**Week 1**

**A / Class Discussion Evidence**

**1. What is Scalability?**

**Definition:**

* Scalability refers to a system's ability to handle increased workload or expand its capacity without compromising performance, reliability, or efficiency.
* Key Characteristics**:**
  + **Horizontal Scaling:** Adding more nodes/devices (e.g., cloud servers).
  + **Vertical Scaling:** Upgrading existing hardware (e.g., CPU/RAM).
  + **Elasticity:** Dynamic scaling based on demand (common in cloud computing).

**2. Scalability in Computing and IoT**

* **In Computing:**
  + **Cloud Computing:** AWS/Azure auto-scaling for variable workloads.
  + **Databases:** Sharding (MongoDB) or replication (PostgreSQL).
* **In IoT:**
  + **Device Scalability:** Supporting thousands of sensors (e.g., smart cities).
  + **Network Scalability:** Protocols like MQTT, CoAP for low-overhead communication.
  + **Data Scalability:** Edge computing to reduce cloud dependency.

**Why It Matters for IoT:**

1. **Massive Device Growth:** 50B+ IoT devices by 2030
2. **Real-Time Processing:** Latency-critical apps (e.g., autonomous vehicles).
3. **Resource Constraints:** Limited power/budget in edge devices.

A graph with purple lines and text

AI-generated content may be incorrect.

Picture 1 – A mind map show the relation of information related to Scalability in IoT.

B / Group Activity Evidence

1. What Internet-of-Things devices, applications or services do you use at home?

* Devices: Smart watch, smart fridge.
* Applications: Apple’s Health app.
* Services: Tracking user info (heartbeat to measuring circadian clock, motion to detect steps, …), detect environment light intensity (to decide whether turning up the light), detect the temperature / moisture to adjust as according.

1. What Internet-of-Things devices, applications or services do you use at university, at work, when traveling, when outside your home?

* Devices: RFID/NFC card (for entering a building / using a lift / transportation), smartphone (GPS)
* Applications: Access Management System (responsible for opening the entrance door / lift usage), Google Maps (access and collect user real-time location to send to Google Maps platform)
* Services: Backend Server / Service (for handling open door / using lift request), Google Maps platform (handling user request for route, nearby places, …),

1. Quantify the amount of storage/data transmission/processing required for the following IoT applications.

For each, list the type of information that it collects. Examples include, key presses, favorite choices, location, distance travelled. Calculate the storage required for that application for a day/month/year.

A. Health Monitoring using a smart watch, heart rate monitor, and internet connected weight scales.

Type of Information:

Smart Watch:

* Heartbeat rate.
* Body temperature.
* Sleep / Circadian cycle.
* Accelerator (steps tracking).
* Bio-electrical impulses (ECG).
* Calories burned.

Heart Rate Monitor:

* Heartbeat rate.

Weight Scales:

* Weight.
* Body composition (fat %, muscle %, water %).

Calculated data storage information is based on DeepSeek references:

Smart Watch:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data Type | Size / Record | Records / Day | Daily | Monthly | Yearly |
| Heart Rate | ~ 0.1 KB | 1,440 | 144 KB (~ 0.14 MB) | 4.32 MB | ~ 53.00 MB |
| Body temperature | ~ 0.1 KB | 1,440 | 144 KB (~ 0.14 MB) | 4.32 MB | ~ 53.00 MB |
| Steps | ~ 0.05 KB | 24 | 1.2 KB | 36KB | 438 KB |
| Sleep Data | ~ 1 KB | 1 | 1.0 KB | 30 KB | 365 KB |
| ECG | ~ 5 KB | 1 | 5 KB | 150 KB | 1.8 MB |
| Total |  |  | ~ 0.3 MB | ~ 9 MB | ~ 108 MB |

Heart Rate Monitor:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data Type | Size / Record | Records / Day | Daily | Monthly | Yearly |
| Heart Rate | ~ 0.1 KB | 1,440 | 144 KB (~ 0.14 MB) | 4.32 MB | ~ 53.00 MB |

Weight Scales

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data Type | Size / Record | Records / Day | Daily | Monthly | Yearly |
| Weight + Body Composition | ~ 0.5 KB | 1 | 0.5 KB | 15 KB | 183 KB (~ 0.18 MB) |

B. Fleet management using IoT devices to track and manage vehicles in logistics and transportation.

Type of information:

* Vehicles registration number / type.
* Departure and Arrival time of a vehicle.
* Route history.
* Real-time GPS Location
* Mileage (Odometer)
* Driver identification name / number / driver license number.
* …

Rough estimation for data storage

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data Type | Size / Record | Records / Day | Daily | Monthly | Yearly |
| GPS Location | ~ 0.1 KB | 1,440 | 144 KB (~ 0.14 MB) | 4.32 MB | ~ 53.00 MB |
| Travel History | ~ 0.1 KB | 8,640 | 864 KB (~ 0.85 MB) | ~ 25.5 MB | ~ 301.00 MB |
| Engine Diagnostics | ~ 0.2 KB | 1,44 | 28.8 KB | ~ 864 KB (~ 0.85 MB) | ~ 10.2 MB |
| Cargo Information | ~ 0.1 KB | 24 | 2.4 KB | 72 KB | 864 KB (~ 0.87 MB) |
| Total for 1 vehicle |  |  | ~ 1.04 MB | ~ 31.2 MB | ~ 374.4 MB |

C. Smart Agriculture using IoT devices for precision farming, crop monitoring, and livestock management.

Type of information:

* Crop:
  + Pest / disease detection.
  + Growth stage.
  + Type.
  + Weather information: air temperature, humidity, rainfall, wind speed / direction.
  + Soil: composition – pH, nutrient level, moisture level, underground temperature.
* Livestock:
  + GPS tracker.
  + Health sensor: body temperature, heartrate, feeding schedule, ID number, vaccination history.

