Simulation of communication in industrial networks – CAN network

Student: Gozman-Pop Maria-Eliza

Structure of Computer Systems Project

Technical University of Cluj-Napoca

Contents

I. Introduction

1.1. Context

The aim of this project is to simulate how CAN based communication works in automotive and industrial systems for real-time data exchange between multiple devices (we often find them under the name ECUs - Electronic Control Units). To make this concept a bit easier, a good example would be the way the components of a car must communicate with each other to provide a good functionality: the engine ECU must be informed when the break is being used, while the AC unit would not be interested in this information. In order to make this more efficient and not have all components linked separately, a CAN bus is very useful, as it links all components, but only notifies the interested parts based on the encoding and identifier of the message.

In more technical terms, [1] CAN stands for *Controller Area Network* and it was developed by Robert Bosch in 1986 as a flexible, reliable, and robust solution for communication within the automotive vehicle. It is a serial, half-duplex, and asynchronous communication protocol and follows a decentralized communication infrastructure. The benefit of a decentralized protocol is that there is no central entity that can control the bus, meaning we can add or remove a node from the bus without disrupting the communication between other nodes. It features high data transmission speed, excellent error handling, automatic re-transmission of faulty messages, and high tolerance to electrical noise.

1.2. Objectives

The project aims to demonstrate how different devices (nodes) can communicate in real-time using a shared communication bus without conflicts. This will be done by simulating the transmission and reception of CAN messages between multiple ECUs, implemented in C++. What will be highlighted in the simulation: message broadcasting, message prioritization via identifiers, and how arbitration is handled using dominant and recessive signals, error preventing and checking mechanisms, and the user will be able to observe these in a real-time manner. The main focus when it comes to implementation is split into the following categories: message transmission, error detecting and error handling. To make the concept of CAN simpler to understand, the project will contain a user interface that will be easy to interact with and it will demonstrate various scenarios.

II. Bibliographic Research

2.1. What is CAN?

[1] The CAN bus was developed by BOSCH as a multi-master, message broadcast system that specifies a maximum signalling rate of 1 megabit per second. Unlike a traditional network such as USB or Ethernet, CAN does not send large blocks of data point-to-point from node A to node B under the supervision of a central bus master. In a CAN network, many short messages are broadcast to the entire network, which provides for data consistency in every node of the system.

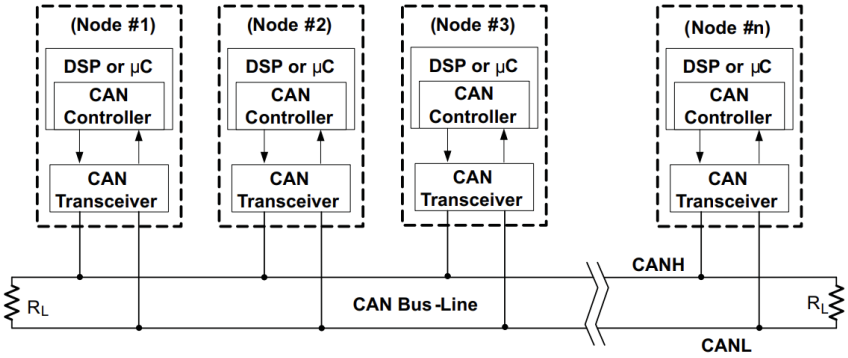

Figure 1. CAN Structure, *ref. [2]*

2.2. How does message prioritization work in CAN?

The original ISO standard laid out what is called Standard CAN. [2] Standard CAN uses an 11-bit identifier for different messages, which comes to a total 2048 different message IDs. CAN was later modified, and the identifier was expanded to 29 bits – this is called Extended CAN. CAN uses a multi-master bus, where all messages are broadcast on the entire network. The identifiers provide a message priority for arbitration.

CAN uses a differential signal with two logic states (called recessive and dominant), meaning it transmits signals over two wires (CANH and CANL). [3] Recessive indicates that the differential voltage between the two wires is less than a minimum threshold voltage, while dominant indicates that the differential voltage is greater than this threshold. Interestingly, the dominant state is achieved by driving a logic ‘0’ onto the bus, while the recessive state is achieved by a logic ‘1’, so a dominant state overrides a recessive during arbitration.

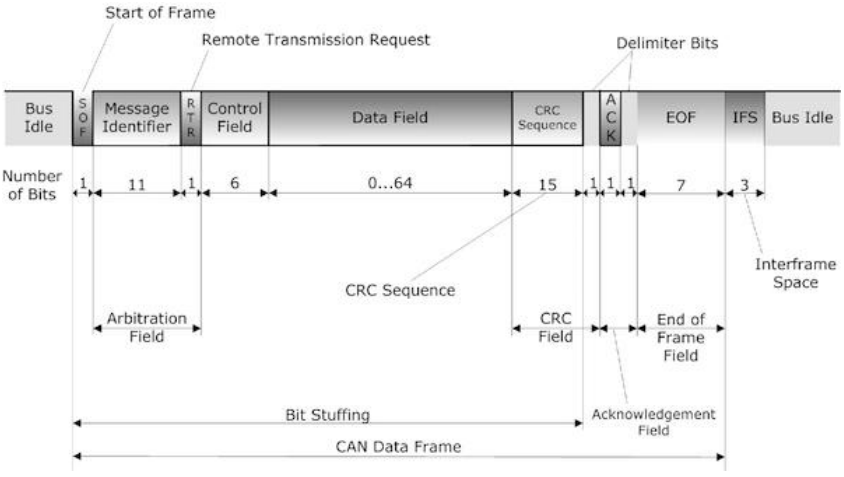

Figure 2. Structure of a CAN message, *ref. copperhilltech.com*

2.3. What are the key features of error detection and handling?

- Non-destructive Arbitration: CAN prioritizes messages based on their identifier. During arbitration, if a node detects a higher-priority message, it backs off, allowing the higher-priority message to continue uninterrupted.
- Bit Monitoring: Each node monitors the bus while transmitting. If a node sends a dominant bit (0) but detects a recessive bit (1), it identifies an error and stops.
- Bit Stuffing: After five consecutive identical bits, a node inserts a stuffed bit of the opposite value. If this rule is violated, a stuffing error is flagged.
- Cyclic Redundancy Check (CRC): CAN uses a CRC field to ensure data integrity. Any mismatch in the CRC value triggers a CRC error.
- Acknowledgment (ACK) Error: After receiving a message, a node sends an ACK bit. If the transmitter doesn't detect the ACK, an ACK error occurs.
- Error Frames and Fault Confinement: Nodes track errors using error counters. Frequent errors can push a node into an error passive or bus off state, limiting its participation to avoid disrupting the bus.
- Automatic Retransmission: When an error is detected, the faulty message is automatically retransmitted until it is successfully sent.

2.4. Algorithms Needed

- **Non-destructive Arbitration**

What is important to note is that a transmitting node constantly monitors each bit of its own transmission. The message with the lowest identifier (highest priority) wins the arbitration process, [3] each node transmits bits of its message identifier, starting from the most significant bit and if a node sends a recessive bit (1) but detects a dominant bit (0) on the bus, it backs off, allowing the node with the lower identifier to continue sending. This functionality is contained entirely within the CAN controller and is completely transparent to a CAN user. This is called non-destructive arbitration, because it uses bit monitoring.

Example:
Node A: 110101 (ID = 53 in decimal)
Node B: 101110 (ID = 46 in decimal)
Node C: 100011 (ID = 35 in decimal)

All three nodes start transmitting their IDs simultaneously, beginning with the MSB.

Node A: 1 NodeB: 1 NodeC: 1

All nodes agree (they all send 1), so arbitration continues.

Node A: 1 NodeB: 0 NodeC: 0

Node A sees a 0 on the bus (due to nodes B and C sending 0), while it sent a 1. Since 0 wins over 1, Node A loses arbitration and stops transmitting.

---------- NodeB: 1 NodeC: 0

Node B detects that Node C sent a dominant 0 while it sent a recessive 1. Since 0 wins, Node B loses arbitration and stops transmitting. Node C continues transmitting its ID since it has the

lowest ID (the highest priority) and no other node is trying to send anymore. At the end of arbitration, Node C wins because it has the lowest ID, meaning it has the highest priority on the bus. Node B and Node A will wait until the bus becomes available again to try transmitting their messages.

- **Cyclic-Redundancy Check (CRC)**

This algorithm is used to detect transmission errors by verifying the integrity of the transmitted message, helping identification of accidental changes to raw data during transmission by appending a special "CRC code" to the message, which the receiver can use to check the data. [2] The 16-bit CRC contains the checksum of the preceding application data for error detection with a 15-bit checksum and 1-bit delimiter. It is based on polynomial arithmetic, where both the message and the CRC are treated as binary polynomials. A fixed polynomial known as the generator polynomial is used to perform the division. The remainder of this division is the CRC code, which is then appended to the message. The receiver performs the same division using the same generator polynomial. If the remainder of the division (after subtracting the CRC) is zero, the message is considered error-free, otherwise, an error has occurred, and the receiver will either discard the message or request a retransmission. The generator polynomial used in CAN (x^15 + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1, 1100100110000001 in binary) is specifically designed for error detection and has excellent properties for detecting common types of errors such as: single-bit errors, burst errors (errors that affect a sequence of bits), bit inversion errors.

Example:

Message: 110101**000** (the last 3 bits were added for the purpose of the operation).

Polynomial: 1011

Start with the first 4 bits of the message (since the polynomial is 4 bits long). Align the polynomial under the first 16 bits of the message and perform XOR operation:

```
        1101
XOR     1011
----------------
        0110
```

Now bring down the next bit (the 5th bit from the original message), shift left and continue the XOR division with the next non-zero part:

```
        1100
XOR     1011
-----------------------------------
        0111
```

By repeating the process, we will get 1111 XOR 1011 = 0100

1000 XOR 1011 = 0011

0110 XOR 1011 = 1101

1010 XOR 1011 = 0001

The CRC bits consist of 0001.

We will add this sequence to the end of the initial message: 110101 + 0001 ⇔ 1101010001.

- **Bit Stuffing Algorithm**

Bit stuffing is a mechanism used to ensure that the transmitted signal stays synchronized between the sender and receiver, by preventing long sequences of identical bits (which could cause desynchronization). Specifically, bit stuffing ensures that the signal has enough transitions from 0 to 1 or 1 to 0, to maintain clock synchronization between the nodes. If a sequence of 5 consecutive identical bits is detected during transmission, the sender automatically inserts (stuffs) an extra complementary bit (the opposite bit) into the bit stream: if there are 5 consecutive 0s, a 1 is inserted; if there are 5 consecutive 1s, a 0 is inserted. This stuffed bit is then removed by the receiver upon detection, so it does not interfere with the actual data being transmitted.

Example:

Message: 0111111010

We notice a sequence of 6 bits of '1', meaning after the first five, a stuff bit with the value 0 will be introduced. The new message will be: 01111**1**01010.

- **Acknowledgment (ACK) Error Handling Algorithm**

After the data and CRC are transmitted, there is an ACK slot (a 1-bit field) in the CAN frame where the receiving nodes must place an acknowledgment bit. If at least one node successfully receives the message (and it passes the CRC check), it will overwrite the ACK slot with a dominant 0 bit (acknowledgment). The sending node expects this dominant 0 during the ACK slot as confirmation that the message was correctly received. In the contrary case, no receivers got the message, or the message was invalid, which will trigger automatic retransmission.

