ELEC3227 Embedded Networked System Coursework Report

Introduction

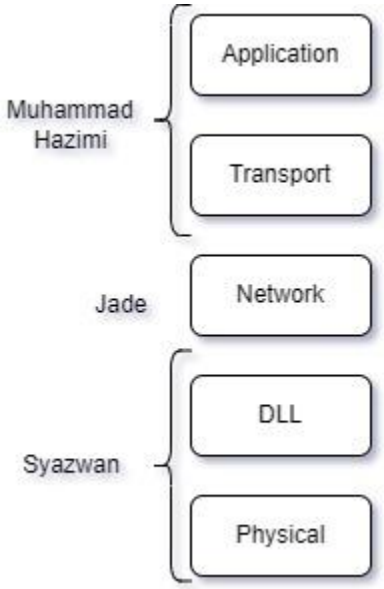
Name: Muhammad Hazimi Bin Yusri

Student ID: 32548419

Team: E2

Layers responsible: Transport and Application Layer

Distribution of Layer Responsibilities:



Application Layer

The Application Layer interfaces directly with end-users, managing user inputs and outputs for the smart lighting system. It translates user interactions, such as button presses, into data for the Transport Layer and handles high-level communication protocols.

Transport Layer

The Transport Layer ensures reliable communication between nodes, segmenting and managing data flow from the Application Layer. It handles data transmission integrity, error detection, and correction, crucial for maintaining the system's communication reliability.

Application and Transport Layer Standards

Muhammad Hazimi Bin Yusri (mhby1g21, Team E1),
Law Shaw Zuan (szl1c21, Team E2)
ELEC3227 Embedded Networked Systems Coursework

Application Layer (APP)

- Hostnames are E11, E12, E13 and E21, E22, E23 for Team E1 and E2.
- One LED on PC6 and one button on PA6, UART communication at Port D, RFM12B-S2 Radio Module at Port B.
    - Leftover pins can be used for optional implementation by individuals.
- Hostname-IP address table is used to resolve hostname into destination IP address for Network Layer.

Application 1: Button interaction with LED

- Only 1 byte of APP data will be used.
- Button LED ON sends a byte of 0xFF, button LED OFF sends a byte of 0x00.
- Button/LED interactions between nodes are hard coded.
    - Button on a node will send data (LED ON/OFF) to only one of its adjacent nodes.
    - E.g.: E11 button will turn on E12 LED, E12 button will turn on E13 LED etc...
    - The ending node will turn on starting node (E23 button turns on E11 LED)

Application 2: UART as input and output for sending strings and specific LED commands

- UART for sending/receiving messages, constrained to 9 bytes or characters maximum.
    - E.g. string commands: SEND E11 <MESSAGES> (Send string messages to E11)
- UART LED commands use 9 bytes, the first left-most byte is reserved for basic ON/OFF byte functionality and the remaining bytes are for individual team use.
- UART commands to interact with LED on other nodes.
    - Example of basic commands: ON E11 (Turn on LED on node E11), OFF E23 (Turn off LED on node E23), ON E11, E22 (Turns on LED on E11, and E22)
    - Example of optional commands that uses additional parameters: ON E11 R5G9B7 (Turn on optional RGB LED with different brightness level given)
        - Clarification: UART to control LEDs will result in 0xFF source port and 0x00 LED port so it can be easily differentiated from button presses.
- The special character ("/") is used to toggle between input or output mode in UART command line interface (cli).

Interfaces:

- APP Layer provides destination port, source port and destination IP address to TRAN Layer for correct connection.
- APP Layer provides APP data with correct length to TRAN Layer for segmentation.

Transport Layer (TRAN)

TRAN:	Control [2]	SRC Port [1]	DEST Port [1]	Length [1]	APP Data [1-114]	Checksum [2]

Figure 1. Segment field structures defined by the coursework

- Using TCP (3-way handshake connection) with segment structure shown below,
  a. Control bits (2 bytes)

Left Byte Assignment

(15)	(14)	(13)	(12:10)	(9)	(8)
SYN	ACK	DR	BUF	PERSONALUSE	PERSONAL USE

** SYN and ACK for handshake, BUF for buffer remaining, DR for disconnect request

Right Byte Assignment

(7:4)	(3:0)
SEQ NUM	ACK NUM

b. SRC Port (1 byte)
   - 0x00 for button presses, 0xFF for UART cli inputs.