Rough estimation for data storage:

Crop Per Hectare

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data Type | Size / Record | Records / Day | Daily Storage | Monthly | Yearly |
| Soil Information | ~ 0.2 KB | 24 | ~ 4.8 KB | ~ 144 KB | ~ 1,728 KB (~ 1.8 MB) |
| Weather Data | ~ 0.2 KB | 24 | ~ 4.8 KB | ~ 144 KB | ~ 1.8 MB |
| Total data storage per hectare |  |  | ~ 10.0 KB | ~ 288 KB | ~ 4.0 MB |

Livestock (a animal / day)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data Type | Size / Record | Records / Day | Daily Storage | Monthly | Yearly |
| GPS Location | ~ 0.1 KB | 144 | ~ 14.4 KB | ~ 432 KB | ~ 5.26 MB |
| Bodily Information | ~ 0.05 KB | 24 | ~ 1.2 KB | ~ 36 KB | ~ 438 KB |
| Feeding schedule | ~ 0.1 KB | 24 | ~ 2.4 KB | ~ 72 KB | ~ 1.5 MB |
| Total data storage per animal |  |  | ~ 18.4 KB | ~ 600 KB (~ 0.6 MB) | ~ 8.0 MB |

1. Imagine a futuristic home in which everything is fully automated with Cloud-based IoT automations.

What would be the consequences if the internet service was cut off, and no local automation services exist.

A. The home air conditioning system

* Cutoff during the usage:
  + The air conditioning system will not be able to adjust the level of air flow / volume according to the changes in environment factors (moisture, air quality level, user activities …) while “temperature” is still allowing this system to function normally as traditional thermostat does not require internet connection or it will continue to release the cool / warm air given that the traditional thermostat system is being replaces with online automation services.
* Cutoff before any usage:
  + Like above, the system is still able to function normally thanks to thermostat given that this is a traditional thermostat instead of relying on internet connection to release or halt the release of cool / warm air. However, no local automation services exist then the air condition system is still to be able to function normally and require human intervention to manually adjust the temperature.

B. Security Access and Alarm System

* Cutoff during the usage
  + Tenants are still able to exist the building but not be able to re-access it because Security Access is not able to send user information (access key card) to the internet to request permission to enter the premise.
  + The Alarm System is not online / working as environmental data extracted from sensors are being kept locally instead of sending to backend server for processing and issue back the appropriate commands accordingly.
* Cutoff before any usage
  + Tenants will not be able to enter the building.
  + The Alarm System in this case will share similar consequences as above.

C. Smart fire, gas, carbon monoxide monitoring system

* Cutoff during the usage:
* The system is still able to pick up data from connecting sensors but will not be able to any adjustment accordingly as no connection is found for this system to send this information to the backend server and receive the instructions back from this latter.
* Cutoff before any usage:
* Like above.

C / Technical Task Evidence

Testing the client-server relationship

A screen shot of a computer program

AI-generated content may be incorrect.

Picture 1 - Server-side result.

A screenshot of a computer

AI-generated content may be incorrect.

Picture 2 – Client-side result.

Experiment with code

A/ The interval of sending the data (repeat the comment every 2 seconds)

A screenshot of a computer

AI-generated content may be incorrect.

Picture 3 – Result from Server-side.

A black screen with white text

AI-generated content may be incorrect.

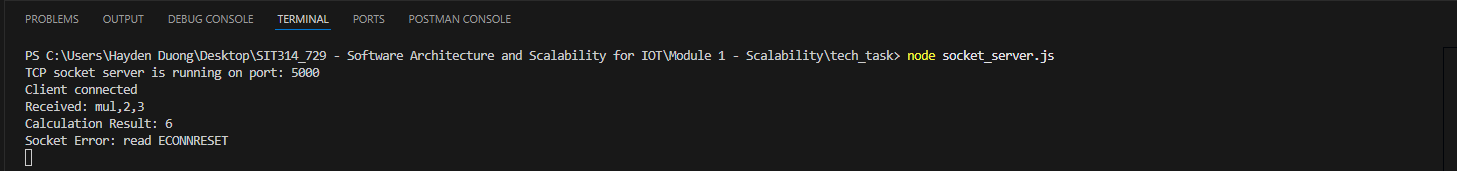
Picture 4 – Result from Client-side.

