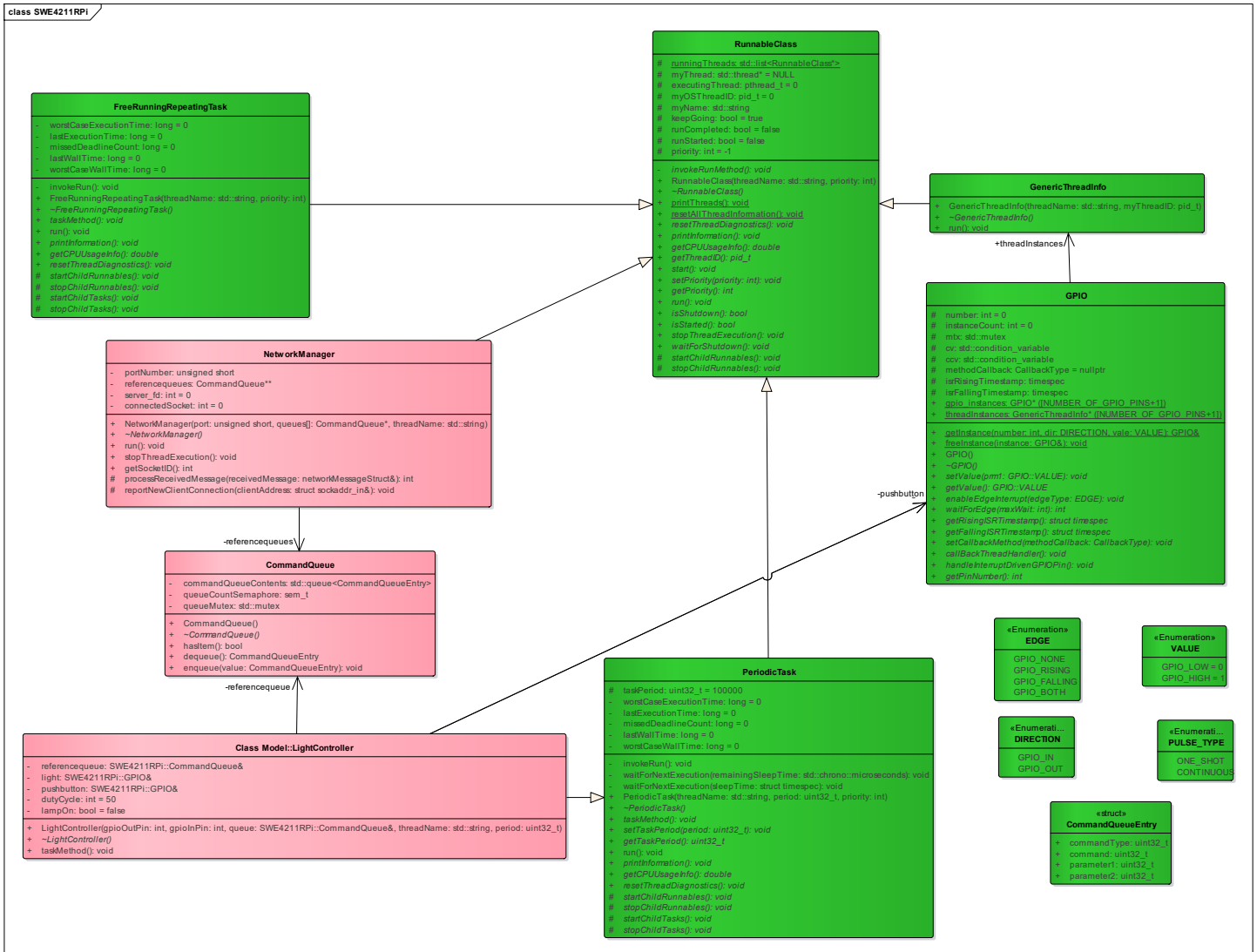


Figure 3: UML Design for the project.

7 Designing our software

The software architecture is intentionally simple, as illustrated in Figure 3. The system consists of three core classes that you will implement: **NetworkManager**, **LightController**, and **CommandQueue**.

Additional supporting classes—highlighted in green—are provided for your use and do not require modification.