Each time an ACK error happens, the transmitter's error counter is incremented. If the error count exceeds certain thresholds, the node might enter the error-passive state or even go to the bus-off state, temporarily halting its ability to transmit messages.

2.5. Helpful Libraries

- Qt: GUI development and signal-slot
- C++ Standard Libraries:
  - **<sstream>:** stream-based string processing, such as formatting strings.
  - **<iomanip>:** formatting output, e.g., setting precision or width in numeric outputs.
  - **<vector>:** dynamic arrays to store collections of data.
  - **<queue>:** implements FIFO queues, for managing messages.
  - **<map>:** key-value pair storage, enabling fast lookup operations.
  - **<bitset>:** handling fixed-size sequences of bits, used for binary data processing.
  - **<stdexcept>:** handling exceptions, such as invalid arguments or runtime errors.
  - **<cstdint>:** provides fixed-width integer types like uint8_t and uint16_t, useful in networking or binary data handling.
  - **<algorithm>:** provides algorithms like sorting, searching, and manipulation of data.

2.6. Virtual Network Simulator Models

CORE

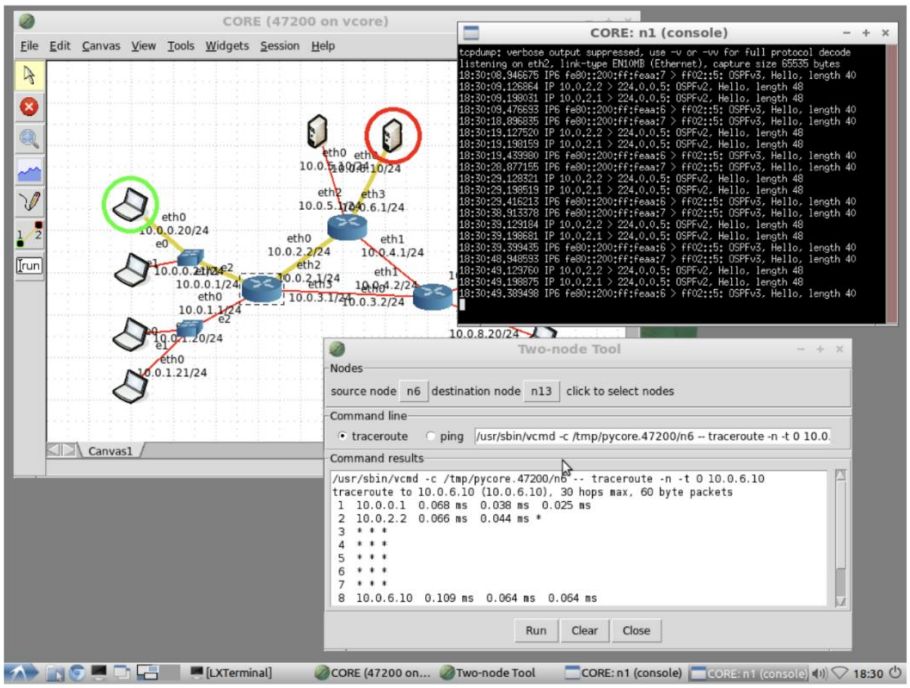

Figure 3. CORE simulator, *ref. networkstraining.com*

Containerlab

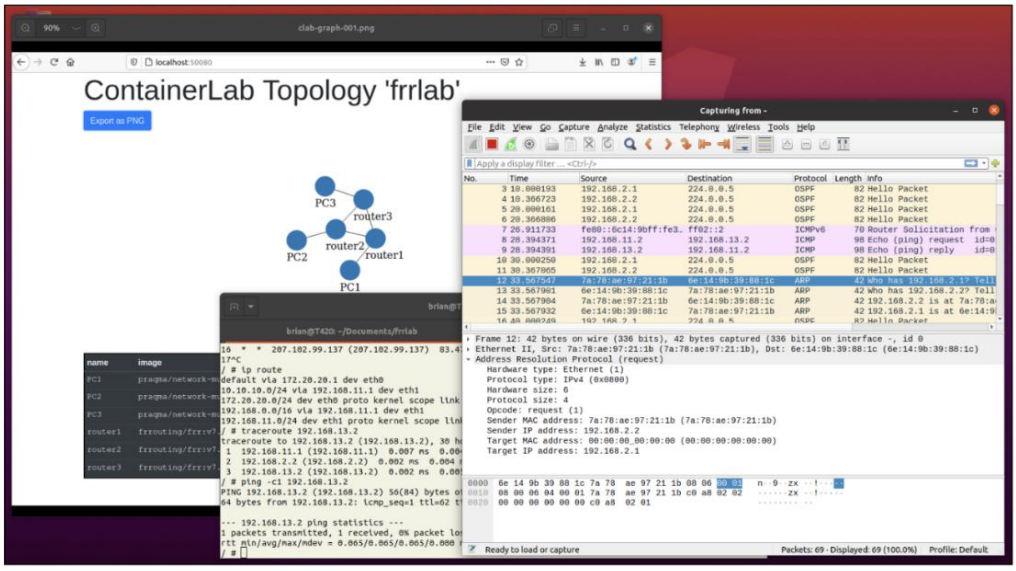

Figure 4. Containerlab simulator, *ref. networkstraining.com*

Cloonix

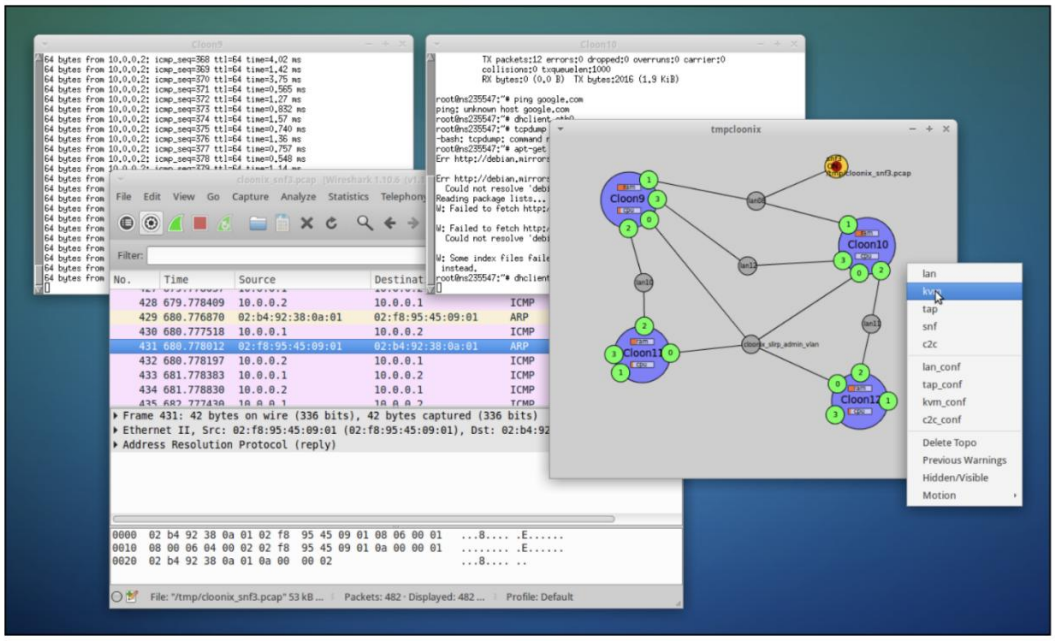

Figure 5. Cloonix simulator, *ref. networkstraining.com*

III. Analysis

3.1. Project Proposal

This project will simulate CAN bus communication using C++, with a focus on implementing the key features of the CAN protocol:

- **message transmission**, implemented by defining message objects that contain fields such as an 11-bit identifier, data fields, control bits, and error-checking bits; the transmission will simulate broadcasting these messages on the shared CAN bus to multiple ECUs, ensuring each message is received by all nodes on the network.
- **message prioritization** will be implemented by using the identifier field, and the non-destructive arbitration mechanism will simulate the competition between messages for control of the bus: when a node transmitting a recessive bit detects a dominant bit, it will stop its transmission, letting the higher-priority message continue.
- **error detection and handling** will be a key part of the project, and features like bit monitoring (checking that the bus state reflects the transmitted signal), cyclic redundancy check verification for detecting data corruption, and acknowledgment errors will be implemented.
- the **user interface** will display ongoing transmissions, the status of the bus, and message collisions, while also simulating various scenarios, such as sending multiple messages simultaneously to observe arbitration in action, and simulating errors to showcase how the CAN protocol handles them.

3.2. Project Analysis

3.2.1. How the project will work

The user should be able to input the number of nodes (and have their priorities be based on the generated id) and select the frequency of messages to be sent to a selected certain node during the simulation. Then, the user can watch how the program processes this data and transforms it into a suggestive visual representation. Later, he can save the information for future use and revision into a log file, which contains more useful and detailed information and error checking, that are not visible in the simulation run time.

Also, the user will have the ability to observe how errors occur in randomly selected nodes, as it would normally happen in a CAN bus, and to see how they are being handled: error counter gets incremented, and faulty nodes might even get removed from the bus, in case of passing a certain error threshold.

The simulation will also display information about each part of the process, and this will make it much more intuitive. The main idea is for this project to be designed in such a way that it is easily understandable even for people that are less experienced in this domain, or less familiar with this concept.

3.2.2. Use Cases

- **Set-Up Simulation**

  - **Number and priority of nodes**
    The user specifies the number of nodes, and their unique identifiers will be the ones to guide the priority (smallest identifier has the largest priority).
    The System initializes the specified nodes and displays confirmation.

  - **Messages Sent**
    The user selects for each node the messages that will be sent (the message IDs are out of the user's concern, as they are generated by the simulation based on the priority of the nodes – both the sender and the receivers), by choosing what nodes will receive that certain message.

  - **Errors**
    The user has the ability to choose if he wants to see the way the bit stuffing error preventing mechanism works in a CAN network. This will be visualized in the message sending process. Also, in the log file, a more detailed explanation of what error mechanisms are implemented, will be available to the user: such as CRC checking and acknowledgement bit consideration.

- **Start Simulation**
  The user initiates the simulation after configuring nodes and messages.
  The simulation begins, and the system starts displaying message transmissions and receptions in real-time.

- **Monitor the Network**
  The User observes the ongoing operation of the CAN network during the simulation. The System displays the messages that are pending/sent in two tables, the arbitration process bit by bit, cases of collision and bit stuffing (at the user's choice). The user can also observe how the error counters (TEC and REC) increment for the faulty nodes, and they will also get eliminated from the bus in a visually suggestive way.

- **Save Current Information**
  The user can save and access the results of the simulation, that are presented in greater detail in the log file, to be analysed in the future.

