Simulation of Communication in Industrial Networks

CAN Network

Student: Mureșan Andrei-Ioan

Structure of Computer Systems Project

Technical University of Cluj-Napoca

January 15, 2025

Contents

Introduction

1.1 Context

The **CAN** (Controller Area Network) communication protocol is widely used in industries such as automotive, aerospace, and industrial automation for robust and efficient communication between various electronic devices. It operates as a multi-master, broadcast communication system, allowing multiple Electronic Control Units (ECUs) to communicate over a shared bus. CAN is favored for its noise immunity, real-time data handling, and fault tolerance, making it an essential part of modern embedded systems. [1]

However, while the **CAN protocol** provides the framework for low-level communication, higher-level protocols like CANopen are necessary to manage more complex operations, such as real-time data exchange, device configuration, and network management. CANopen builds on top of CAN, providing the necessary structure for distributed control systems by organizing data, commands, and system states using standardized messages such as **PDOs**, **SDOs**, and **NMT** commands.

In the automotive industry, for example, ECUs responsible for tasks such as engine control, anti-lock braking, and power steering communicate through the CAN bus. CANopen further organizes this communication by providing a standardized framework to manage real-time data, configuration parameters, and device states.

1.2 Objective

The objective of this project is to develop a simulation of a CAN communication system using **Express** and **React** and **p5** (UI). The simulation will model the behavior of multiple **ECUs** communicating over a CAN bus, implement message arbitration, and integrate CANopen protocol features for real-time data exchange, network management, and device configuration. The simulated environment will be a basic car.

1.3 Project proposal

This section states what features will be implemented in the final simulation of the CAN protocol. The details of each feature will be presented in future sections.

List 1.3.1: Specific goals

1. Simulating the low-level CAN bus communication protocol, including bit-level arbitration.

2. Implementing the CANopen application layer protocol which defines real-time data exchange via PDOs (Process Data Objects), configuration and diagnostics via SDOs (Service Data Objects), and network management using NMT (Network Management) messages, by the help of the Object Dictionary.

3. Creating a GUI using React and p5 that will enable the user to:
   - visualize the message transmission
   - display the state of the bus
   - display the states of the ECU
   - enabling users to interact by sending commands
   - configurating devices and adding new devices

Bibliographic Research

2.1 What is CAN?

CAN (Controller Area Network) is a serial multi-master, message broadcast communication protocol that was developed by BOSCH [1].

Its domain of application ranges from high-speed networks to low-cost multiplex wiring. In automotive electronics, engine control units, sensors, anti-skid-systems, etc. are connected using CAN with bitrates up to 1 Mbit/s. At the same time, it is cost effective to build into vehicle body electronics, e.g. lamp clusters, electric windows etc. to replace the wiring harness otherwise required. [2]

CAN operates at the **Physical Layer** and **Data Link Layer** of the OSI model, managing the transmission of messages and ensuring the integrity of data through mechanisms such as arbitration and error detection.

Remark 2.1.1: CAN vs CANopen

CANopen, is a higher-layer protocol built on top of CAN, providing a framework for how data is organized, exchanged, and managed in a network. It operates at the **Application Layer** and defines how devices interact. So, it is not the same thing as the **CAN** bus protocol, which operates at a lower level. [4]

2.2 The CAN bus

The bus in the CAN protocol refers to the physical connection that links multiple devices (ECUs) together, allowing them to communicate and exchange data. It consists of two wires: **CANH** (CAN High) and **CANL** (CAN Low), which form a twisted-pair cable.

The CAN bus operates using differential signaling, where the data is transmitted by the difference in voltage between the CANH and CANL wires, rather than an absolute voltage level. [2] This differential method makes the system highly resistant to external noise because any noise that affects both wires equally will be canceled out when the voltage difference is measured.

List 2.2.1: Bus states

- **Dominant State** (Logical 0): When the bus is in the dominant state, CANH is driven to a higher voltage (around 3.5V), while CANL is driven to a lower voltage (around 1.5V). This voltage difference (approximately 2V) represents a dominant bit, which is interpreted as a logical 0. [1]
- **Recessive State** (Logical 1): In the recessive state, both CANH and CANL are at the same voltage (around 2.5V), creating a 0V differential, which is interpreted as a logical 1. [1]

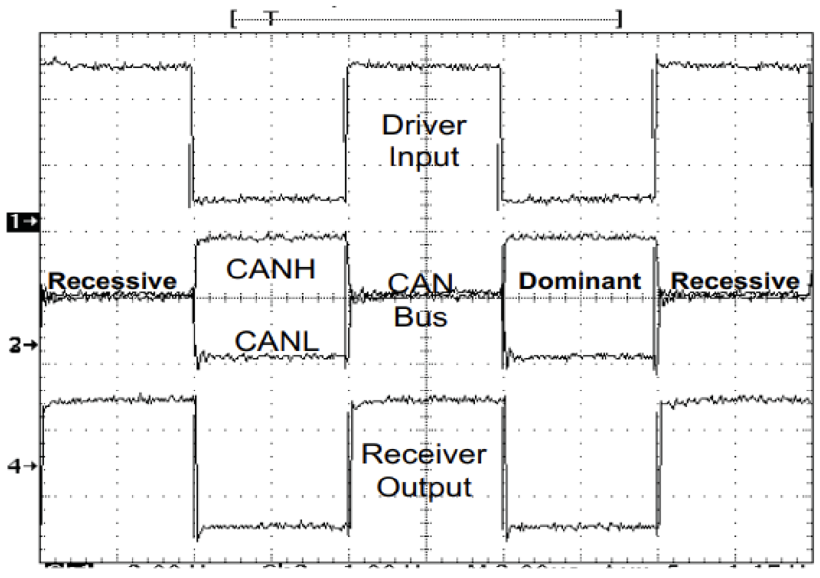


Figure 1. States of the bus under an oscilloscope [1]

The system defaults to the recessive state when no data is being transmitted. When an ECU sends data, it drives the bus into the dominant state for logical 0s, and the other ECUs can detect this change on the bus.