B/ Modify the server to support different calculations

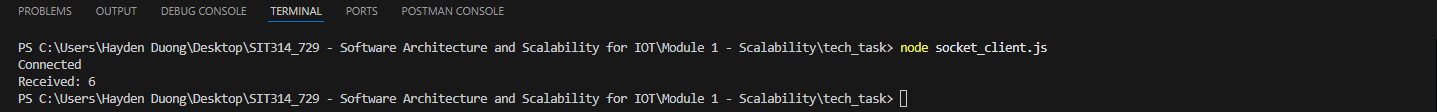
A screenshot of a computer

AI-generated content may be incorrect.

Picture 5 – Code-addition to socket\_server.js



Picture 6 – Result from Server-side



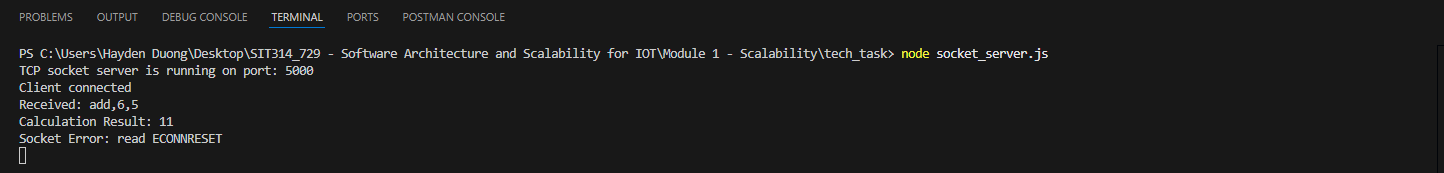
Picture 7 – Result from Client-side

C/ Modify the client code so it reads a random number and sends it as part of the calculation

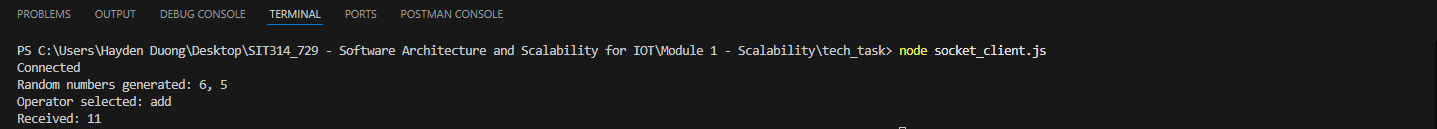
A screenshot of a computer

AI-generated content may be incorrect.

Picture 8 – Code addition & modification to socket\_client.js



Picture 9 – Result from Server-side

Picture 10 – Result from Client-side

D/ Number of clients

A screen shot of a computer

AI-generated content may be incorrect.

Picture 11 – Created a new JS file called stress\_test.js to simulate several clients are connecting to socket\_server.js (code suggested by DeepSeek)

A screenshot of a computer

AI-generated content may be incorrect.

Picture 12 – Result from Server-side

A black screen with white text

AI-generated content may be incorrect.

Picture 13 – Result from Client-side

E/ Connect between other computers (need to share stress\_test.js for other group members to achieve different IP addresses display in the server-side terminal – tested with)

A screenshot of a computer

AI-generated content may be incorrect.

Picture 14 – Result from Server-side

A screen shot of a computer

AI-generated content may be incorrect.

Picture 15 – Result from Client-side

A black screen with white text

AI-generated content may be incorrect.

Picture 16 – Result from the Server-side when another team member used my “stress\_test.js” to test connection to the “socket\_server.js”

F/ Put Pressure on Server

A screenshot of a computer program

AI-generated content may be incorrect.

Picture 16 – Result from Server-side

A black screen with many small colored spots

AI-generated content may be incorrect.

Picture 17 – Result from Client-side.

Week 2

**A / Class Discussion Evidence**

**IoT Nodes**

**1. What capabilities does an IoT node need?**

* **Sensing/Actuating:** Measures environmental data (temperature, humidity, motion) or performs actions (turning devices on/off).
* **Processing:** Basic data computation (microcontrollers, SoCs).
* **Communication:** Network connectivity (Wi-Fi, Bluetooth, LoRa, NB-IoT, etc.).
* **Power Management:** Energy efficiency (battery, solar, energy harvesting).
* **Security:** Data protection (encryption, authentication).

**2. What hardware components does an IoT node need?**

* **Sensors/Actuators:** Collect data or interact with the environment.
* **Microcontroller (MCU):** Processes data (e.g., ESP32, Arduino, Raspberry Pi Pico).
* **Communication Module:** Wi-Fi, BLE, Zigbee, LoRa, or Cellular (4G/5G).
* **Power Source:** Battery, solar panel, or wired connection.
* **Memory:** Temporary data storage (RAM/Flash).

**3. Identify some commercially available IoT nodes.**

* **Raspberry Pi (Compute Module 4):** For complex IoT applications.
* **ESP32/ESP8266:** Low-cost, supports Wi-Fi/BLE.
* **Arduino MKR Series:** Supports LoRa, NB-IoT.
* **STM32 B-L4S5I-IOT01A:** Multi-protocol IoT development board.
* **Commercial IoT Devices:** Xiaomi Mi Temperature Sensor, Philips Hue Smart Lights.