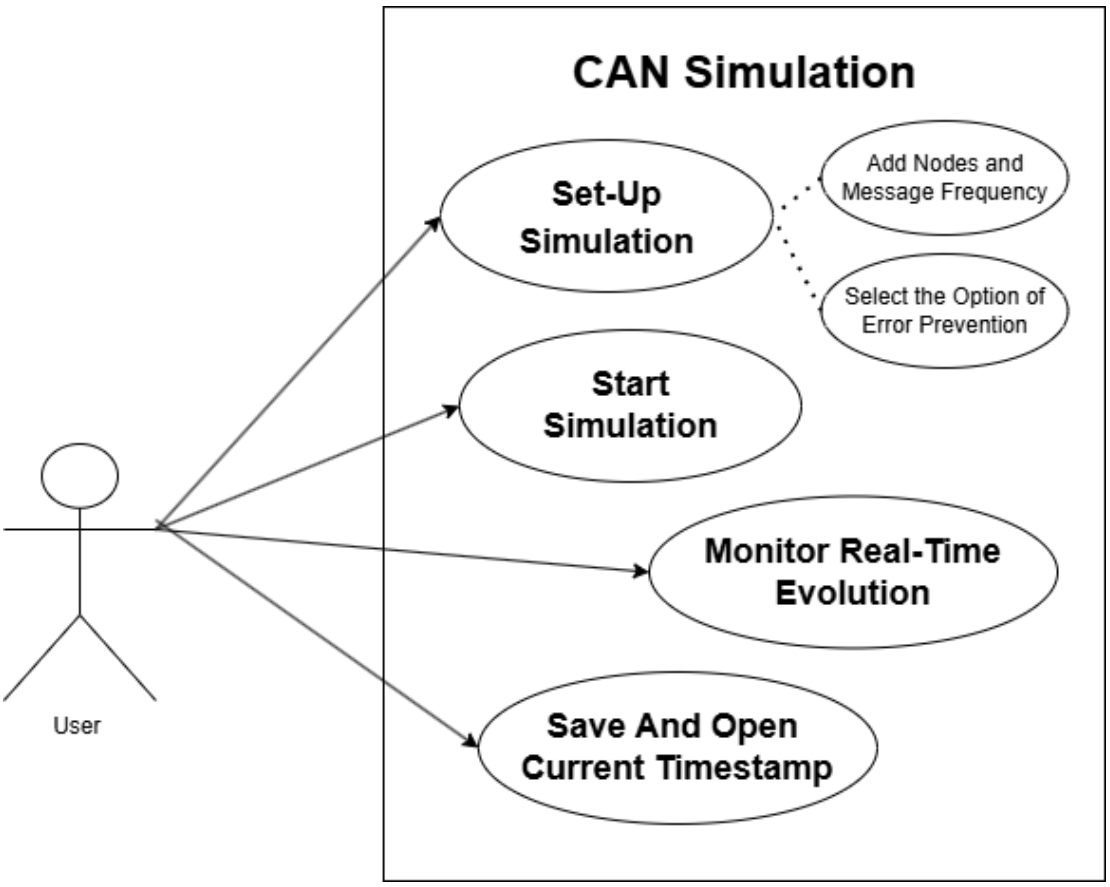

Figure 6. Use Case Diagram

User Guide

In order to make the simulation as intuitive as possible, there are a lot of visuals added to the functionality, so that it's easy to monitor. The user can have a highly educational experience by configuring their own custom simulation like this:

- **Adding Nodes to the Simulation:**

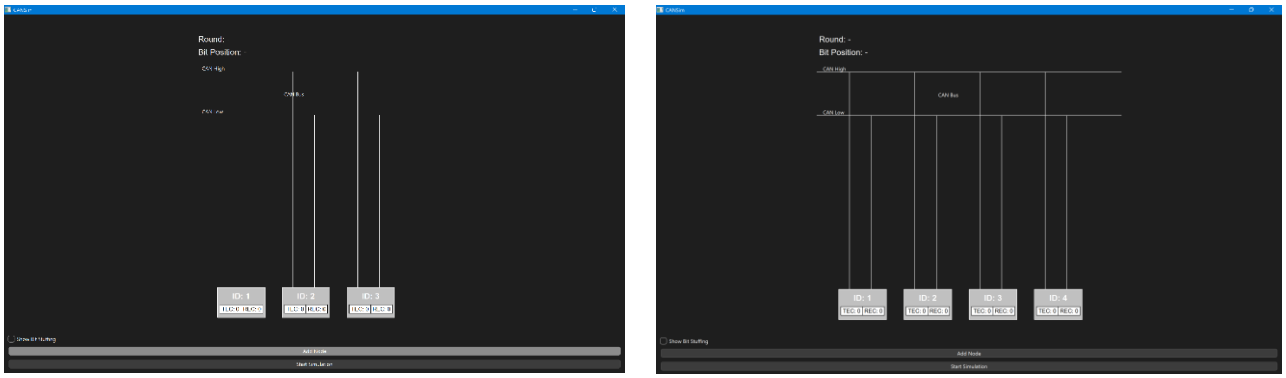

Figure 7: (GUI) Adding Nodes

As highlighted in the pictures, the user can add as many nodes, and the CANBus will get larger and larger; in order to do this, the user only has to press the AddNode button, and the simulation will generate the ids in the order they were added in (what is important to mention is that the priority of the nodes is based on their ides – smaller id means higher priority.

- **Selecting the Bit Stuffing Visibility Option**

As was mentioned, the error preventing mechanism – bit stuffing – used in Can, will be easily visualized on the bus after this configuration. All the user must do is select the option in the checkbox and look for the differences.

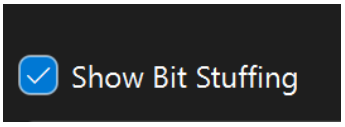

Figure 8: (GUI) Bit Stuffing Checkbox

What the user will see is the presence of the opposite bit, after five consecutive bits of the same type. This will not interfere with the arbitration, and will simulate the hardware specific event very well, in the message identifier transmission.

- **Configuring the Messages:**

The user is being given the opportunity to configure the way the messages that will be sent, by selecting the options that mostly concerns them (data will be generated randomly as this is not that important). What is important to mention is that the nodes are clickable items and the result of this is that the user can choose to which nodes the corresponding node will send a message to (the frequency – message/min – must be selected, which introduces the concept of timed messages). This is also very simple to use, the receiving nodes must be selected from the window.

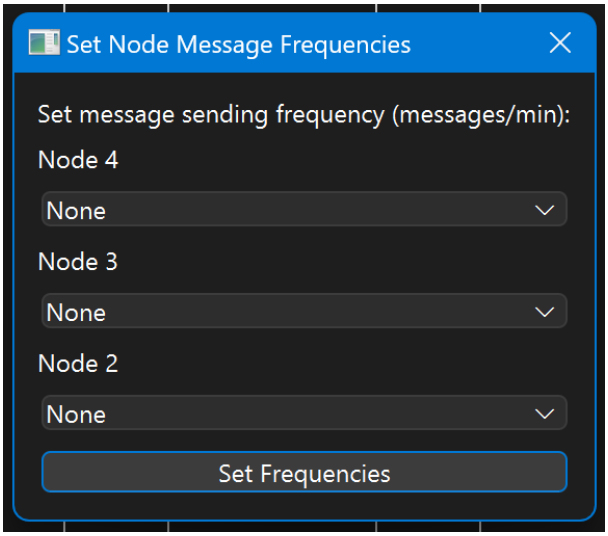

Figure 9: (GUI) Configure Messages

What is important to mention is that the nodes are clickable items and the result of this is that the user can choose to which nodes the corresponding node will send a message to (the frequency – message/min – must be selected, which introduces the concept of timed messages). This is also very simple to use, the receiving nodes must be selected from the window.

- **Starting the Simulation:**

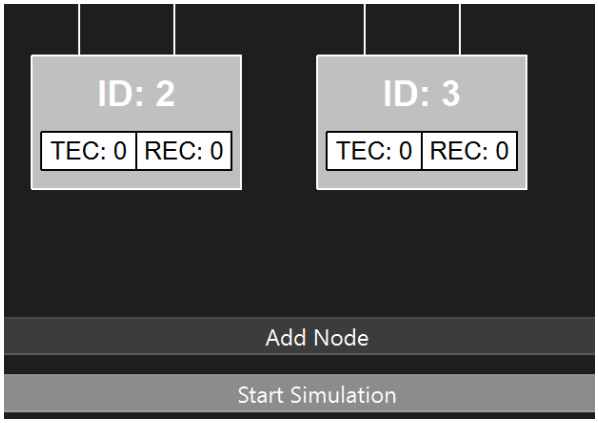

Figure 10: (GUI) Start Simulation Button

Once all the messages have been configured and the user is happy with their selection, it can start observing the CAN Bus working process by pressing the start simulation button.

- **Message Panels:**

There are two maeesage panels: one for the messages that are to be sent (in pending state) and one for the messages that have been sent. The simulation offers real time visualization of the progress: each message is sent at a time (11 clock cycles on the bus for 11 bit identifiers, or 12 in certain cases of bit stuffing), based on the priority of its identifier, and it is being sent to the second table. In the figure, the first picture shows a section from the beginging of the simulation, and the second picture shows the simlation into a state more towards the end.

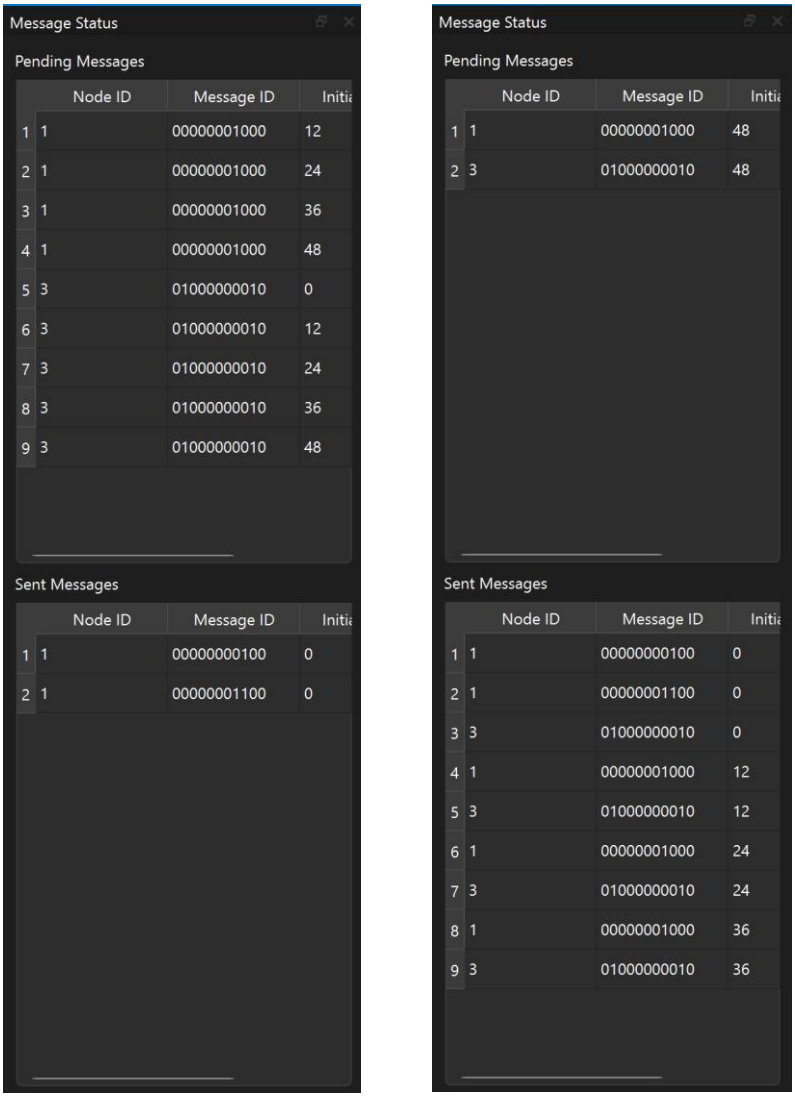

Figure 11: (GUI) Message Panels

- **Wire Colouring (Lighting Up)**

In order to have a visually suggestive simulation, the colour of the nodes has been set up in a particular way: it is known that each node can either send 0, 1 or not send anything. To send 0 – dominant state – the difference between the two voltages applied on the wires must be almost 0, meaning they have identical voltages. This is represented by the colour yellow on both wires, to suggest a "medium" voltage on them ~ 2.5V. The option to send 1 is represented by the corresponding CANLow wire set as red, and the corresponding CANHigh as green; here, to send 1, the difference between the voltages must be about 2V, this meaning that CANLow has applied a small voltage ~ 1.5V, and CANHigh a large voltage ~ 3.5V. The main CANLow and CANHigh, the ones of the bus, respond to these situations accordingly: if there is at least one node in dominant state, the bus will also be in dominant state, if there are only recessive states, the bus will also be in recessive state. If the bus is inactive (when all nodes are inactive, the colour will be white).