c. DEST Port (1 byte)
   - 0x00 for LEDs, 0xFF for UART cli outputs.

d. Length (1 byte)
   - Length of APP Data bytes which are not 0x00, for example during TCP initial and disconnect handshake, it would be 0 as no APP Data is sent.
   - Unused bytes of UART messages will be set to 0x00, thus will not always be 9 bytes.

e. APP Data (9 bytes)
   - Leftmost 1 byte for TCP handshake, leftmost 1 byte for LED with button, and 9 bytes for UART messages/commands.

f. Checksum (2 byte)
   - ~~Even parity bit check~~
     - ~~Simpler checksum is used to reduce the processing time requirement as both NET and DLL layer are already using CRC~~

   XOR Checksum is used instead.

Interfaces:

- TRAN Layer provides destination IP address to Network Layer.
- TRAN Layer provides segment to Network Layer

Design

Introduction

The target device for our network protocol is the Il Matto, which is equipped with the Atmel Mega644P microcontroller and an RFM12B radio module. Given the hardware constraints of this microcontroller in terms of memory and processing speed, our design focuses on balancing efficiency and functionality.
Memory specification adapted from Il Matto Quick Reference Sheet:

ATMEGA	164P/PA	324P/PA	644P/PA	1284P
Flash	16K	32K	64K	128K
Boot	2K	4K	8K	8K
EEPROM	½K	1K	2K	4K
SRAM	1K	2K	4K	16K

This means whole integrated program when compiled and ready to flash should be under 64kb. We have 2kb of EEPROM and 4kb of SRAM which is what will determine the number limit of variables we can use in our whole program (including all layers and other libraries). Thus, most variables that uses low number of integer should use uint8_t (1 byte unsigned integer, which is the lowest memory assignment available when programming in avr C) to optimize memory usage efficiently. The memory size used can be found using avr-size which is only available in binutils-avr which is accessed using Linux WSL.

Application Layer

The Application Layer is designed with two main functions in mind, compatible with the Il Matto's hardware capabilities:

1. Button-LED Interaction: Utilizing minimal data bytes, this feature is optimized for the microcontroller's memory and processing capabilities, ensuring low power consumption and quick response times.
2. UART-Based Communication: We've limited messages to 9 bytes to ensure clear and effective communication within the constraints of the Il Matto's hardware.

Basic System Pin Assignments:

LED → PC6 ; additional LEDs will be at Port C for standardization and easier wiring

Button → PA6 ; additional Buttons will be at Port A for standardization and easier wiring, debouncing is done in software.

UART at Port D

RFM at Port B

Transport Layer

The Transport Layer uses a TCP-like model, tailored to provide reliability in data transmission, which is crucial for a smart lighting system with additional communication functionality. TCP, or Transmission Control Protocol, is a connection-oriented protocol that ensures all data sent by one device is received completely intact by another. Main features:

1. Reliability and Order: TCP's ability to manage ordered data streams and ensure complete data reconstruction at the destination makes it suitable for applications where data integrity is crucial, like in our smart lighting system.

2. Error Checking and Retransmission: TCP's extensive error checking and acknowledgment mechanisms, along with the ability to retransmit lost packets, provide a stable and reliable communication channel, which is essential for the consistent operation of the lighting system.