**Communications**

**1. What do we need to consider in IoT communications?**

* **Range:** Short-range (Bluetooth) vs. long-range (LoRa, Cellular).
* **Power Consumption:** Battery-powered devices need low-power protocols (NB-IoT, Zigbee).
* **Bandwidth:** Small data (sensors) vs. large data (video streaming).
* **Latency:** Real-time applications (industrial IoT) require low latency.
* **Security:** Encryption to prevent cyberattacks.

**2. What IoT communication technologies exist?**

* **Short-range:**
  + **Wi-Fi** (ESP32 for high-speed data).
  + **Bluetooth/BLE** (wearables, beacons).
  + **Zigbee** (smart home devices like Philips Hue).
* **Long-range (LPWAN):**
  + **LoRa** (agriculture, smart cities).
  + **NB-IoT** (low-power, wide-area coverage).
  + **LTE-M** (4G/5G for mobile IoT).

**3. Identify IoT nodes with different communication technologies.**

* **LoRa:** **Dragino LoRa Node** (environmental monitoring).
* **NB-IoT:** **Quectel BC66** (smart city devices).
* **Zigbee:** **Xiaomi Aqara Sensors** (home automation).
* **Wi-Fi/BLE:** **ESP32 DevKit** (prototyping).

**Cloud Computing for IoT**

**1. How does Cloud computing support IoT?**

* **Data Storage & Processing:** Handles massive IoT data (Big Data).
* **Scalability:** Expands as more devices connect.
* **Real-time Analytics:** Processes live data (e.g., predictive maintenance).
* **AI/ML Integration:** Enables smart analytics (e.g., image recognition from IoT cameras).
* **Remote Management:** Controls devices via cloud platforms (AWS IoT Core, Azure IoT Hub).

**2. Key differences between public, private, and hybrid cloud**

|  |  |  |  |
| --- | --- | --- | --- |
| Cloud Type | Definition | Pros | Cons |
| Public | Shared cloud services (AWS, Azure, Google Cloud). | Low cost, easy scheduling. | Less customization, security concerns. |
| Private | Dedicated cloud for one organization (on-premise/hosted). | High security, customizable. | Expensive, requires maintenance. |
| Hybrid | Combines public + private clouds | Flexible, balanced cost/security | Complex integration |

B / Group Activity Evidence

Consider the following applications and answer the questions about the most appropriate technologies to use.

|  |  |
| --- | --- |
| 1. A smart home lighting system. | |
| What properties does your IoT node need? | * Low power consumption. * Real-time control. * Local wireless connectivity. * Sensor (motion, light). |
| What microcontroller would you use? | * ESP32 / ESP8266. * Cheap and common. * Wi-Fi integrated. * Able to handle the signal and communicate with server. |
| What communications standard would you use? | * Wi-Fi. |
| What edge computing infrastructure would you use? | * Local Gateway: * Raspberry Pi. * Home Control Hub |
| Would you use cloud computing? | * Yes & No. |

|  |  |
| --- | --- |
| 1. A forest fire monitoring system. | |
| What properties does your IoT node need? | * Low power (on Battery / Solar). * Long-range communication. * Environmental factor sensors. * High durability. |
| What microcontroller would you use? | * STM32 / Arduino MKR WAN 1310. |
| What communications standard would you use? | * LoRa Wan – Long-range communication (up to 10Km) |
| What edge computing infrastructure would you use? | * Local LoRa Gateway. |
| Would you use cloud computing? | * Yes |

|  |  |
| --- | --- |
| 1. A smart air-conditioning system for a large building. | |
| What properties does your IoT node need? | * Real-time environmental sensors (temp, CO2, moist). * Manually controllable (Hybrid: Automate + Manually). |
| What microcontroller would you use? | * ESP32 * Wi-Fi integrated |
| What communications standard would you use? | * Wi-Fi – for within floor. * Bluetooth Low Energy (BLE) – for within a small area. * Ethernet – for between floors |
| What edge computing infrastructure would you use? | * Building Management System. |
| Would you use cloud computing? | * Yes. * Analyze power-usage. * Predictive maintenance. |

|  |  |
| --- | --- |
| 1. A river monitoring system for the remote Australian outback. | |
| What properties does your IoT node need? | * Solar-powered. * Waterproof. * Long-range communications. * Environmental factor sensors (water level, quality). |
| What microcontroller would you use? | * STM32. * Arduino MKR WAN 1310. * Adafruit Feather M0 + LoRa. |
| What communications standard would you use? | * LoRa WAN. * NB-IoT. |
| What edge computing infrastructure would you use? | * Remote Gateway. * Local LoRa Gateway |
| Would you use cloud computing? | * Yes. * Observe water-level for flood-warning. |

|  |  |
| --- | --- |
| 1. A driver-less taxi system for a smart city. | |
| What properties does your IoT node need? | * High-speed processing. * Real-time sensors (camera, GPS). * Reliable & fast networking. |
| What microcontroller would you use? | * Raspberry Pi 5. * Support Cellular 5G |
| What communications standard would you use? | * 5G. |
| What edge computing infrastructure would you use? | * On-board Edge Server – within the vehicle. * Roadside Edge Server. |
| Would you use cloud computing? | * Yes. * Fleet-management. |

|  |  |
| --- | --- |
| 1. A system for automatic robotic maintenance on Mars. | |
| What properties does your IoT node need? | * Radiation tolerance and high durability. * Autonomy. * Strong and Reliable Communication. |
| What microcontroller would you use? | * RAD-tolerant MCU. * TI TMS570 |
| What communications standard would you use? | * UHF / VHF. * X-band. * Delay Tolerant Networking (DTN). |
| What edge computing infrastructure would you use? | * Local AI. * Self-Decision-making system. |
| Would you use cloud computing? | * No. * Due to high latency between Earth and Mar. |