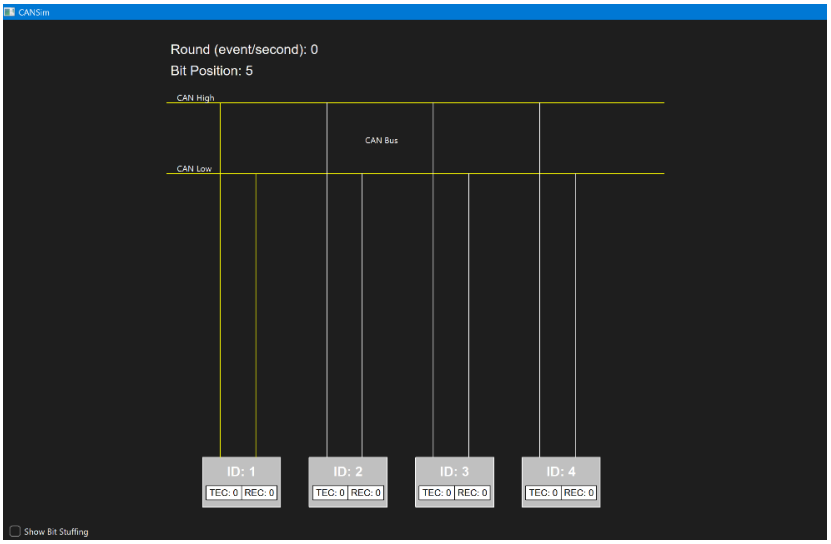

Figure 12: (GUI) Dominant Wire State with only Dominant State Sending

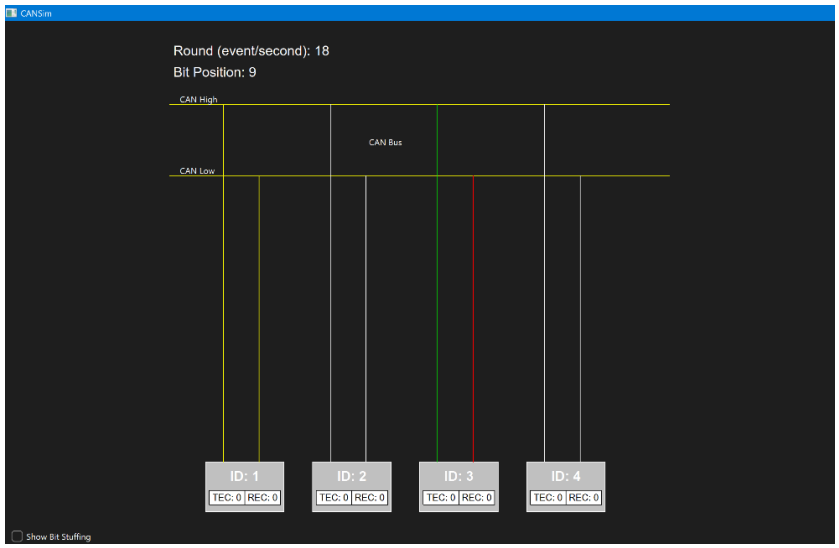

Figure 13: (GUI) Dominant Wire State with both Dominant and Recessive States Sending

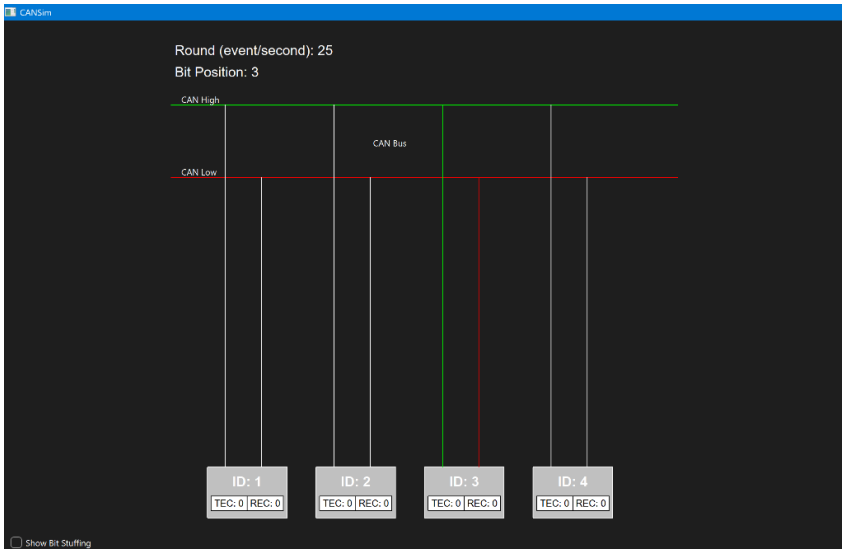

Figure 14: (GUI) Recessive Wire State

- **Node Information**

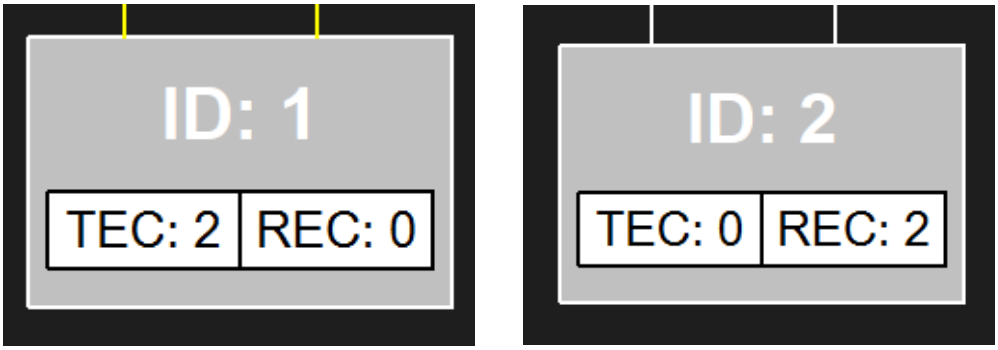

Figure 15: (GUI) TEC and REC counters

The node has the two error counter tags that are being updated in real time, and that visually suggest the problems that are happening during the simulation.

Figure 16: (GUI) Idle Node State

Once the established error counter threshold is passed, the node is sent into an idle state, and it cannot send or receive other messages. It is basically eliminated from the bus, and it can no longer participate in the arbitration, the messages that have been scheduled for it in the beginning will remain in the pending messages table.

Figure 17: (GUI) Received Message State

This state of the node marks the moment when the node received the message, it happens very fast, so I chose a colour that will attract the users attention towards it.

- **Round and Bit Position Labels**

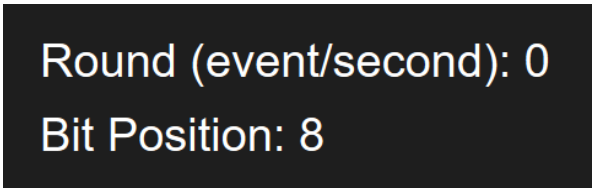

Figure 18: (GUI) Guiding Labels

In order to make the arbitration easier, there were two labels added that guide the user and help them understand a bit better what and more importantly “when” everything happens.

- **Log Of Events File**

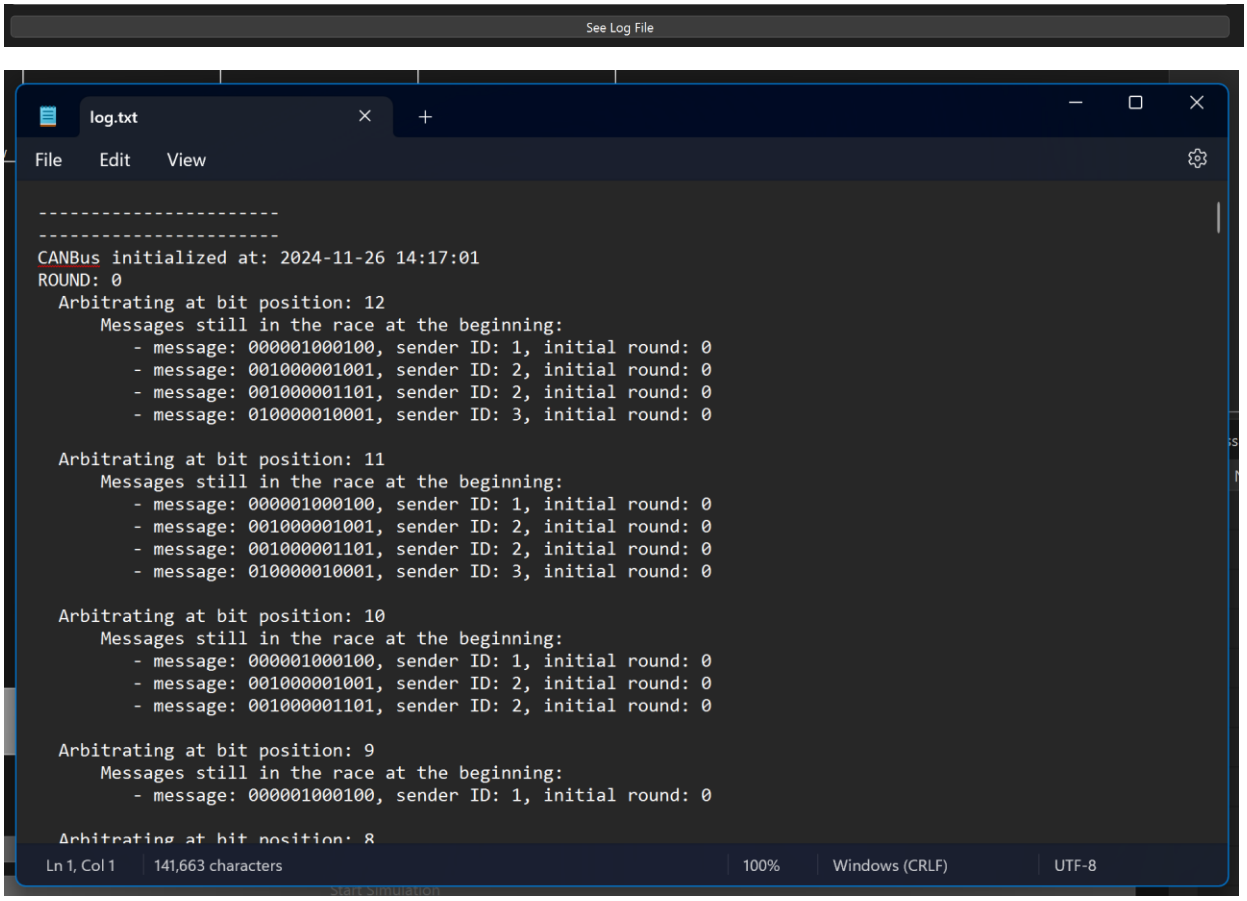

Figure 19: (GUI) Log Button and File Output

Log can be accessed during the simulation using the log file button, the output in this file is much more detailed and more helpful in understanding how CAN works, because it contains error checking as well.