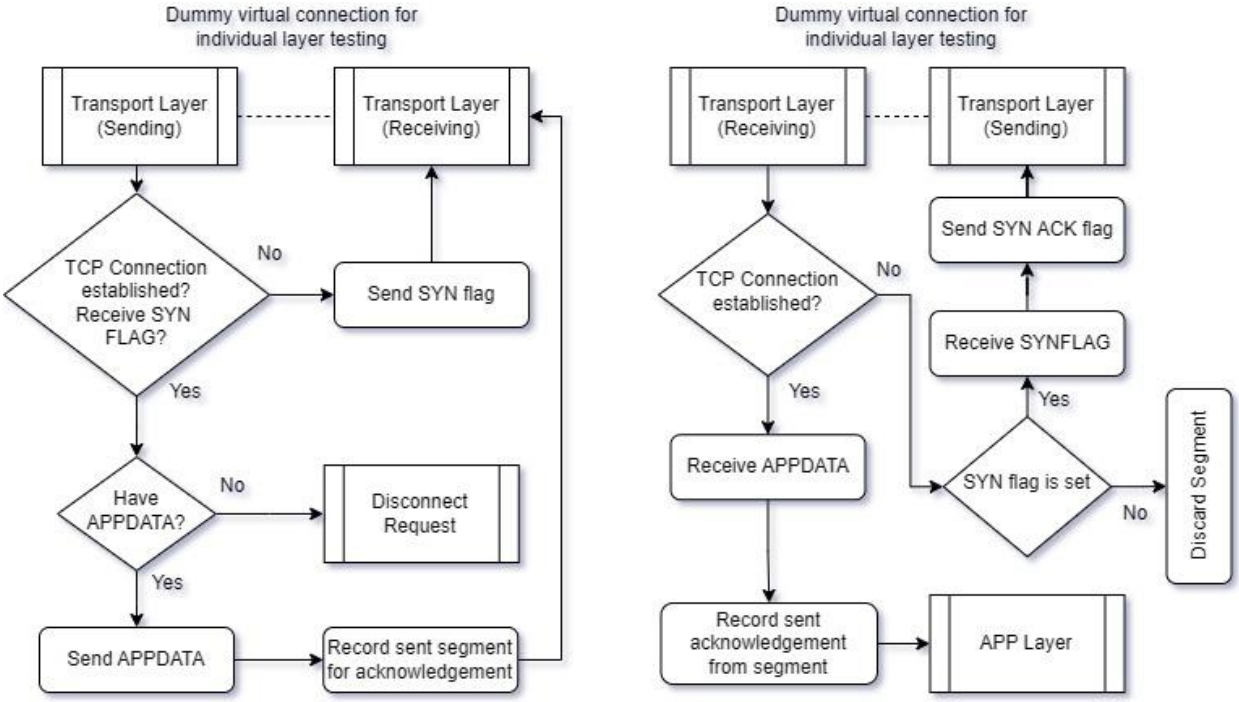
3. Segment Structure and Sliding Window Control: Our custom segment structure and sliding window control mechanisms are designed to be resource-efficient, taking into account the limitations of the Il Matto platform.

In summary, the choice of TCP over UDP for our smart lighting system is driven by TCP's reliability, error-checking, and ordered data transfer capabilities, which are essential for the accuracy and stability of smart lighting control. This approach ensures that our design is not only practical but also robust and efficient, aligned with the specific constraints of the Il Matto platform.

The absence of **Bandwidth Allocation** or **Congestion Control** is deliberate, due to the periodic and bursty nature of the data rate, to reduce microcontroller load for improved power efficiency and less overhead. In rare cases of congestion and collision, CSMA p-persistent from DLL combined with TCP connection-oriented transport should minimize its impact.

On the other hand, for button or commands that intended for multiple recipients, this is achieved by issuing multiple TCP segments from Application layer sequentially or using flooding IP (0) as agreed in network layer to send to all connected device on the network. Thus, this eliminates the only drawback of using TCP instead of UDP while retaining reliability as main core features.

Flow Chart:



For integrated version, the virtual connection is achieved by passing the parameters to layer responsible next, this is further discussed in Layered architecture part in Implementation section.

Implementation

The implementation of our protocols on the Il Matto, a single-threaded microcontroller, required meticulous design strategies to ensure efficient execution across all layers of our network stack. Here's an overview of our approach:

1. Interrupt-Driven Design: Leveraging interrupts was crucial for real-time responsiveness. We utilized an Interrupt Service Routine (ISR) for Timer0 overflow in transport.c for periodic uart buffer refresh, USART0_RX_vect to detect special character '/' for sending commands and pin change on buttons. This approach allows the microcontroller to immediately respond to events, while maintaining normal program execution otherwise (idle listen).
2. Decoupled Layer Codes: Our code was structured into distinct modules, each representing a layer in the stack. Each Layer is in its own .c and have its own .h, this is to simplify integration later, with use of Make and Makefile.
3. Layered Architecture Adherence: Each module both provides an interface for inter-layer communication and implements specific functionalities. Particularly in the integration, we utilized state machine flags (such as NextFunction/LastFunction) and pointer references to directly manipulate segments, application data, and target IP addresses for network layer interaction, refer to E1main.c. The exception to this is between Application/Transport and DLL/Physical as both are implemented by same person each so personal choice is tolerated.
4. Modularity and Code Reusability: Keeping the code for each layer in separate modules not only clarified our implementation but also enhanced reusability. The modular design means that our transport layer code, for instance, could be adapted for different network layers, provided they conform to the same interface. I also wrote custom library functions for application layer interfaces such as uart.h, led.h and button.h so its easier to synchronize changes while minimizing memory usage by reducing multiple different libraries.