C / Technical Task Evidence

**IoT Weather Warning System - Technical Documentation**

This document details the specific modifications and additions made to the IoT Weather Warning System's components, explaining the reason behind each change.

**Fire Data Node (fire\_node.js – Code Snippet 1) - New Implementation**

The fire\_node.js was introduced as a new component to fetch real-time fire danger ratings from the Country Fire Authority (CFA) Victoria and send them to the central Weather Service.

**Key Implementation Details and Reasoning:**

* **Fetching RSS Feed:** We used the rss-parser library to retrieve the XML RSS feed from https://www.cfa.vic.gov.au/cfa/rssfeed/tfbfdrforecast\_rss.xml.
  + **Reasoning:** rss-parser simplifies the process of parsing RSS/Atom feeds into manageable JavaScript objects, allowing us to easily access feed items.
* **Handling Status code 406 Error:** Initially, requests to the CFA RSS feed resulted in a Status code 406 (Not Acceptable) error. This was resolved by configuring rss-parser to send standard HTTP headers.
  + **Explanation:** Many web servers, including those hosting RSS feeds, inspect HTTP request headers (like User-Agent and Accept) to determine if the client is a legitimate browser or an automated script. By sending common browser-like headers, we made our requests appear valid to the CFA server.
* **Correcting latestItem.description to latestItem.content:** After resolving the 406 error, a new issue arose where cheerio.load() expected a string, but latestItem.description was undefined. Debugging revealed that rss-parser mapped the RSS item's <description> XML tag to the content property of the parsed JavaScript object, not description.
  + **Explanation:** rss-parser has conventions for how it maps XML elements to JavaScript object properties. In this case, the main HTML content within the <description> tag was made available via the content property.
* **Parsing HTML with cheerio:** The extracted content (which is an HTML string) contains the fire danger ratings. cheerio was used to navigate this HTML structure.
  + **Reasoning:** cheerio provides a jQuery-like syntax for parsing and manipulating HTML, making it easy to find specific elements (like <p> tags) and extract their text or HTML content.
  + **Process:** We specifically looked for the <p> tag containing "Fire Danger Ratings", then extracted data from the *next* <p> tag, splitting it by <br> tags to get individual area ratings.
* **Data Transmission to Weather Service:** The extracted fire warning levels are formatted as a JSON object (e.g., {"Central": "NO RATING", "Mallee": "LOW"}) and sent via TCP socket to the Weather Service using the command fire,{"JSON\_DATA"}.
  + **Reasoning:** JSON is a lightweight and human-readable format for transmitting structured data, making it suitable for this purpose.
* **Periodic Updates:** The fetchFireWarning function is called every 5 minutes using setInterval.
  + **Reasoning:** Fetching too frequently (e.g., every few seconds) can lead to IP blocking or rate-limiting by the external RSS feed provider. A 5-minute interval is generally more polite and sustainable for public data sources.

**Weather Service Node (weather\_service.js – Code Snippet 2) - Modifications**

The central weather\_service.js was modified to correctly receive and process the new fire data from fire\_node.js and to handle request commands for localized warnings.

**Key Modifications and Reasoning:**

* **Correct JSON Payload Parsing:** The primary issue here was that the data.toString().split(',') method incorrectly split the incoming JSON string from the fire\_node.js.
  + **Explanation:** When fire\_node.js sends fire,{"Central":"NO RATING", ...}, the JSON string itself contains commas. A simple split(',') would break this JSON into multiple pieces (e.g., command[1] would be {"Central":"NO RATING", command[2] would be "East Gippsland":"NO RATING"}). This led to JSON.parse() receiving an incomplete and invalid string.
  + **Solution:** We implemented a more robust parsing method using indexOf(',') and substring() to ensure only the *first* comma separates the command name from the entire JSON payload.
* **Handling fire command:** A new case "fire": was added to the switch statement to process the incoming fire warning data.
  + **Explanation:** This case is responsible for parsing the rawCommandValue (which is now the full JSON string) into the fireWarningLevels object using JSON.parse(). This object then stores the fire danger ratings for each region.
* **Handling request command with Localization:** The request command was updated to accept an areaToRequest parameter.
  + **Explanation:** Previously, request might have been generic. Now, rawCommandValue after the split for a request command directly contains the area name (e.g., "Central"). The service uses this area name to look up the specific fire warning from the fireWarningLevels object and includes it in the response.

**Warning Request Node (warning\_request.js – Code Snippet 3) - Modifications**

The warning\_request.js client was modified to enable requesting warnings for specific localized areas.