This sums up the way to use and understand the CAN Simulation implemented. It’s a set up relatively simple to understand, being highly interactive and visual. The user does not require a lot of knowledge in this domain to understand the working principles of the CAN Bus, helped by the many different scenarios that can be created.

IV. Design

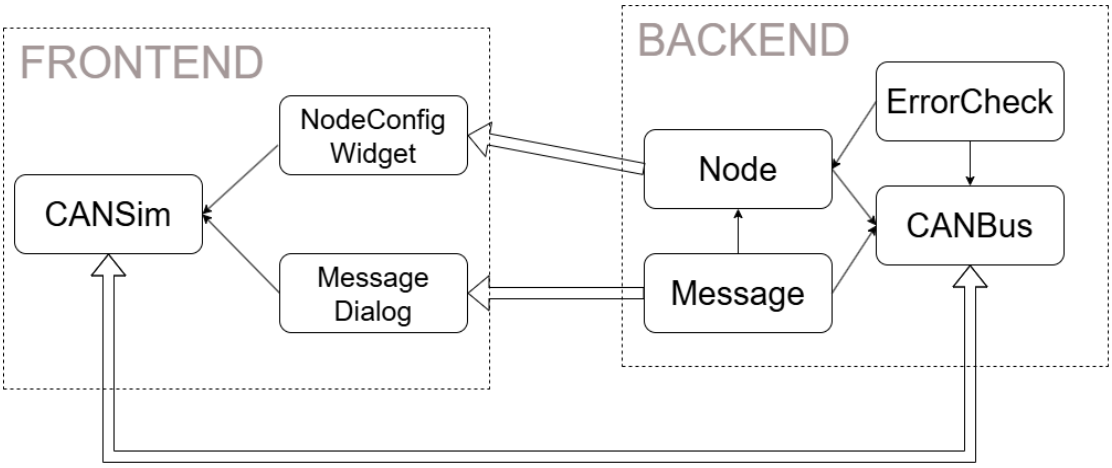

Figure 20. General Architecture Diagram

The initial design for the CAN simulation includes three core classes: Node, Message, and CANBus. These classes establish the fundamental components and operations for a basic CAN bus system, providing the following essential functionalities.

The classes have been implemented in C++, chosen for the core logic because of its high performance and low-level memory control, which are essential for simulating real-time systems like CAN. It also provides object-oriented programming, and the classes model real-life components, making the code much more modular. Data is handled by using message encapsulation, error checking, and node communication, so that the simulation can handle message transmission and error-checking operations with high efficiency.

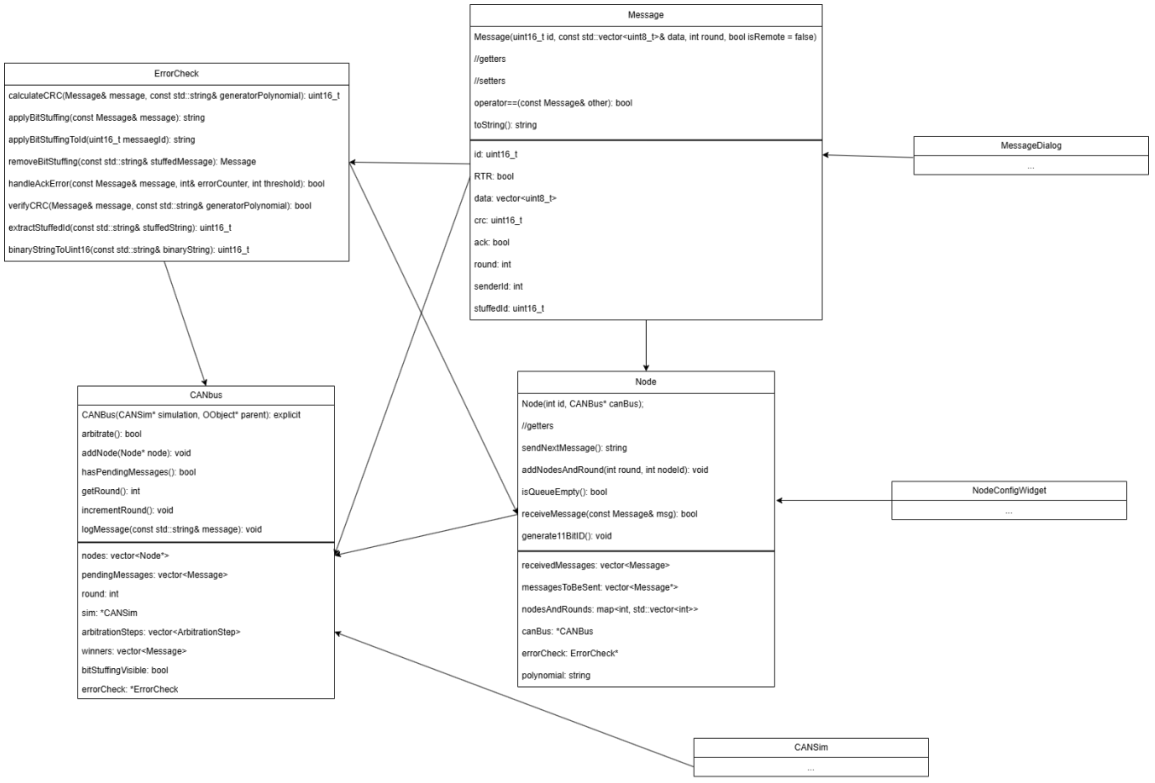

Figure 21. Class Diagram for Message and Arbitration

To this initial design, the user interface related classes have been added as it can be seen in the diagram bellow: NodeWidgetConfig, MessageDialog and CANSim. The classes work together to build a GUI that is easy to understand and use, offering the visual representation of the nodes, the CAN bus and the wires, and also message processing visuals.

The classes have been designed with the use of the Qt framework, which provides the tools for building a cross-platform, responsive, and interactive graphical user interface (GUI). This allows users to interact with the CAN bus simulation in real-time, configure nodes, initiate message transmission, and observe bus behaviour, making it highly educational. Qt has an event-driven architecture that aligns well with the CAN protocol simulation, where actions (such as sending messages or handling errors) occur in response to specific events.

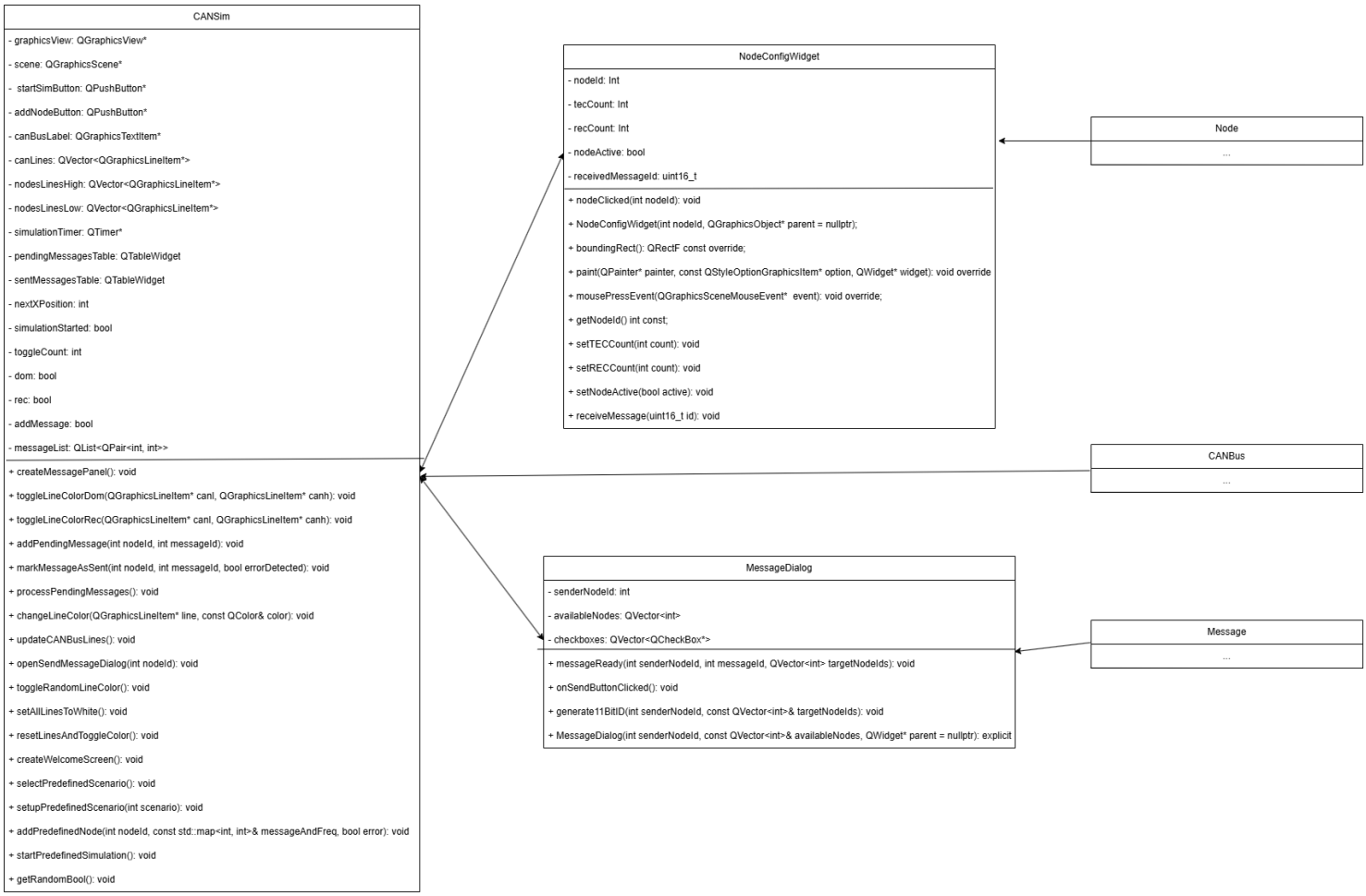

Figure 22. Class Diagram for Graphical User Interface

The simulation is structured around mimicking the key behaviours and communication dynamics of a Controller Area Network (CAN) system, which allows multiple electronic nodes to communicate efficiently in real-time within a shared bus architecture. The Central Controller (CANSim) serves as the orchestrator of the entire simulation environment. It sets up and coordinates all components, ensuring the nodes and the CAN bus operate in sync. Each Node in the simulation is equipped with a unique identifier and has capabilities to send and receive messages; they can initiate messages to be broadcast over the CAN bus or listen for messages relevant to them. The CANBus class

simulates the core behavior of a CAN bus as a shared communication medium. In an actual CAN network, only one message can occupy the bus at a time, and there are mechanisms to detect and handle message collisions. NodeConfigWidget allows users to interact with each node individually, such as configuring settings or initiating message transmission. MessageDialog lets users select target nodes and prepare messages, mimicking how a real-world system might let users or automated systems control message flow and set up communication tasks. The Message class represents individual data packets transmitted over the CAN bus; each message has metadata (like a unique identifier) and content, simulating how CAN messages encapsulate data and ensure its delivery to the correct nodes.