This implementation strategy was tailored to align with the capabilities and constraints of the Il Matto microcontroller, ensuring that our smart lighting system was not only functional but also efficient and adaptable.

I also advocated for the approach of using bit manipulation instead of string or array manipulation for most of the code algorithm and memory as it uses less memory while also conforming to the communication protocol we are designing which is bit by bit defined. The usage of unpacked struct helped a lot in making sure no unnecessary padding causes corruption or bugs.

Code management

Our project's implementation leveraged Git and a private GitHub repository for version control, facilitating efficient collaboration and code management. This approach allowed team members to work on separate aspects simultaneously without overlap, thanks to branching.

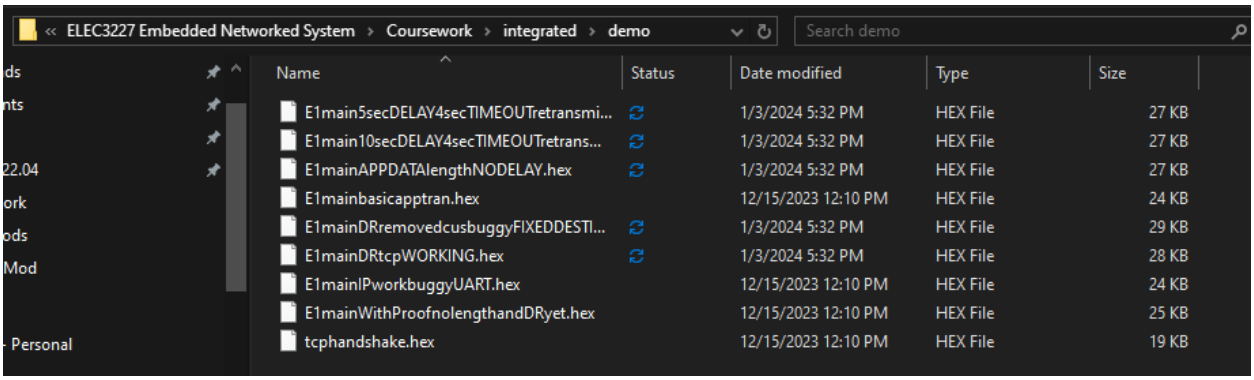
I adopted a bottom-up development approach, focusing initially on basic functionalities before advancing to more complex features. This strategy ensured a stable foundation, allowing for incremental improvements and efficient debugging. Regular commits to Git provided a reliable record of progress and a fallback option in case of any major issues.

The Git commit history reflects this systematic development process, demonstrating how we managed to build a complex TCP protocol incrementally without introducing critical bugs or errors.

Results and Analysis

Challenges in Windows 10 OS:

Installed Scoop.sh to get latest avr-gcc instead of WinAVR20100110 that's outdated and also the installer messes up Path environment variables. This makes it easier to manage version of compiler (avr-gcc), make and avrdude to flash. I rewrote some of the Makefile for my personal testing so its easier to see whats happening under the hood. However, to check the size of program especially in terms of SRAM occupied, avr-size from binutils-avr is needed but its only available for Linux OS distro, so I used WSL (Windows Subsystem for Linux) which allows me to access it without needing to use more cumbersome and laggy GUI based virtual machines or dual booting into Linux which would jeopardize my workflow. After setup is done, development and testing is pretty streamlined as I did unit testing frequently right after finishing a features and saved the .hex in separate demo folder and .elf files in git for easier access.



UART latency

One of the immediate patterns I recognize is that the UART debug output using PuTTy causes very high latency as it is a blocking function. This causes problem in benchmarking the latency response of my protocol algorithm as debug output is also needed to see if its working correctly.

One method is to compile and run the TCP layer .exe separately using gcc instead of avr-cc however that would not be accurate and inconclusive as it is now running in a different hardware (dedicated PC) instead of the avr microcontroller which is the target device. Having access to better debugging hardware and software such as AVR dragon might help but for now a simple traditional method suffices, in which I commented off the uart message code after verifying the tcp layer is working, the results averages out to latency under 500 ms after command/button is sent/pressed which is acceptable.