**Key Modifications and Reasoning:**

* **Dynamic Area Request:** A new const areaToRequest variable was introduced.
  + **Explanation:** This allows the client to specify which Victorian region it wants to query (e.g., "Central"). This makes the request node more flexible and capable of simulating requests from different geographical locations.
* **Purpose:** This modification ensures that the system can demonstrate the delivery of **localized warnings**, which was a key requirement.

**Code of fire\_node.js, modified weather\_service.js, modified warning\_request.js**

// fire\_node.js

// This code implements a TCP socket client that connects to a weather service

// and fetches fire warning levels from an RSS feed, sending updates to the server.

const net = require("net");

const Parser = require("rss-parser");

const cheerio = require("cheerio");

const host = "127.0.0.1";

const port = 6000;

const CFA\_RSS\_FEED\_URL = "https://www.cfa.vic.gov.au/cfa/rssfeed/tfbfdrforecast\_rss.xml";

// Create a TCP client

const client = new net.Socket();

// Initialize variables to store fire warning levels

let fireWarningLevels = {};

// Function to fetch fire warning levels from the RSS feed

async function fetchFireWarning() {

    try {

        // Use rss-parser to fetch and parse the RSS feed

        // Ensure the parser is configured correctly

        // to handle the specific structure of the CFA RSS feed

        // Note: The CFA RSS feed may have specific headers or content types

        // that need to be handled, so we use a custom parser configuration

        let parser = new Parser({

            headers: {

                'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36',

                'Accept': 'application/xml, text/xml, \*/\*'

            }

        });

        let feed = await parser.parseURL(CFA\_RSS\_FEED\_URL);

        // Check if the feed has items

        // If items are present, process the first item (usually today)

        // If no items are found, set a default warning level

        if (feed.items.length > 0) {

            const latestItem = feed.items[0];

            // Use cheerio to parse the content of the latest item

            // and extract the fire warning levels

            // This assumes the content is HTML and contains the relevant information

            const $ = cheerio.load(latestItem.content);

            // Initialize fireWarningLevels to an empty object

            fireWarningLevels = {};

            // Find the paragraph that contains the "Fire Danger Ratings"

            const fireDangerRatingsHeaderP = $('p').filter(function() {

                return $(this).text().trim() === 'Fire Danger Ratings';

            });

            // If the header is found (or there is a sentence called "Fire Danger Ratings"), proceed to extract the ratings

            if (fireDangerRatingsHeaderP.length > 0) {

                // Find the next paragraph that contains the actual ratings

                // This assumes the ratings are in the next paragraph after "Fire Danger Ratings"

                const actualRatingsP = fireDangerRatingsHeaderP.next('p');

                if (actualRatingsP.length > 0) {

                    // Split the ratings by line breaks and process each line

                    // to extract area names and their corresponding ratings

                    // e.g., "Central: NO RATING<br>East Gippsland: NO RATING<br>Mallee: NO RATING<br>North Central: NO RATING<br>..."

                    const rawRatings = actualRatingsP.html();

                    // Split the raw ratings by line breaks and store them in an array

                    // This assumes the ratings are separated by <br> tags

                    // Outcome: ratingLines = ["Central: NO RATING", "East Gippsland: NO RATING", ...]

                    const ratingLines = rawRatings.split('<br>');

                    // Process each element of ratingLines to extract area names and ratings

                    // and store them in the fireWarningLevels object

                    ratingLines.forEach(element => {

                        // Trim whitespace and match the pattern "Area: Rating"

                        // e.g., "Central: NO RATING"

                        // Outcome: fireWarningLevels = { "Central": "NO RATING", "East Gippsland": "NO RATING", ... }

                        const trimmedElement = element.trim();

                        const match = trimmedElement.match(/^(.+?):\s\*(.+)$/);

                        // If the match is successful, extract area name and rating

                        // and store them in the fireWarningLevels object

                        // eg., match = ["Central: NO RATING", "Central", "NO RATING"] where:

                        // match[0] = "Central: NO RATING" (full match) comes from ^

                        // match[1] = "Central" (area name) comes from (.+?)

                        // match[2] = "NO RATING" (rating) comes from \s\*(.+)$

                        if (match && match.length === 3) {

                            let areaName = match[1].trim();

                            let rating = match[2].trim();

                            // fireWarningLevels["Central"] = "NO RATING"

                            fireWarningLevels[areaName] = rating;

                        }

                    });

                } else {

                    console.warn("Warning: Could not find the actual fire ratings paragraph.");

                    fireWarningLevels = { "ALL": "NO RATING - PARAGRAPH\_NOT\_FOUND" };

                }

            } else {

                console.warn("Warning: Could not find 'Fire Danger Ratings' header.");

                fireWarningLevels = { "ALL": "NO RATING - HEADER\_NOT\_FOUND" };

            }

            console.log("Fire warning levels fetched successfully:", fireWarningLevels);

        } else {

            console.log("No items found in the RSS feed.");

            fireWarningLevels = { "ALL": "NO RATING - NO\_ITEMS" };