When two ECUs try to write a 1 and a 0 at the same time, the bus will be taken to the dominant state, the 0 winning the bus. This helps in arbitration.

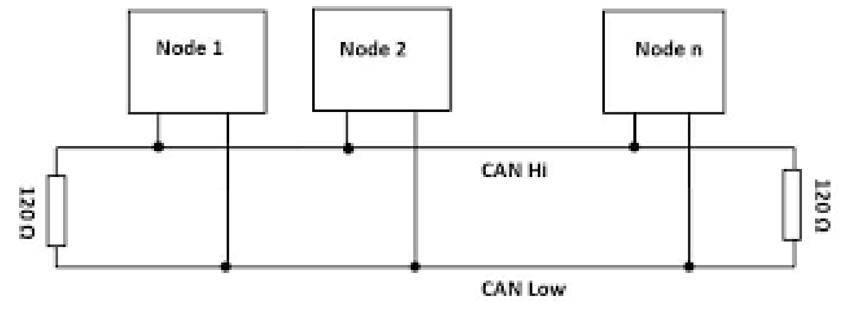


Figure 2. CAN bus [3]

2.3 Transceiver, CAN controller and DSP

A CAN device is made from multiple components, each with specific purposes.

The **CAN transceiver** is the component that interfaces between the **CAN controller** (operating at the Data Link Layer) and the physical bus, operating at the Physical Layer. Its main job is to convert the digital signals from the CAN controller into electrical signals that are transmitted over the CAN bus and to convert electrical signals from the bus back into digital data for the controller to process. Its key functionalities are signal conversion, bus monitoring, error handling and physical isolation. [1]

The **CAN controller** is the core component responsible for managing the **Data Link Layer** of the CAN protocol. It handles all communication-related tasks between the **DSP** (or microcontroller) and the **CAN transceiver**, ensuring that messages are correctly formatted, transmitted, and received according to the CAN protocol. Its key functionalities are message framing, message arbitration, error detection and message buffering. [1]

A **Digital Signal Processor (DSP)** is a specialized microprocessor optimized for performing high-speed mathematical operations, particularly useful in real-time applications. In the context of a CAN system, the DSP typically handles the higher-level processing tasks of an ECU, such as:

- Signal processing for sensors and actuators.
- Control algorithms for systems like engine control, motor drives, or industrial automation.

The **DSP** ensures that these real-time tasks are executed quickly and efficiently, without introducing delays into the system. It can process signals from sensors, apply complex control algorithms, and determine outputs that need to be sent to other devices on the CAN network. Although the DSP does not directly handle CAN message framing or transmission, it works alongside the **CAN controller** and **transceiver** to ensure smooth and efficient communication with other nodes in the network.

In *Figure 3* we can observe such configuration.

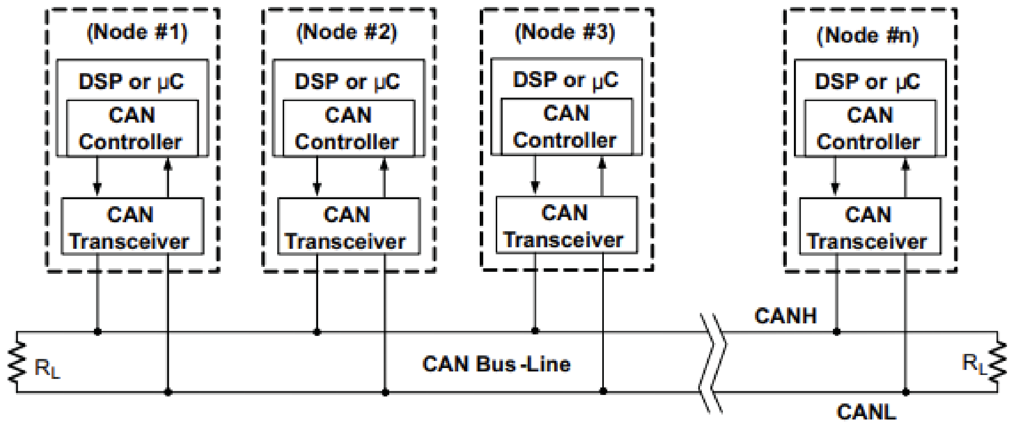


Figure 3. CAN system [1]

2.4 Arbitration

Arbitration in a **CAN** system is the process used to determine which Electronic Control Unit gets to send its message when multiple ECUs try to transmit simultaneously. CAN uses a non-destructive bitwise arbitration mechanism, meaning that the message with the highest priority (lowest **CAN-ID**) always wins without causing data loss.[2]

Remark 2.5.1: Low level explanation

During arbitration, each ECU monitors the bus while transmitting its message. If an ECU detects that another ECU is sending a dominant bit (0) while it is sending a recessive bit (1), it stops transmitting, as the bus has been taken by a higher-priority message. This process ensures that only one message is transmitted at a time and that the message with the highest priority is always sent first.[1]

2.5 CANopen

CANopen is a higher-layer communication protocol built on top of the CAN protocol, designed for embedded systems requiring structured data exchange, configuration, and network management. While CAN handles the low-level transmission of messages, CANopen adds organization and control at the Application Layer. [4]

At the core of CANopen is the **Object Dictionary**, a table that defines how each device's data and settings are structured. It enables standardization across the network, making it easier for devices to communicate and be configured.

CANopen defines two key message types:

- **PDOs** (Process Data Objects): Used for real-time data exchange, allowing devices to send and receive operational data like sensor readings and control commands quickly and efficiently.
- **SDOs** (Service Data Objects): Used for configuration and diagnostics, allowing one device to read or write to another device's Object Dictionary entries, enabling dynamic updates to device settings.