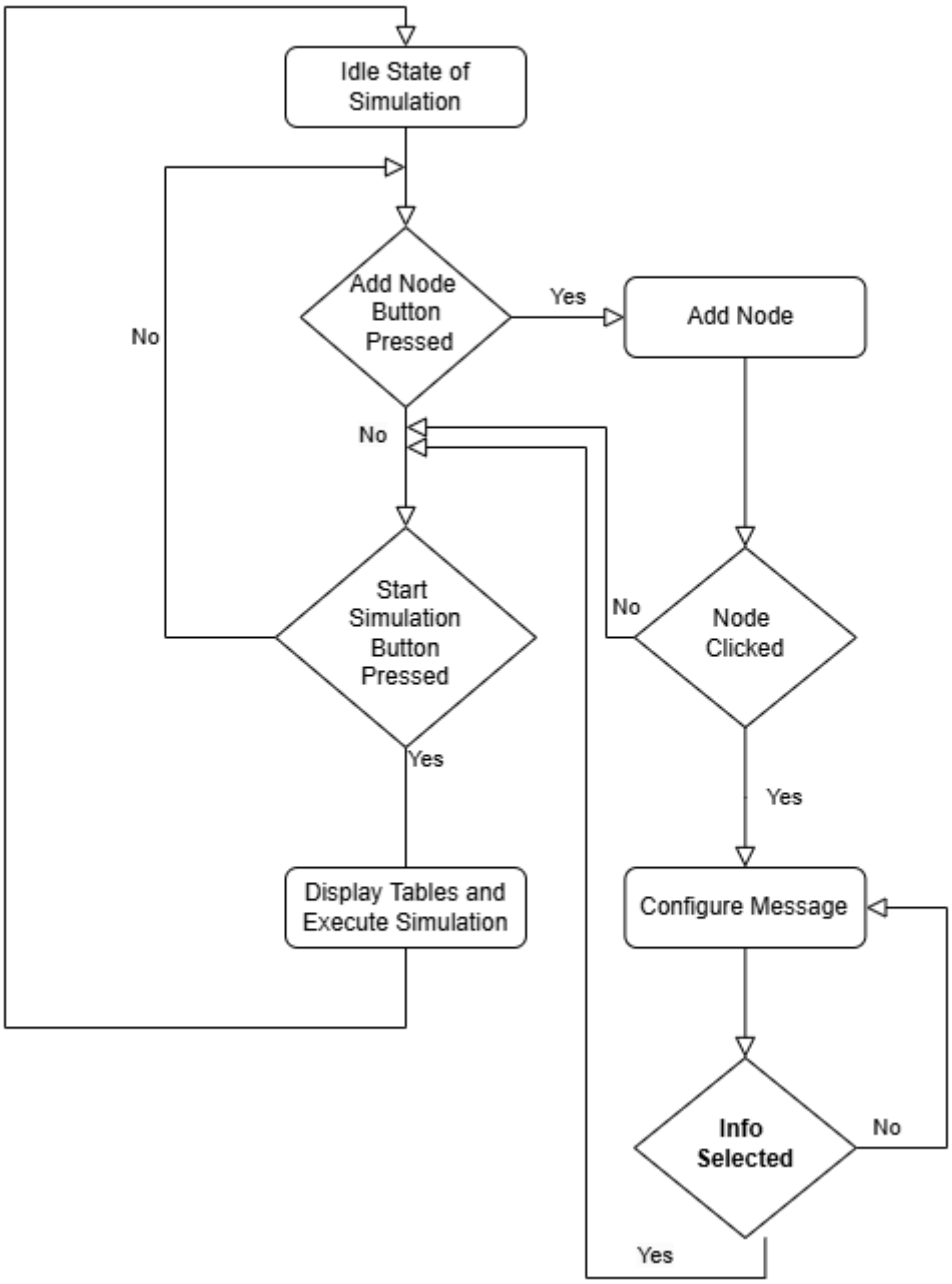

Figure 23. Flowchart Diagram

V. Implementation

5.1.

The Node class represents a device on a CANBus. It manages sending and receiving messages to and from the CANBus, as well as performing error checking and handling message queueing. Each Node has a unique ID and participates in message exchanges based on the CAN protocol.

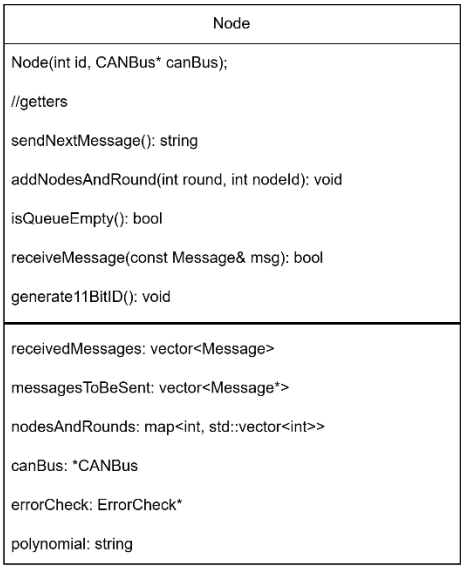

Figure 24. The Node Class

- **Attributes:**

- nodeId: a unique identifier for the node, used to identify the sender or receiver of a message.
- receivedMessages: a vector storing messages received by the node, acts as a log of messages this node processes from the CANBus.
- messagesToBeSent: a queue storing messages the node intends to transmit, these messages are queued and sent in sequence.
- canBus: a pointer to the CANBus object to which this node is connected, nables interaction with the CANBus, where messages are broadcasted.
- nodesAndRounds: A map associating communication rounds with node IDs, helps track which nodes participate in which round of message exchanges.
- errorCheck: a utility for performing error-checking operations like CRC calculation and bit-stuffing.
- polynomial: a polynomial used for CRC calculations.

- **Methods**:

- Node(int id, CANBus* bus): constructor for initializing a node with a unique ID and associating it with a CANBus.
- bool receiveMessage(const Message& msg): handles receiving messages broadcast on the CANBus, returns true if the CRC verification passes, and false otherwise.
- string sendNextMessage():transmits the next message in the messagesToBeSent queue, and return the bit-stuffed message string for transmission.

- void addNodesAndRound(int round, int nodeId): associates a node ID with a communication round, and maps a given round to a nodeId, storing this mapping in nodesAndRounds.
- vector<uint8_t> generateRandomData(size_t size): generates random data for CAN messages.
- void generate11BitID(): constructs 11-bit message identifiers based on the node's role and associated nodes in each round.
- int getNodeId() const: retrieves the unique ID of the node.
- const std::vector<Message>& getReceivedMessages() const: retrieves all messages received by the node.

5.2.

The Message class Represents data and ID for arbitration, encapsulating all information needed for a transmission, including priority (via id) and content (data), which nodes share and arbitrate over on the CANBus.

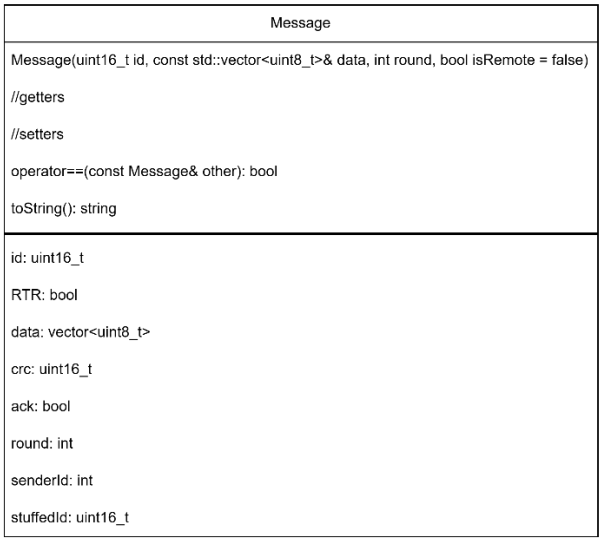

Figure 25. The Message Class

- **Attributes**:

- id: the message's unique identifier, represents the sender and receiver in an encoded format.
- data: a vector of bytes (up to 8) representing the message payload, contains the actual information being transmitted.
- round: an integer indicating the communication round, used to track which round the message is associated with.
- CRC: a 16-bit value representing the Cyclic Redundancy Check and a delimiter for error detection, which ensures the integrity of the message.
- ACK: a boolean flag for acknowledgment, that indicates whether the message has been acknowledged by the receiver(s).
- senderId: an integer identifying the node that created this message.
- stuffedId: a16-bit ID that includes bit-stuffing, used for the stuffed transmission over the CANBus.

- **Methods**:

- Message(uint16_t id, const std::vector<uint8_t>& data, int round, bool isRemote): constructor that initializes the message with its ID, data, round, and acknowledgement flag.
- getters (e.g., getId(), getData(), getRound())
- setters (e.g., setId(), setData(), setRound())
- toString(): converts message data into a string format

5.3.

The CANBus class Manages network-wide message arbitration, transmission, and reception for all connected nodes. It coordinates transmission rounds, ensuring each node sends messages synchronously and fairly across rounds.

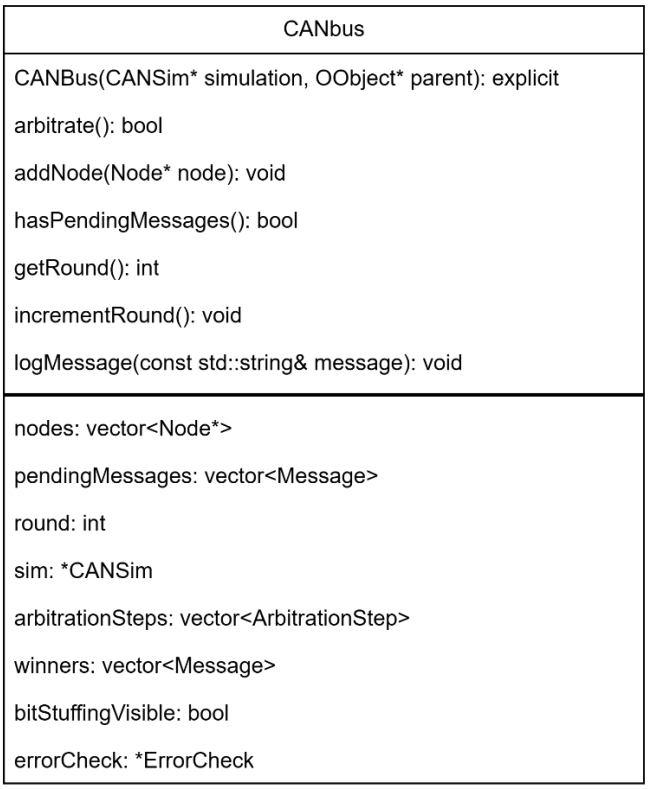

Figure 26. The CANBus Class

- **Attributes**:

- round: tracks the current communication round for arbitration.
- sim: a pointer to the CANSim object, which manages the overall simulation environment.
- nodes: a list of pointers to Node objects connected to the bus.
- pendingMessages: a vector holding messages waiting to be transmitted.
- arbitrationSteps: a collection of steps detailing each stage of arbitration, used for debugging or visualization.
- bitStuffingVisible: a flag indicating whether to consider bit-stuffing during arbitration.
- winners: a list of messages that successfully completed arbitration and were transmitted.