Optimising ISR (Interrupt Service Routine) usage

Due to nature of ISR being blocking, lots of bugs and unexpected situation can occur if its triggered haphazardly without enough error catching net to deal with it, this can lead to freezes and crashes. One of which is found numerous times due to UART buffer overflow which causes PuTTy to freezes and needing a restart, this is not ideal, thus I have a Timer Overflow interrupt that refreshes the buffer every now and then to fix this. ISR calls when other process is running doesn't cause a bug as the layer procedures itself is blocking in nature and need to be finished before other functions outside can be called.

Main Features

All basic main features of Application and Transport layer using TCP is achieved as outlined in standard document, except for sliding window control which should make use of BUF flags but as there is no support

for APPDATA above 9 byte, thus defeats the purpose of having it in the first place. Following is the features mentioned with its more specific implementation result and analysis of TCP protocol:

1. **TCP handshake with Disconnect Request:** Following the flow chart and usual tcp handshake protocol, it is implemented successfully and tested using dummy virtual connection method. This allow for reliability as the sender only sends data when it knows it is connected to the intended receiver than blindly shooting in the dark and praying. Utilised the SYN, DR and ACK flags in segment headers.
2. **Sequence number and order:** This is achieved thanks to the allocated bit segments, using simple bit masking and operations. A Error handling is used and called whenever a mismatch occurs, utilizing the SEQ NUM flag in segment headers.
3. **Acknowledgment**: This is vital for previous system as it is dependent on this, utilized the ACK NUM flag in segment headers.
4. **Timeout/Lost Retransmission:** Reusing the same timer overflow interrupt for UART refresh, awaiting_ack/sent_segment are matched to see if there is missing acknowledgment which triggers retransmission.
5. **Buffer:** Simple buffer for received messages which works on first in first out (FIFO) basis when full.
6. **Error Checking:** Slightly complex but effective XOR Checksum is used on the whole segment including APPDATA. This method can detect all single-bit errors and most multi-bit errors, but it's not foolproof and can miss some types of errors.

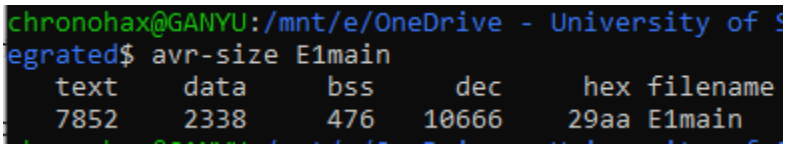
XOR Checksum analysis

The justification behind using of XOR checksum is its simpler implementation despite more computation required. It is also more robust considering the data value I'm working with where most of byte is either 0x00 or 0xFF. In cases where there is multi-bit errors XOR checksum cant detect, using simple majority vote on some of the segment headers field, I can easily deduce the real value and correct the error. This can also utilize the whole 2 byte of checksum field better.

Integration

Integration is successful with network layer, in which I just created another virtual network connection pass to mimic successful DLL/PHY layer. However, unfortunately integration with DLL/PHY results into unsolvable compiler/make error, most likely due to our unfamiliarity in integrating the provided rfm12b library properly.

AVR-size analysis



Using avr-size on .elf (E1main), and not the .hex as this doesn't contain relevant compiler information, the total SRAM taken by integrated Application, Transport and Network layer is still below 4kb.

Critical Reflection and Evaluation

Reflecting on our project, I realize there were several key improvements that could have made our process more efficient. One of the main takeaways for me is the benefit of jumping into hands-on coding earlier. Spending less time initially on detailed planning and standards and more on actual coding would have given us a better sense of our practical limits and strengths, leading to more realistic planning.

I also see now how crucial it is to get familiar with tools like avr-size early in the project. This tool could have been a great help in optimizing our code and managing memory from the outset, and I wish we had incorporated it sooner into our workflow.

My approach to measuring latency was quite basic, primarily involving the commenting out of UART messages. Looking back, investing in a more sophisticated method for this would have provided us with much more accurate and useful data, enhancing our performance evaluation.

Another area for improvement was in the integration of our system components. Starting this process earlier, even with just the basic features, and then building upon it gradually, could have streamlined our development significantly akin to Agile software development methodologies. It's something I'll certainly bear in mind for future projects.

In sum, this project taught me the value of early practical coding, the strategic use of development tools from the beginning, and the importance of early system integration for a more efficient development lifecycle.