        }

    } catch (error) {

        console.error("Error fetching fire warning levels:", error.message);

        fireWarningLevels = { "ALL": "ERROR\_FETCHING" };

    }

    // Send the fire warning levels to the server

    // Check if the client is connected before sending data

    // If the client is not connected, attempt to reconnect and send data

    if (client.readyState === 'open') {

        client.write(`fire,${JSON.stringify(fireWarningLevels)}`);

        console.log(`Sent: fire,${JSON.stringify(fireWarningLevels)}`);

    } else {

        console.log("Client is not connected, attempting to reconnect and send data.");

        // Attempt to reconnect if the client is not connected

        // This is a simple retry mechanism to ensure the client can send data

        // after reconnecting to the server

        if (client.connecting === false && client.destroyed === false) {

            client.connect(port, host, () => {

                console.log("Reconnected to Weather Service");

                client.write(`fire,${JSON.stringify(fireWarningLevels)}`);

                console.log(`Sent after reconnect: fire,${JSON.stringify(fireWarningLevels)}`);

            });

        } else {

            console.log("Client is already connecting or destroyed. Will try again next interval.");

        }

    }

}

// Connect to the weather service

client.connect(port, host, () => {

    console.log("Connected to Weather Service");

    fetchFireWarning();

    setInterval(fetchFireWarning, 300000);

});

// Handle incoming data from the server

client.on("data", (data) => {

    console.log(`Received from Weather Service: ${data.toString()}`);

});

// Handle socket error event

client.on("error", (error) => {

    console.error(`Client Error: ${error.message}`);

});

// Handle socket end event

client.on("close", () => {

    console.log("Connection closed");

});

Code Snippet 1 – fire\_node.js

// weather\_service.js

// This code implements a TCP socket server that listens for weather data updates

// and responds to requests for weather conditions.

const net = require("net");

const port = 6000;

// Initialize variables to store weather data

let temp;

let wind;

let rain;

let fireWarningLevels = {};

// Create a TCP server

const server = net.createServer((socket) => {

    console.log("Client connected");

    // Set the encoding for the socket

    socket.on("data", (data) => {

        const strData = data.toString().trim();

        console.log(`Received: ${strData}`);

        const firstCommaIndex = strData.indexOf(",");

        if (firstCommaIndex === -1) {

            console.error("Invalid command format: No comma found.");

            socket.write("error");

            return;

        }

        const name = strData.substring(0, firstCommaIndex);

        const rawCommandValue = strData.substring(firstCommaIndex + 1);

        // Process the command based on the name

        switch (name) {

            case "temp":

                temp = parseFloat(rawCommandValue);

                console.log(name + “ : “ + temp);

                result = "ok";

                break;

            case "rain":

                rain = parseFloat(rawCommandValue);

                console.log(name + “ : “ + rain);

                result = "ok";

                break;

            case "wind":

                wind = parseFloat(rawCommandValue);

                console.log(name + “ : “ + wind);

                result = "ok";

                break;

            case "fire":

                try {

                    fireWarningLevels = JSON.parse(rawCommandValue);

                    console.log("Received Fire Warning Levels:", fireWarningLevels);

                    result = "ok";

                } catch (error) {

                    console.error("Error parsing fire warning levels:", error);

                    result = "error";

                }

                break;

            case "request":

                const requestedArea = rawCommandValue;

                console.log(`Request for weather conditions in area: ${requestedArea}`);

                const currentFireWarning = fireWarningLevels[requestedArea] || "NO DATA";

                // Respond with the current weather conditions

                if(temp > 20 && rain < 50 && wind > 30){

                    result = `Weather Warning. Fire Warning Level: ${currentFireWarning}`;

                }

                else {

                    result = `Everything is fine. Fire Warning Level: ${currentFireWarning}`;

                }

                break;

        }

        // Send the result back to the client

        socket.write(result.toString());

    });

    // Handle socket end event

    socket.on("end", () => {

        console.log("Client disconnected");

    });

    // Handle socket error event

    socket.on("error", (error) => {

        console.log(`Socket Error: ${error.message}`);

    });

});

// Handle server errors

server.on("error", (error) => {

    console.log(`Server Error: ${error.message}`);

});

// Start the server and listen on the specified port

server.listen(port, () => {

    console.log(`TCP socket server is running on port: ${port}`);

});

Code Snippet 2 – Modified weather\_service.js to complement with fire\_node.js

// warning\_request.js

// This code implements a TCP socket client that sends a request for weather conditions

// and listens for responses from the server.

const net = require("net");

const host = "127.0.0.1";

const port = 6000;

// Specify the area for which to request weather conditions

// List of areas can be defined based on the fire warning levels:

// Central, East Gippsland, Mallee, North Central, North East, Northern Country, South West, West and South Gippsland, Wimmera

const areaToRequest = "Central"; // Specify the area for which to request weather conditions

// Create a TCP client that connects to the server

const client = net.createConnection(port, host, () => {

    console.log("Connected to Weather Service");

    // Set the encoding for the client

    setInterval(() => {

        client.write(`request,${areaToRequest}`);

        console.log(`Sent request for weather conditions in area: request,${areaToRequest}`);

    }, 2000); // Interval in milliseconds (2000ms = 2 seconds)