This section offers a general overview of the class structure. For full documentation, refer to the `baseline_docs` folder in the repository. If you use Doxygen to generate documentation from your source code, it will be output to the `docs` folder.



a. Command Queue

The **CommandQueue** acts as a temporary buffer for commands received from the **NetworkManager**. It supports two primary operations: **enqueue** and **dequeue**, plus **hasItem()**, which reports whether an item is waiting. Each item in the queue is a **CommandQueueEntry**, which contains four fields:

- **commandType** – 0 for commands routed from the network, or **BROADCAST_MESSAGE_TYPE** for entries sent to every queue (such as **REPORT_NEW_IPADDRESS**)
- **command** – the specific instruction
- **parameter1** and **parameter2** – additional data associated with the command

To ensure thread safety, the queue uses two synchronization mechanisms:

- A C++ **std::mutex** protects against race conditions by preventing concurrent access during enqueue and dequeue operations.
- A **counting semaphore**, initialized to zero, serves as a signaling mechanism. Each time a command is enqueued, the semaphore count is incremented, indicating that an item is available for processing. Dequeuing waits on the semaphore, which decrements the count and blocks the caller if the queue is empty. The semaphore's value therefore equals the number of items in the queue.

b. Light Controller

The **LightController** class manages a single light, but the project requires two instances—one for each light connected to **GPIO 13** and **GPIO 21**. In addition to controlling the light, each instance also monitors a pushbutton connected to **GPIO 16** or **GPIO 19**.

The **LightController** is a subclass of the **PeriodicTask** class and holds a reference to its own **CommandQueue**. It receives this reference in its constructor and is configured to execute its task every **10 milliseconds**.

Calling **start()** launches a **C++11 thread** whose loop calls **taskMethod()** once per period. On each call, **taskMethod()** dequeues and processes every waiting entry. The entry's **command** field is a **bitmapped 32-bit value**. The **LightController** tests it against the bit masks in the table below, checking **LIGHTOFFCMD** first, then **LIGHTONCMD**, then **LIGHTPWMADJUSTMENTCMD**. Values that match none of the masks, such as broadcast IP reports, are ignored.

	Value (Hex)	Action	Parameter(s)
LIGHTPWMADJUSTMENTCMD	0x01000000	Sets the PWM duty cycle.	The first parameter of this message will be the duty cycle. The value will be between 0 and 1000, with 0 representing 0% duty cycle and 1000 representing 100% duty cycle. Values outside this range are ignored. The LightController stores the duty cycle as a whole percentage (0–100).
LIGHTONCMD	0x00100000	Turns this controller's light on.	None
LIGHTOFFCMD	0x00010000	Turns this controller's light off.	None

On each call, **taskMethod()** also reads its pushbutton. While the button is held, the light is lit at the current duty cycle, even if it has been commanded off. The LEDs and pushbuttons are active low.



c. Network Manager

The **NetworkManager** class is responsible for reading incoming messages from a **POSIX socket**. Each received message signals an operation to be performed by the system.

Messages follow a simple protocol consisting of a structure with **ten 32-bit integers**:

Word	Purpose	Description
1	Security Mode	Always set to 0 for this lab. In future labs, this field will hold encryption-related values.
2	Message ID	A unique identifier that increments each message sent to the device.
3	Timestamp High	Together form a 64-bit value representing the number of milliseconds since the Unix epoch.
4	Timestamp Low	
5	Message Type	For this lab, always set to <code>COMMAND_MSG_TYPE</code>
6	Destination	Indicates the target queue. A value of 1 routes the message to queue 0, and 2 routes it to queue 1. Your implementation must validate this field against the system's supported queue count, defined in <code>NetworkCfg.h</code> .
7	Command	The instruction to be executed passed directly to the queue.
8	Parameter 1	Two values specific to the command.
9	Parameter 2	
10	Checksum	A 32-bit XOR of the previous nine integers, used to verify message integrity.

The **NetworkManager** is instantiated with a **port number** and a **array of command queue pointers**, along with a thread name. Once its thread is started, it opens a TCP socket on that port (9090 in `main.cpp`) and accepts one client at a time. Upon receiving a message, it performs the necessary **endianness conversion**, verifies the checksum, confirms the message type, validates the destination, and enqueues the command into the appropriate queue. Whenever a client connects or disconnects, it also enqueues a `REPORT_NEW_IPADDRESS` entry on every queue.

This process continues as long as the `keepGoing` flag remains true. Calling `stopThreadExecution()` sets `keepGoing` to false and ends the message-handling loop.