- **Methods**:

- CANBus(CANSim* simulation, QObject* parent): constructor that initializes the CANBus object with a reference to the simulation, opens a log file to track events and writes an initialization timestamp.
- void logMessage(const std::string& message): appends a message to the log.txt file for record-keeping.
- void incrementRound(): advances the round attribute by one, logs the current round after arbitration completes.
- void addNode(Node* node): adds a new node to the bus and updates the list of connected nodes.
- bool arbitrate(): performs the arbitration process to determine which message will be transmitted in the current round.

    - Step 1: Filter Messages by Round: Filters messages whose round is less than or equal to the current round and logs if there are no messages to transmit or none for the current round.
    - Step 2: Apply Bitwise Arbitration: compares message identifiers bit-by-bit to select the highest-priority message: if bitStuffingVisible is true, uses stuffed IDs for arbitration, tracks contenders (messages still in the race) and non-contenders at each bit position, the message with the lowest value at the most significant differing bit wins.
    - Step 3: Process the Winning Message: logs the winner's details, including sender ID, stuffed ID, and CRC and notifies nodes that match the winner's receiver bits. If the CRC is valid, the nodes acknowledge the message, and the message is removed from pendingMessages.
    - Step 4: Handle Non-Contenders: adds losing messages back to the pendingMessages if not already present.

5.4.

The ErrorCheck class is responsible for simulating various error-checking mechanisms commonly used in CAN communication. These include CRC calculation, bit stuffing, and bit un-stuffing, as well as error detection and correction mechanisms like acknowledgment and CRC validation.

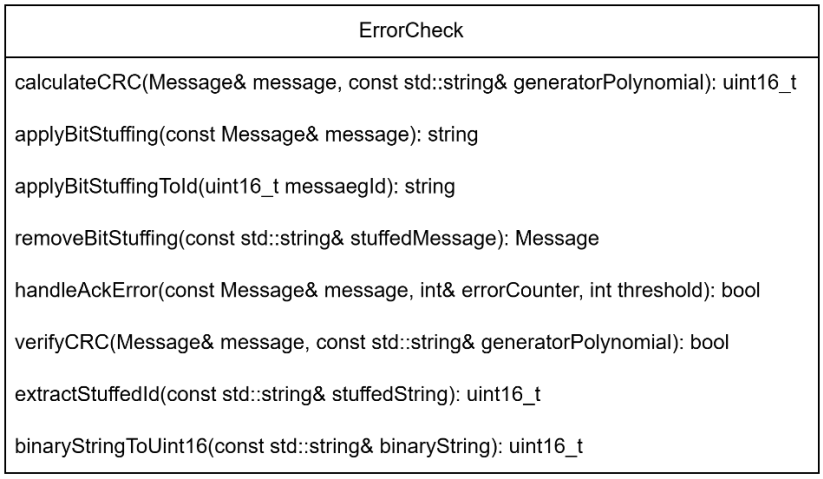

Figure 27. The ErrorCheck Class

- **Methods:**

- calculateCRC: calculates the CRC (Cyclic Redundancy Check) for a message using a given generator polynomial.
- applyBitStuffing: Implements bit stuffing by inserting the opposite bit after five consecutive bits of the same type.
- **removeBitStuffing**: removes stuffed bits from a binary string to reconstruct the original message.
- handleAckError: detects and handles acknowledgment (ACK) errors.
- verifyCRC: validates the CRC of a message.
- extractStuffedId: the stuffed ID from a binary message string.
- applyBitStuffingToId: specifically applies bit stuffing to a message's 11-bit identifier.
- binaryStringToUint16: converts a binary string into a 16-bit unsigned integer.

5.5.

The CANSim class represents a graphical simulation of a CAN (Controller Area Network) bus system, using Qt for rendering and interaction. This simulation includes visual representations of CAN High and CAN Low lines, multiple nodes connected to the bus, and controls for sending messages between nodes.

The CANSim class primarily manages the following components:

- Graphics View and Scene: a graphical window to display the CAN bus and nodes.
- CAN High and Low Lines: Represent the two communication lines in the CAN bus, with color-coded line segments to indicate data transmission.
- Nodes: Represent devices on the CAN bus capable of sending and receiving messages.
- Message Handling: Manages sending, pending, and completed messages, including error-checking and status tables.

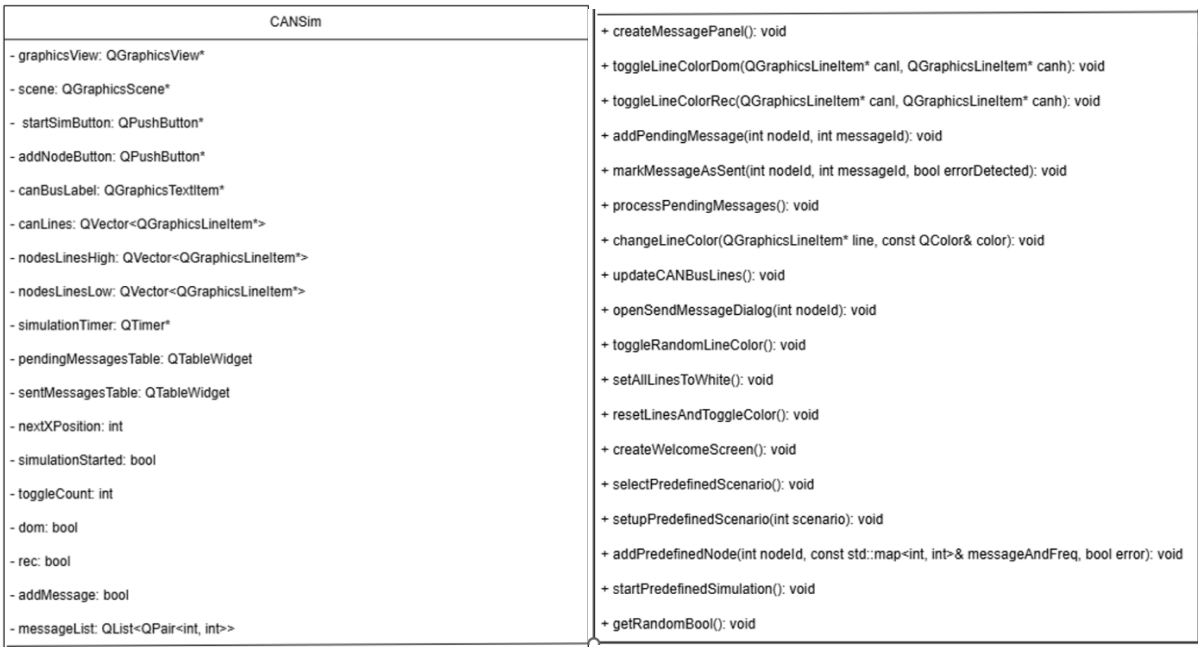

Figure 28. The CANSim Class

- **Attributes:**
  - Graphics Elements:
    - graphicsView: the widget displaying the simulation's graphical scene.
    - scene: the object that holds all graphical items, including lines, nodes, and labels.
    - canLines: a vector of objects representing CAN High and Low lines.
    - nodeLinesHigh and nodeLinesLow: Lists of connection lines between each node and CAN High or Low, allowing for individual color toggling.

  - Control Elements:
    - startSimButton: button to initiate the simulation.
    - addNodeButton: button to add new nodes to the CAN bus before starting.
    - bitStuffingCheckBox: checkbox that selects error prevention mechanism
  - Simulation State:
    - simulationStarted: Flag indicating if the simulation is active.
    - nextXPosition: Tracks the x-coordinate for placing new nodes.
    - dom and rec: Flags representing dominant (yellow) and recessive (red and green) states for data transmission indication.
  - Message Handling:
    - pendingMessagesTable and sentMessagesTable: Tables in a dock widget to track the status of messages—either pending or sent (with error status).
    - messageList: A list of message IDs and their source nodes, representing messages queued for transmission.
    - addMessage: A flag for handling messages during initialization.

- **Methods**:

- void toggleRandomLineColor(): toggles line colours on the CAN lines and node connection lines to simulate data transmission states.
- void setAllLinesToWhite(): resets all lines to white, simulating the idle state of the bus.
- void openSendMessageDialog(int nodeId): Opens a dialog for a specific node to send a message. It allows selecting target nodes and queues the message in messageList.
- void createMessagePanel(): Sets up a dock widget with tables to track pending and sent messages.
- void addPendingMessage(int nodeId, int messageId): adds a message to the pending table.
- void markMessageAsSent(int nodeId, int messageId, bool errorDetected): Marks a message as sent, transferring it from the pending table to the sent table with error status.
- void updateCANBusLines(): adjusts the CAN High and Low line lengths based on the position of the latest node added.
- void processSimulation(): the main method that implements the visualization of the CAN simulation and processes the data received by CANBus;
- void addNode(Node* node): adds the node to the CANSim nodes list
- Node* findNodeById(int id): finds a node in the list by the id
- vector<Message> collectAllMessages() const: collects the messages from all the nodes and puts them in a list

5.6.

The NodeConfigWidget class represents individual "nodes" in a graphical scene. Each NodeConfigWidget object represents a configurable node that can be clicked to trigger a specific event. It inherits from QGraphicsObject, which allows it to be part of a QGraphicsScene, enabling interaction and graphical representation.

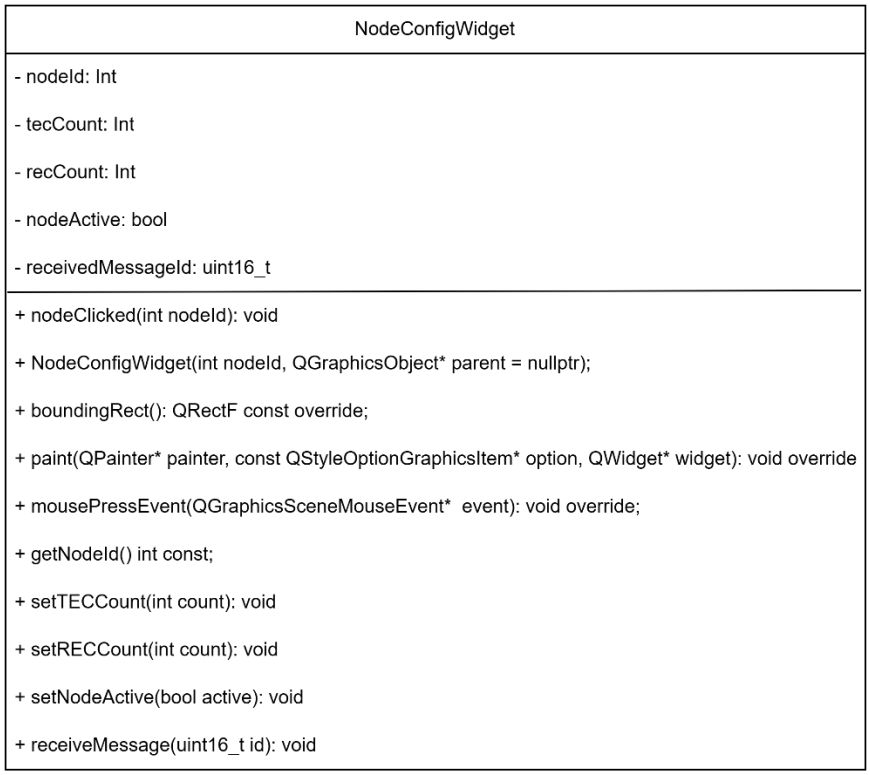

Figure 29. The NodeConfigWidget Class

- **Attributes:**

  - int nodeId: the unique identifier for each node.