});

// Handle incoming data from the server

client.on("data", (data) => {

    console.log(`Received: ${data.toString()}`);

//    process.exit(0);

});

// Handle socket error event

client.on("error", (error) => {

    console.log(`Error: ${error.message}`);

});

// Handle socket end event

client.on("close", () => {

    console.log("Connection closed");

});

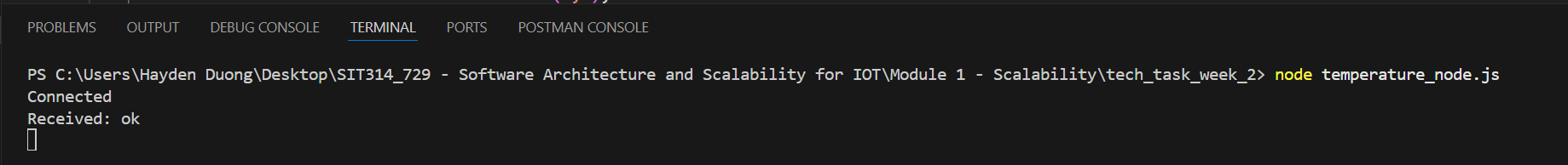
Code Snippet 3 – Modified warning\_request.js

Photo:

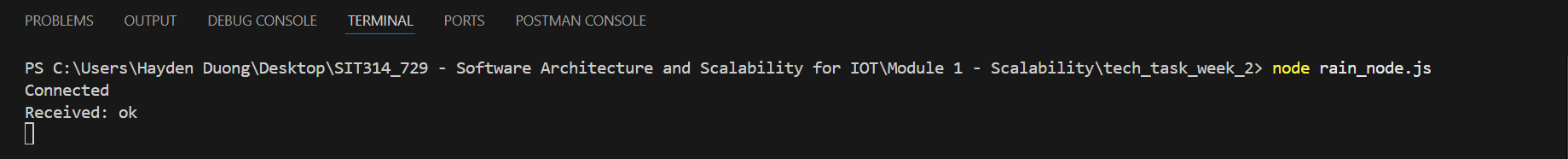
A screenshot of a computer

AI-generated content may be incorrect.

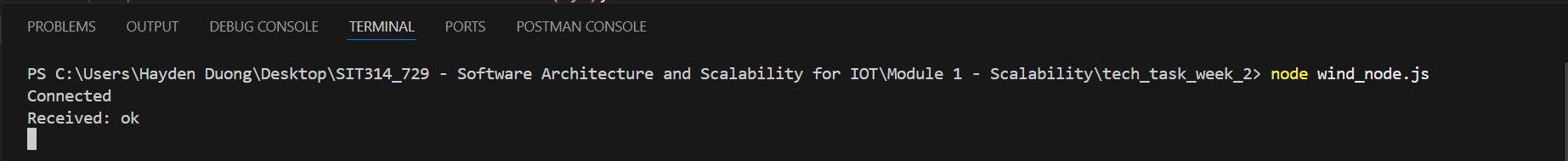
Picture 1 – weather\_service.js server received data from temperature\_node, rain\_node, wind\_node, fire\_node, and from warning\_request



Picture 2 – temperature\_node is successfully connected to weather\_service server.



Picture 3 – rain\_node is successfully connected to weather\_service server.

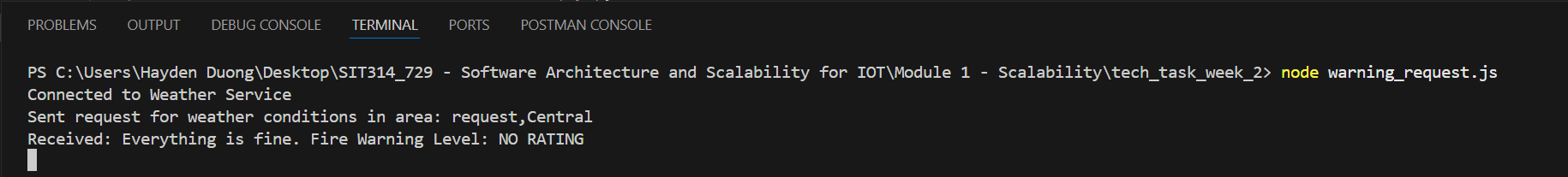


Picture 4 - wind\_node is successfully connected to weather\_service server.

A black screen with white text

AI-generated content may be incorrect.

Picture 5 – fire\_node is successfully connected to weather\_service server.



Picture 6 – warning\_request is successfully connected to weather\_service server and received warning result.

GitHub Link for code of both week 1 and week 2 technical task: <https://github.com/HaydenDuong/SIT314_729---Software-Architecture-and-Scalability-for-IOT/tree/main/Module%201%20-%20Scalability>

Module Summary:

From this module, I learned about the concept of IoT – what components make up of it, how there are interconnected / communicate with one another (from receiving the signals to response accordingly to those signals), the architecture (N-Tier) and different kinds of nodes that could present in a common IoT structure that we are use daily. These will serve as a strong foundation for me to prepare for the upcoming materials and knowledge in the following weeks to build up.