Additionally, NMT (Network Management) messages are used to control the operational states of devices, ensuring synchronized operation within the network.

CANopen is widely used in industries such as automotive, industrial automation, and medical devices, where real-time communication and device configuration are critical. It enhances CAN's capabilities by providing a flexible, structured approach to data management and device coordination. [4]

Analysis

3.1 Project Proposal

The final simulation for the CAN serial communication will provide a comprehensive visualization of multiple key aspects. It will demonstrate message arbitration, where several nodes attempt to send messages on the CAN bus at once. The simulation will reveal how conflicts are resolved based on priority, visually illustrating each bit as it is transmitted during the arbitration process, with a focus on dominant and recessive bits and their impact on the bus state.

Bitwise transmission will also be displayed, using a binary waveform to show changes in the bus state with each bit, making the influence of dominant bits clear. To deepen the realism, error detection and handling mechanisms, such as cyclic redundancy checks (CRC), acknowledgement errors, and bit-stuffing errors, will be included. Users will see how CAN manages errors, including the retransmission of faulty frames, adding a layer of robustness to the communication.

The simulation will cover various CAN frame types—data frames, remote frames, error frames, and overload frames—providing insights into each frame's role in the network. Fields within these frames, such as identifiers, data length, control, and CRC, will be highlighted to showcase their functions in message structuring. Users will also be able to adjust the bus load and observe the impact on message timing and delivery. Visual feedback will illustrate the influence of bus load on data flow, including latency effects on transmission.

Since the project includes CANopen protocol elements, the simulation will feature higher-level protocol functionalities like Process Data Objects (PDOs) and Service Data Objects (SDOs). This will enable the visualization of data transfers, command responses, and parameter configurations, key elements of CANopen communication. Additionally, the network management protocol (NMT) within CANopen will be represented, showing node states such as operational, pre-operational, and stopped, along with state transitions.

List 3.1.1: Simulation features

- See the top view of a wireframe car
- See a graphic representation of multiple ECUs, the CAN bus, the two resistors at each end of the bus, and each ECU connection to the bus
- See each ECU current state by hovering with the mouse over the drawing of a specific component
- Start the simulation
- Select configuration of the simulation. The configuration will be a set of commands that will specify the number of ECUs, what happens at each discrete time etc.
- Add a configuration
- Edit a configuration (e.g. add an ECU)
- See the log of the simulation
- See the resulting values of the simulation (e.g. total number of sent messages
- Visualize the current state of the bus in the means of voltage levels of each wire. A graph with the x axis as time and y axis will be drawn
- Visualize the arbitration process by seeing what components are trying to write on the bus at the same time

3.2 Functional and non-functional requirements

The following are the functional and non-functional requirements for the CAN simulator project. The **functional requirements** cover the core features, such as ECU and CAN bus visualization, configuration management, and real-time monitoring of simulation states. The **non-functional requirements** focus on usability, performance, compatibility, and reliability to ensure a smooth and effective simulation experience.

List 3.2.1: Functional and non-functional requirements

Functional:

- Top view of wireframe car - Display a top-down wireframe image of a car as a background or reference element in the simulation interface.
- ECU and CAN Bus Visualization - Render multiple ECUs, the CAN bus, and terminating resistors on both ends. Each ECU should connect visually to the bus.
- Simulation Start Function - Provide a button to initiate the simulation, updating all visuals in real-time.
- Control simulation time – either automatic time or step time. Transition to the next step will be controlled by a "Next step" button
- Configuration Management - Enable users to add and edit configurations, including modifying ECU counts and timing actions.
- Simulation Log Display - Show a real-time log of simulation actions, including message transmissions and state changes.
- Result Summary Display - Provide a summary view of key metrics, such as total messages sent, at the end of each simulation.
- Provide a summary view of key metrics, such as total messages sent, at the end of each simulation. - Graph voltage levels on the CAN bus over time, with the x-axis as time and y-axis as voltage.
- Arbitration Visualization - Visually represent the arbitration process, showing which components attempt to write on the bus at the same time.

Non-functional:

- Performance - Ensure smooth updates in real-time, even with multiple ECUs and high bus activity.
- Usability - Design an intuitive interface with clear icons, tooltips, and controls for easy interaction.
- Scalability - Support configurations with varying numbers of ECUs without performance degradation.
- Documentation - Offer brief, user-friendly documentation explaining each control and feature.

3.3 Use cases

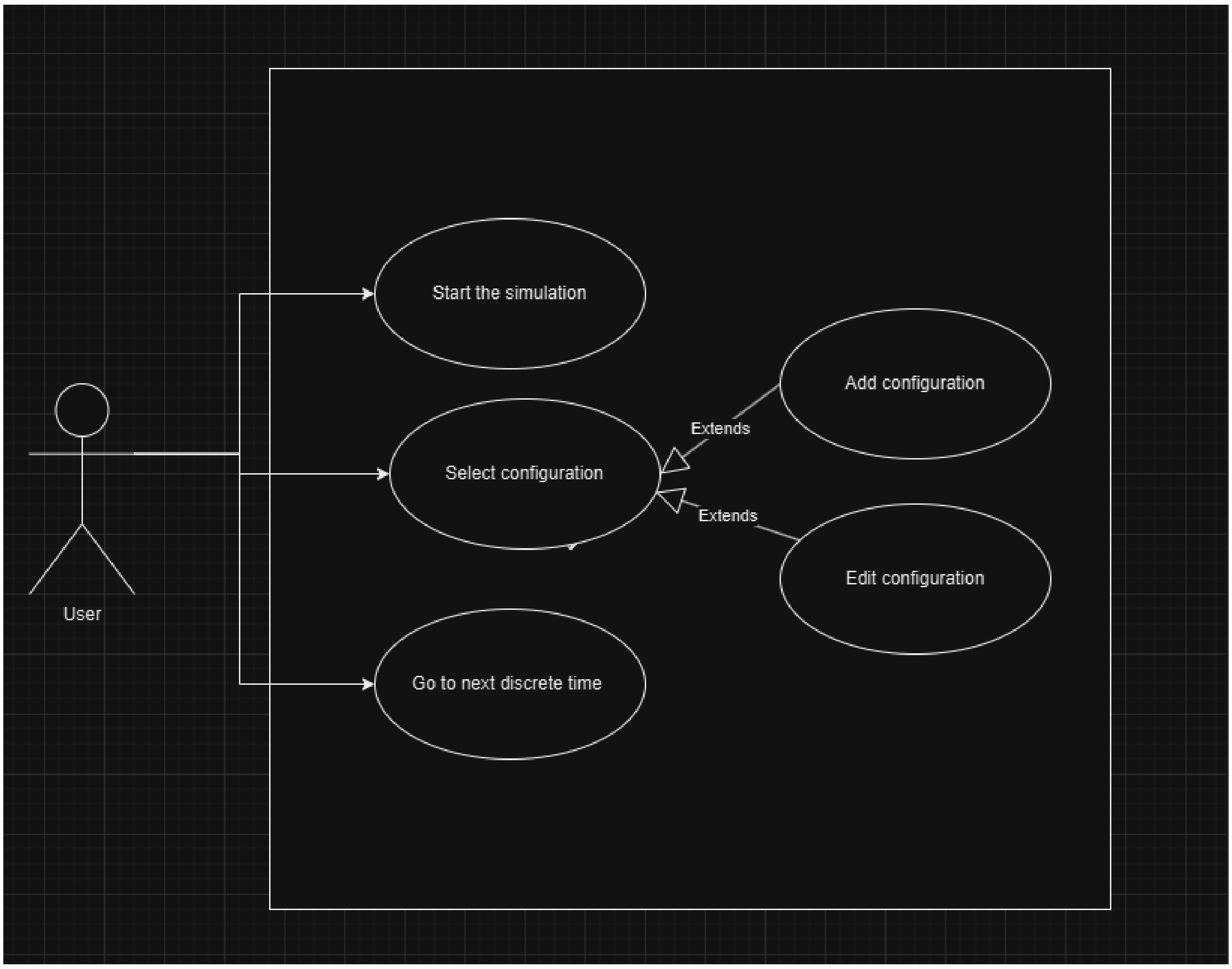


Figure 4. Use case diagram

Use case:

- **User**: The actor in the diagram is the user who interacts with the CAN simulator to control and monitor the simulation process.
- **Start the Simulation**: This use case allows the user to initiate the CAN simulation. Once started, the simulator will display the behavior of the configured network of ECUs, the CAN bus, and associated features.
- **Select Configuration**: The user can choose a predefined or previously saved configuration for the simulation. This selection will set up specific parameters, such as the number of ECUs, the timing of events, and the behavior of each component in the network. This is a primary step before starting the simulation.

    - **Add Configuration** (extends): This option allows the user to create a new configuration for the simulation. By adding a configuration, the user can specify details such as ECU count, actions, and discrete events. It extends the "Select Configuration" use case because it provides a way to set up configurations to choose from.
    - **Edit Configuration** (extends): This feature enables the user to modify an existing configuration. The user can adjust parameters, such as adding or removing ECUs and changing event timings. It also extends the "Select Configuration" use case by allowing adjustments to an existing configuration.
- **Go to Next Discrete Time**: This use case allows the user to advance the simulation to the next discrete time step manually. This feature is essential for observing the state of the network at specific intervals, providing detailed control over the simulation’s progression.

Design

4.1 Solution and architecture

The simulation will be provided via a web interface. The software will be created based on the client-server architecture. The client-server architecture will use **Express** as the backend server, **React** as the frontend framework, and **p5.js** integrated with React for visualizing the simulation. Communication between the client and server will be facilitated by a **RESTful API**. The server will handle simulation computations and send JSON files to the client, which the React frontend will then render for visualization.

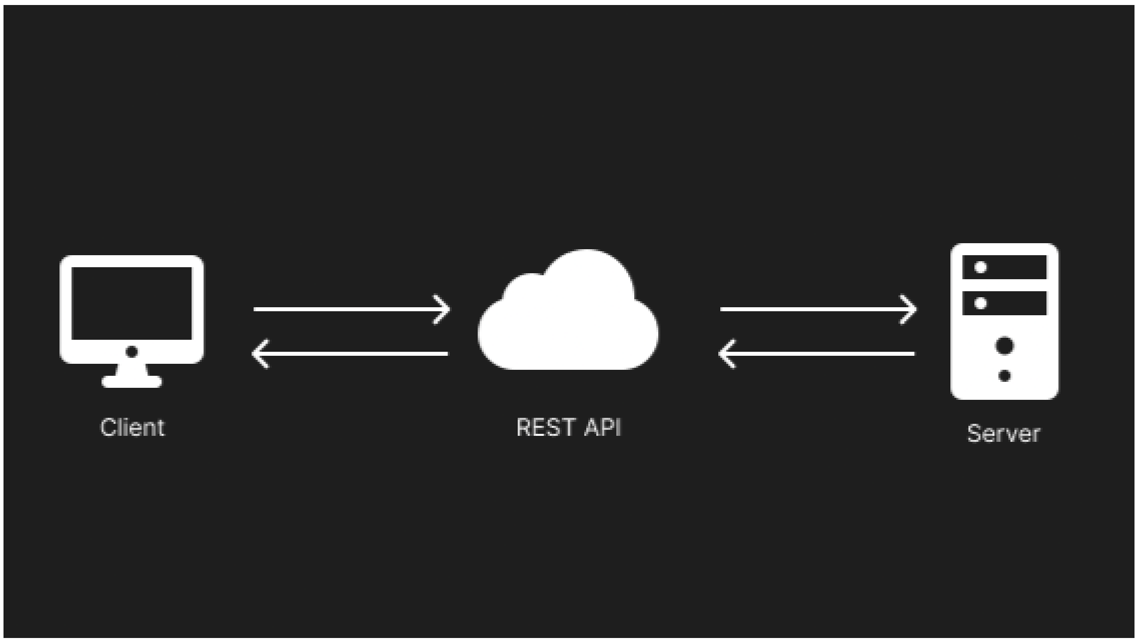


Figure 5. Client-server architecture [5]

4.2 Architecture Overview

1. Server (Backend):

- **Express** is used to set up a server to handle API requests from the client. This server will manage the configuration and execution of the CAN simulation, processing the results and storing them temporarily for retrieval by the client.
- The server will perform simulations in discrete time steps, computing values for each ECU and the CAN bus. These values are structured in JSON files, representing each time step's result, which the client can retrieve.

- **RESTful API**: The server exposes a RESTful API to allow the client to interact with the simulation, manage configurations, and fetch simulation results in JSON format.

2. Client (Frontend):

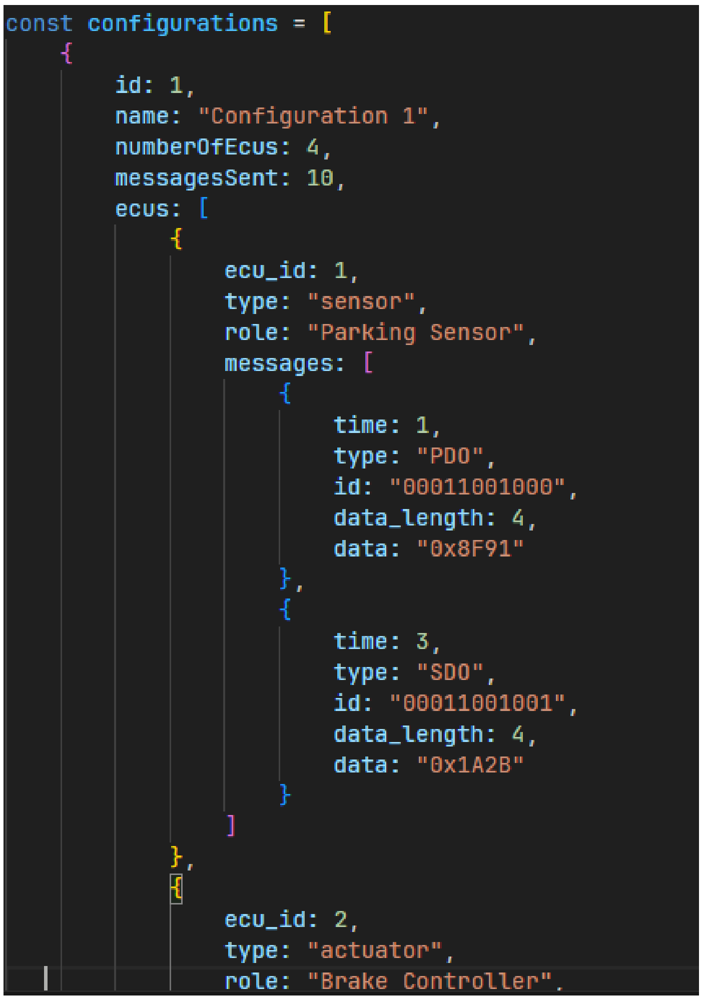
- **React** handles the user interface, providing options to start the simulation, select configurations, and view results.
- **p5.js** is integrated within React to create interactive visualizations. It visualizes each ECU's state, the bus's voltage levels, and the arbitration process based on the data fetched from the server.
- The client sends requests to the backend server to fetch simulation results and configuration data, displaying them dynamically using p5.js sketches embedded in the React components.

3. API endpoints and HTTP methods:

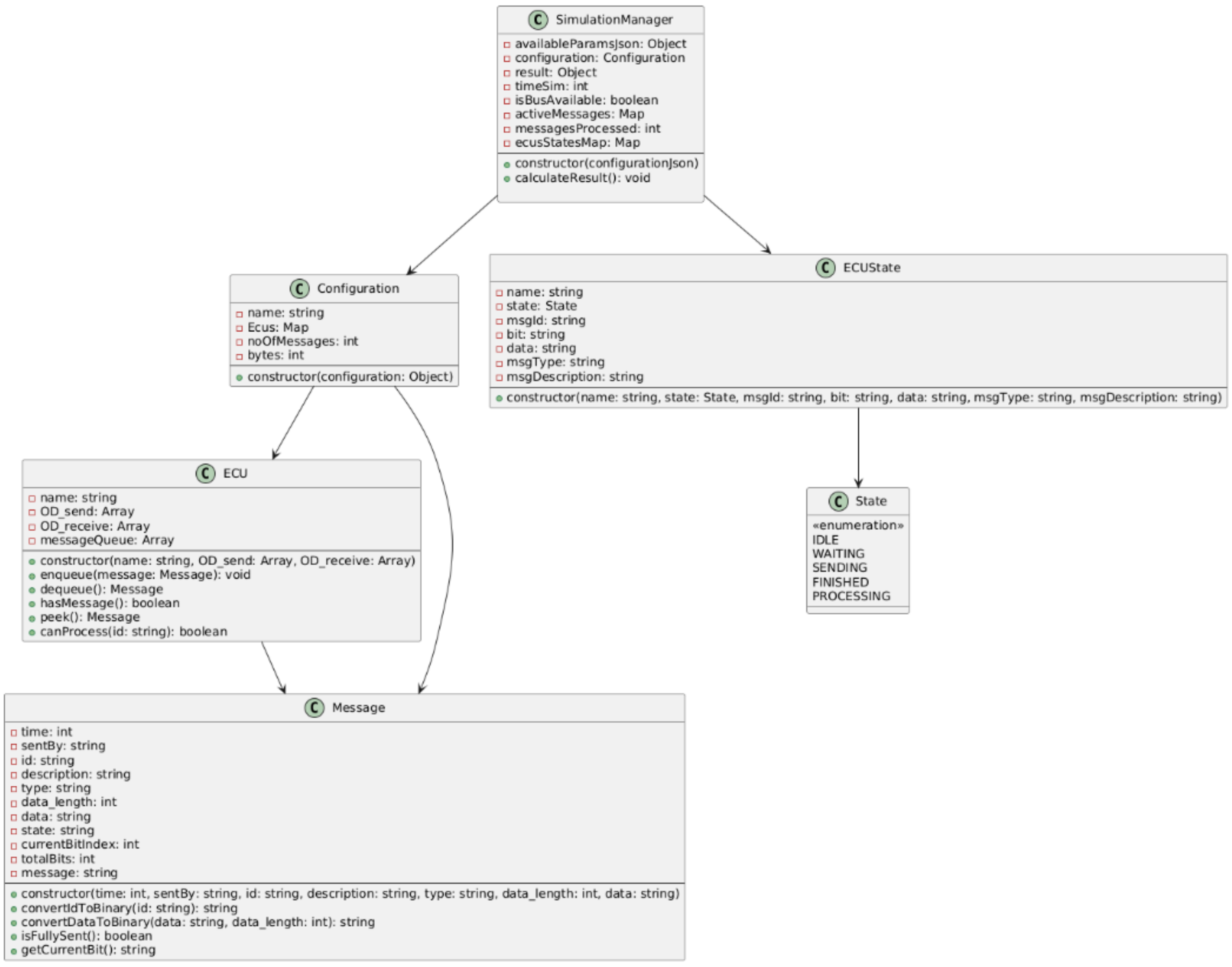
- **GET /api/configurations -** Returns an array of JSON objects representing available configurations. Each configuration contains details like the number of ECUs, timing actions, and other simulation parameters. It allows the client to display a list of configurations for the user to select.
- **POST /api/configurations -** Adds a new configuration for the simulation. It consists of a JSON object with configuration details (e.g., ECU count, actions at each time step). The response confirms creation with the ID of the new configuration. It enables the client to save new simulation setups, allowing for reusable configurations.
- **PUT /api/configurations/:id -** Updates an existing configuration by its unique ID. Accepts a JSON object with updated configuration details. Returns the modified configuration after confirmation. Allows the client to modify configurations by adding or removing ECUs and adjusting timing settings.
- **DELETE /api/configurations/:id -** Deletes a specific configuration by ID. Returns a confirmation message. Enables the client to remove unused or obsolete configurations.
- **POST /api/simulation/start -** Initiates a new simulation based on the selected configuration. Accepts a JSON object containing the configuration ID to be used. Returns a session ID for tracking the specific simulation instance. Starts the simulation on the server with the specified configuration.
- **GET /api/simulation/step/ -** Retrieves simulation results for a specific discrete time step in an ongoing or completed simulation. Returns a JSON object with computed values for the specified time step, including ECU states, bus voltage levels, and arbitration details. Allows the client to display data for a particular time step.
- **GET /api/simulation/log -** Retrieves the full log of the simulation for a particular session.

- **GET /api/simulation/results -** Retrieves the final results of a completed simulation. Returns a JSON object containing summarized data such as total messages sent and other key metrics. Allows the client to display an overview of the simulation after completion

4.3. Configuration JSON



```
const configurations = [
    {
        id: 1,
        name: "Configuration 1",
        numberOfEcus: 4,
        messagesSent: 10,
        ecus: [
            {
                ecu_id: 1,
                type: "sensor",
                role: "Parking Sensor",
                messages: [
                    {
                        time: 1,
                        type: "PDO",
                        id: "00011001000",
                        data_length: 4,
                        data: "0x8F91"
                    },
                    {
                        time: 3,
                        type: "SDO",
                        id: "00011001001",
                        data_length: 4,
                        data: "0x1A2B"
                    }
                ]
            },
            {
                ecu_id: 2,
                type: "actuator",
                role: "Brake Controller",
```



SimulationManager.mjs:

- Manages the overall simulation, including configurations, ECU states, and active messages.
- Responsible for controlling the bus, handling arbitration, and generating results.

Configuration.mjs:

- Represents the simulation configuration, including ECUs and messages.
- Initializes ECUs and assigns messages to their respective queues based on the provided configuration.

ECU:

- Models an Electronic Control Unit, including its message queues and operational data.
- Manages message sending and receiving capabilities.

ECUState:

- Tracks the current state of an ECU, including the message being processed.
- Used to log and monitor the ECU's state during the simulation.

Message:

- Represents a message in the CAN network, including its ID, data, and type.
- Converts data to binary and tracks the progress of transmission.

State:

- An enumeration representing the different states an ECU or message can have (e.g., IDLE, SENDING).
- Used throughout the simulation to manage ECU and message behavior.

Implementation

5.1 Class description

SIMULTAIONMANAGER.mjs

Description:

The SimulationManager class is the core of the simulation, orchestrating all activities in the CAN network. It manages the configuration, tracks the state of the ECUs and messages, and produces the final simulation results. The class is initialized with a configuration file that defines the ECUs and messages involved in the simulation. This configuration is validated and processed to set up the simulation. The SimulationManager ensures that the simulation adheres to the CAN protocol, handling arbitration, message transmission, and conflicts. It also keeps track of simulation metrics such as bus states, arbitration resolutions, and message processing status. This class is responsible for logging each step of the simulation and packaging the final results for analysis.

Responsibilities:

- **Load and Validate Configuration**: Loads the configuration file, validates it, and initializes the simulation setup, including ECUs and messages.
- **Orchestrate the Simulation**: Manages the entire simulation lifecycle, from start to finish, handling each time step.

Key Properties:

- **availableParamsJson**: A static property that loads and stores the available ECUs and messages metadata for validation and configuration.
- **configuration**: An instance of the Configuration class representing the current simulation setup, including ECUs and messages.
- **result**: A structured object containing the simulation's final results, including logs, ECU states, and performance metrics.messageRate: The number of messages sent per second by the ECU.
- **timeSim**: Tracks the simulation's elapsed time in discrete steps.
- **isBusAvailable**: A boolean flag indicating whether the CAN bus is free for message transmission.
- priority: The priority of messages generated by the ECU.
- state: The current state of the ECU (e.g., "Idle", "Sending", "Receiving").

Key Methods:

- **calculateResult()**: Drives the simulation by iteratively processing messages, handling arbitration, managing ECU states, and logging results until all messages are sent.updateState(state): Updates the ECU's current state.
- **handleBusIdle()**: Manages the state of the CAN bus when it is idle, initiating message transmissions and transitioning the bus state to "dominant."
- **handleBusOccupied()**: Handles the CAN bus when it is occupied, resolving arbitration conflicts and progressing message transmission for ECUs.
- **refreshActiveMessages()**: Updates the activeMessages map, checking each ECU for messages that are ready to be sent at the current simulation time.
- **getNoOfActiveMessages()**: Returns the number of ECUs with active messages ready for transmission.
- **addTimeInfo(busState, logMessages, arbitrationMessage)**: Records the state of the simulation at the current time step, including ECU states, bus state, and any arbitration messages.

ECU.mjs

Description:

The ECU class models an Electronic Control Unit in a CAN network. Each ECU is a standalone unit that interacts with other ECUs by sending and receiving messages. It maintains a queue of messages it needs to send and has metadata defining which messages it can send or process.

Responsibilities:

- Represent an individual ECU in the network with its unique sending and receiving capabilities.
- Manage a queue of messages that are scheduled to be sent.
- Determine whether a particular message can be processed based on its ID.
- Provide utility methods to manage and interact with the message queue.

Key Properties:

- **name**: The name of the ECU (e.g., ECM, ABS).
- **OD_send**: A list of message IDs the ECU can send.
- **OD_receive**: A list of message IDs the ECU can process.
- **messageQueue**: A queue of Message objects that the ECU is scheduled to send

Key Methods:

- **enqueue(message)**: Adds a Message instance to the ECU's message queue.
- **dequeue():** Removes and returns the next Message from the queue for processing.
- **hasMessage():** Checks if there are any messages left in the queue.
- **canProcess(id)**: Determines whether the ECU can process a message with the given ID.

MESSAGE.js

Description:

The Message class represents a single message in the CAN network. It includes all the relevant metadata about the message, such as its ID, data, sender, and type. The class also handles binary conversion for IDs and data, ensuring compatibility with the CAN protocol.

Responsibilities:

- Model a single message, including its metadata and content.
- Convert IDs and data into binary formats required by the CAN protocol.
- Track the progress of the message as it is transmitted bit by bit.
- Determine when the message has been fully sent.

Key Properties:

- **time**: The timestamp when the message is ready to be sent.
- **sentBy**: The name of the ECU that sends the message.
- **id:** The unique ID of the message, converted to binary format.
- **description:** A textual description of the message.
- **type:** The CAN message type (e.g., TPDO, RPDO).
- **data_length:** The size of the message's data in bytes.
- **state:** The current state of the message (e.g., undispatched, sending, sent).

- **currentBitIndex:** Tracks the progress of transmission by indicating the current bit being sent.

Key Methods:

- **isFullySent()**: Checks if the message has been completely transmitted.
- **getCurrentBit()**: Returns the current bit being transmitted.

ECUState.mjs

Description:

The ECUState class tracks the real-time state of an ECU during the simulation. It provides detailed metadata about the ECU's activity, such as the message it is processing, its type, and its current state.

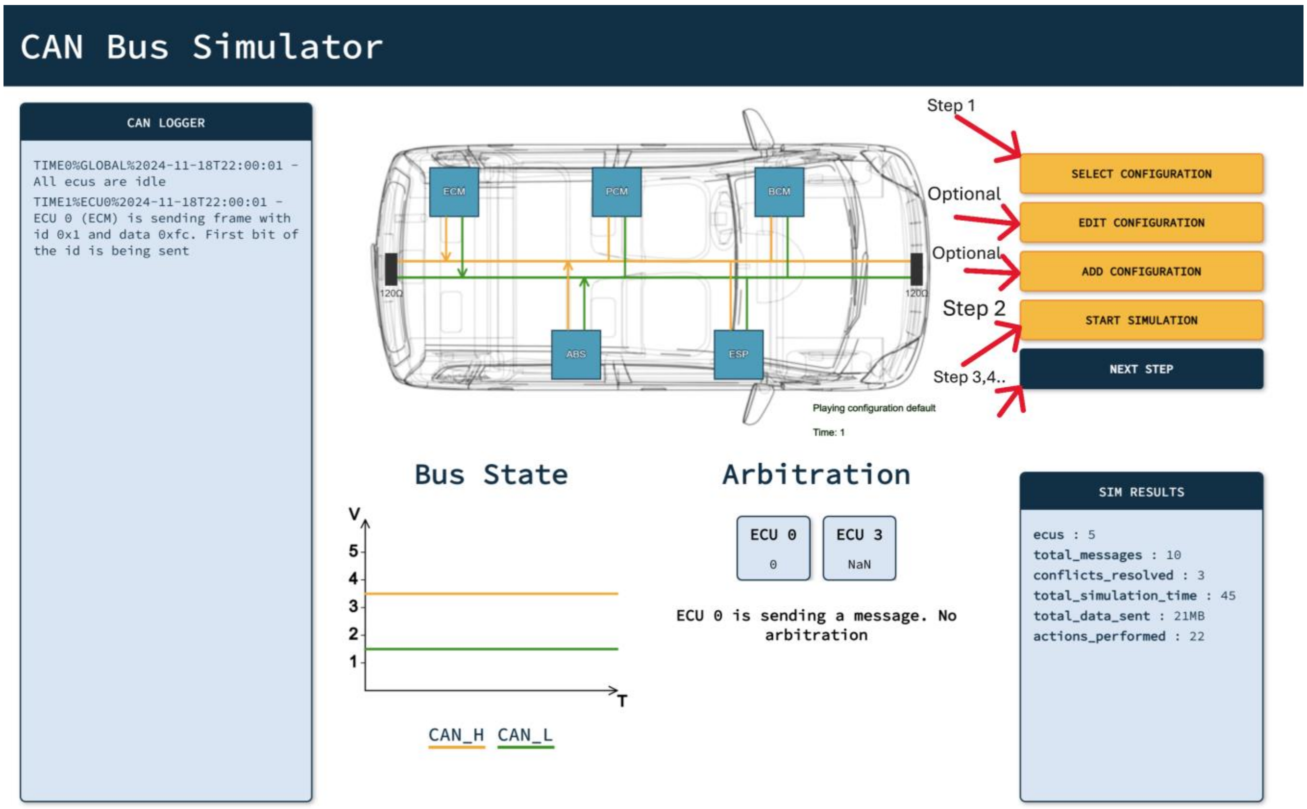
Responsibilities:

- Represent the current state of an ECU during the simulation.
- Store metadata about the message being processed by the ECU.
- Enable logging and debugging by providing detailed state information.

Key Properties:

- **name**: The name of the ECU whose state is being tracked.
- **state**: The current state of the ECU (e.g., IDLE, SENDING, PROCESSING).
- **msgId**: The ID of the message being processed.
- **bit**: The bit of the message currently being transmitted.
- **data**: The data content of the message being processed.
- **msgType**: The type of the message (e.g., TPDO, RPDO).
- **msgDescription**: A textual description of the message.

5.2 Navigating the UI



```
TIME0%GLOBAL%2024-11-18T22:00:01 -
All ecus are idle
TIME1%ECU0%2024-11-18T22:00:01 -
ECU 0 (ECM) is sending frame with
id 0x1 and data 0xfc. First bit of
the id is being sent
```

ECU 0 is sending a message. No arbitration

```
ecus : 5
total_messages : 10
conflicts_resolved : 3
total_simulation_time : 45
total_data_sent : 21MB
actions_performed : 22
```

Figure 6. UI Description

Step 1: Select Configuration

- Purpose: Choose a predefined simulation configuration.
- Description: This step allows the user to select a configuration file that defines the simulation's parameters, such as the number of ECUs, arbitration rules, and bus states. These configurations can be predefined or user-defined.
- Action: Click the "SELECT CONFIGURATION" button to load the desired setup.

Optional Steps:

Edit Configuration

- Purpose: Modify the selected configuration.
- Description: Users can edit parameters like ECU properties, message priorities, and simulation duration.
- Action: Click the "EDIT CONFIGURATION" button to open an editor.

Add Configuration

- Purpose: Create a new configuration from scratch.
- Description: Allows users to define new simulation parameters and save them for future use.
- Action: Click the "ADD CONFIGURATION" button to input new configuration details.

Step 2: Start Simulation

Purpose: Begin the simulation process based on the selected or edited configuration.

Description: This step initializes the simulation, loads the timeline, and prepares the visual representation of the CAN bus.

Action: Click the "START SIMULATION" button to begin.

Step 3: Next Step

- Purpose: Advance through the simulation timeline step by step.
- Description: This step updates the simulation visuals and logs in real-time. Users can see:
    - CAN Logger: Displays time-stamped log messages showing ECU activities.
    - Bus State: A graphical representation of the bus voltage states (CAN_H and CAN_L) over time.
    - Arbitration: Details about message arbitration, including which ECU's message is sent.
    - Sim Results: Summary statistics of the simulation, such as total messages sent and arbitration conflicts resolved.
- Action: Click the "NEXT STEP" button to advance to the next point in the timeline.

Visualization Details:

- CAN Logger: Real-time log of activities showing message transmissions and bus states.
- ECU Diagram: Visual placement of ECUs in the vehicle, with arrows showing message flows.
- Bus State Graph: Displays the voltage levels (CAN_H and CAN_L) on the bus over time.

- Arbitration Panel: Shows active ECUs and details of arbitration, if any.
- Sim Results Panel: Provides a summary of the simulation results, including:
    - Number of ECUs.
    - Total messages.
    - Resolved conflicts.
    - Total data sent.

Testing & Validation

6.1 Testing the arbitration

The arbitration mechanism in the CAN project was subjected to rigorous testing to ensure its correctness, reliability, and adherence to the CAN protocol standards. The testing process involved simulating various scenarios to validate the system's ability to handle bus access conflicts and resolve them effectively, ensuring the highest-priority message gains access to the bus.

Bibliography

[1] Texas Instruments ™, *"Introduction to the Controller Area Network"*, 2002 – revised 2016.

[2] Robert Bosch GmbH., *"CAN specification"*, 1991

[3] www.picotech.com, *https://www.picotech.com/library/knowledge-bases/oscilloscopes/can-bus-serial-protocol-decoding*

[4] CSSElectronics, *https://www.youtube.com/@CSSElectronics-CAN-Logger-X000*

[5] Quoc Viet Ha, *https://www.linkedin.com/pulse/rest-api-fundamental-concept-best-practice-edward-ha-yqsdc/*