- **Methods:**

  - Constructor: NodeConfigWidget(int nodeId, QGraphicsObject* parent = nullptr):iInitializes a node with the given ID.
  - Destructor: ~NodeConfigWidget(): cleans up resources when the node is deleted.
  - QRectF boundingRect() const: defines the boundaries for the node's graphical representation.
  - void paint(QPainter* painter, const QStyleOptionGraphicsItem* option, QWidget* widget) override: handles rendering of the node.
  - void mousePressEvent(QGraphicsSceneMouseEvent* event) override: processes mouse click events on the node.
  - int getNodeId() const: eturns the ID of the node.
  - Signal: void nodeClicked(int nodeId): emitted when the node is clicked, passing the node’s ID.

5.7.

The MessageDialog class provides a dialog interface for selecting target nodes to send a message to, starting from a "sender" node, it generates a unique 11-bit message ID based on the selected nodes and it inherits from QDialog, making it a pop-up dialog window.

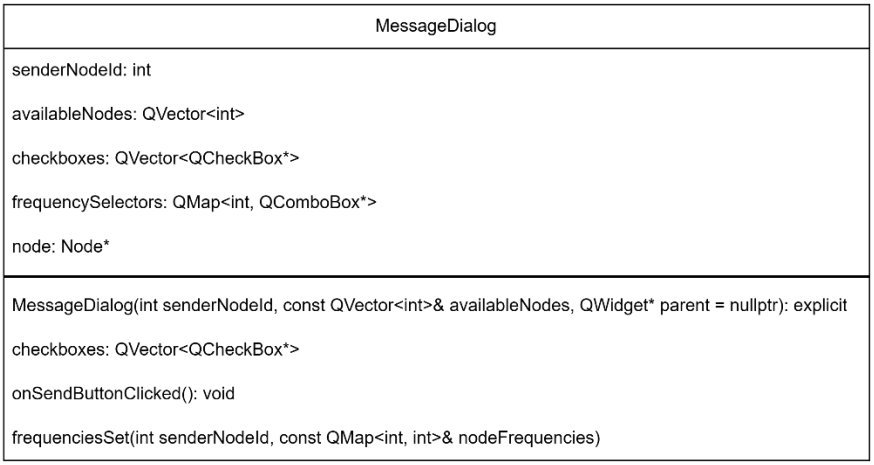

Figure 30. The MessageDialog Class

- **Attributes:**

  - int senderNodeId: the ID of the sender node initiating the message.
  - QVector<int> availableNodes: a list of node IDs available as targets for the message.
  - QVector<QCheckBox*> checkboxes: a collection of checkboxes for selecting target nodes.
  - Node* node: the corresponding node to the message Dialog
  - QMap<int, QComboBox> frequencySelectors: the selectors that choose the frequency for the corresponding message

- **Metods:**
  - Constructor: MessageDialog(int senderNodeId, const QVector<int>& availableNodes, QWidget* parent = nullptr): initializes the dialog with a sender node ID and a list of available nodes.
  - void onSendButtonClicked(): triggered when the "Send Message" button is clicked; collects selected nodes and sends the message.
  - int generate11BitID (int senderNodeId, const QVector<int>& targetNodeIds): generates an 11-bit ID for the message based on the sender and selected target node IDs.
  - frequenciesSet(int senderNodeId, const QMap<int, int>& nodeFrequencies): the function that processes the frequencies selected by the user
  - signal: void messageReady (int senderNodeId, int messageId, QVector<int> targetNodeIds): emitted when a message is ready, passing the sender ID, message ID, and target node IDs.

VI. Testing and Validation

- **Test Case 1: CAN Bus Arbitration with No Error Transmission**

Test Objective:
To verify the correct execution of arbitration, message transmission, CRC verification within a CAN bus simulation.

Preconditions:

- Nodes are registered: Node 1, Node 2, and Node 3.
- The messages with their frequencies are set up like this:
  - Node 1 sends messages to node 2 with a frequency of 5msg/min;
  - Node 2 sends messages to nodes 1 and 3 with a frequency of 5msg/min;
  - Node 3 sends messages to node 1 with a frequency of 1msg/min;

Test Steps:

1. **Start Simulation:** Initialize the CAN bus at a specific timestamp.
2. **Arbitration Process – Round 0:**
   - Perform bit-by-bit arbitration from the 11th to the 1st bit position.
   - Ensure the message ID 00000000010 wins as it is the one with the most priority, while the messages with ID 00100000101 and ID 01000000001 will remain in the queue as they lose the arbitration.
3. **Round 1:**
   - Following up, the message with ID 00100000101 will win, leaving the message from Node 3 in the pending messages list.
4. **Round 2:**
   - The message with the least priority now gets sent as there are no messages that have been added to the queue, as we are only in round 2.
   - This message will no longer show up in the arbitration, as the frequency of the message was 1msg/min, and the simulation will only last 1 minute.
5. **Rounds 3-59:**
   - The rest of the rounds will follow the arbitration of the messages from nodes 1 and 2, while the ones with no messages will get skipped as they are of no interest.
6. **Message Transmission:**
   - Transmit the winning message for each round.
   - Verify bit-stuffing is applied correctly during transmission.
   - Calculate and append the CRC.
7. **Message Reception:**
   - Simulate message reception by all nodes.
   - Check CRC validation on Node 3.
8. **Error Handling:**
   - Verify if the acknowledgment (ACK) bit is set by the receiving nodes.
   - If CRC verification passes, remove the message from the pending list.
   - Update nodes' error counters accordingly.

Expected Results:

- The correct message is transmitted after each winning arbitration.
- The CRC verification passes on the receiving nodes.
- Receiving nodes set the ACK bit correctly.
- Nodes' error counters are updated (they stay 0 in this case as this simulation is supposed to show the process with no errors):
    - Node 1: TEC: 0, REC: 0
    - Node 2: TEC: 0, REC: 0
    - Node 3: TEC: 0, REC: 0

Actual Results:
The actual results correspond to the expected ones as we can see in the log file:

```
ROUND: 0
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 0
       - message: 00100000101, sender ID: 2, initial round: 0
       - message: 01000000001, sender ID: 3, initial round: 0

  Arbitrating at bit position: 10
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 0
       - message: 00100000101, sender ID: 2, initial round: 0
       - message: 01000000001, sender ID: 3, initial round: 0

  Arbitrating at bit position: 9
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 0
       - message: 00100000101, sender ID: 2, initial round: 0

  Arbitrating at bit position: 8
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 0
.
.
Winner 00000000010, sender ID: 1, initial round: 0
Stuffed Message:
000001000010101001100110110100000101100110001110001001110001001110110101000110001001000101
1000001000001000001000001000001000001000001000001000001000001000001000001000001000000011
CRC: 0010010001011100
Node 2 received the message, CRC verification was valid.
 - ack bit was set to valid by node: 2
Winning message was received by at least one node. Removing it from the pending messages.
NODE ERROR COUNTERS :
- node 1  TEC: 0  REC: 0
- node 2  TEC: 0  REC: 0
- node 3  TEC: 0  REC: 0

ROUND: 1
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00100000101, sender ID: 2, initial round: 0
```

```
        - message: 01000000001, sender ID: 3, initial round: 0

  Arbitrating at bit position: 10
     Messages still in the race at the beginning:
        - message: 00100000101, sender ID: 2, initial round: 0
        - message: 01000000001, sender ID: 3, initial round: 0

  Arbitrating at bit position: 9
     Messages still in the race at the beginning:
        - message: 00100000101, sender ID: 2, initial round: 0
.
.
Winner 00100000101, sender ID: 2, initial round: 0
Stuffed Message:
001000001101101001100110110100000101100110001110001001110001001110110101000110000011010101
100000100000100000100000100000100000100000100000100000100000100000100000100000100000110
CRC: 0000101010111000
Node 1 received the message, CRC verification was valid.
 - ack bit was set to valid by node: 1
Node 3 received the message, CRC verification was valid.
Winning message was received by at least one node. Removing it from the pending messages.
NODE ERROR COUNTERS :
- node 1  TEC: 0  REC: 0
- node 2  TEC: 0  REC: 0
- node 3  TEC: 0  REC: 0

ROUND: 2
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
        - message: 01000000001, sender ID: 3, initial round: 0

  Arbitrating at bit position: 10
     Messages still in the race at the beginning:
        - message: 01000000001, sender ID: 3, initial round: 0

  Arbitrating at bit position: 9
     Messages still in the race at the beginning:
        - message: 01000000001, sender ID: 3, initial round: 0
.
.
Winner 01000000001, sender ID: 3, initial round: 0
Stuffed Message:
010000010001101001100110110100000101100110001110001001110001001110110101000110010010010110
100000100000100000100000100000100000100000100000100000100000100000100000100000100000011
CRC: 0100100101100100
Node 1 received the message, CRC verification was valid.
 - ack bit was set to valid by node: 1
Winning message was received by at least one node. Removing it from the pending messages.
NODE ERROR COUNTERS :
- node 1  TEC: 0  REC: 0
- node 2  TEC: 0  REC: 0
- node 3  TEC: 0  REC: 0

No messages for the current round: 3
No messages for the current round: 4
No messages for the current round: 5
```

No messages for the current round: 6
No messages for the current round: 7
No messages for the current round: 8
No messages for the current round: 9
No messages for the current round: 10.
.
.
And so on…

- ## Test Case 2: CAN Bus Arbitration with Repeated Transmission Failures when Receiver Node is Faulty

Test Objective:
To test the arbitration, retransmission, CRC validation, and error handling in a CAN bus simulation where repeated failures result in nodes being disabled due to exceeding error thresholds, when the receiver node is faulty.

Preconditions:

- Nodes are registered: Node 1 and Node 2.
  - Node 2 is faulty
- The messages with their frequencies are set up like this:
  - Node 1 sends messages to Node 2 with a frequency of 5msg/min;

Test Steps:

1. **Start Simulation:**
   - Initialize the CAN bus at a specific timestamp.
2. **Arbitration Process – Round 0:**
   - Perform bit-by-bit arbitration from the 11th to the 1st bit position.
   - Confirm that message ID 00000000010 from Node 1 wins arbitration.
3. **Message Transmission – Round 0:**
   - Transmit the winning message.
   - Apply bit-stuffing and append CRC 0101101111011000.
4. **Message Reception – Round 0:**
   - Simulate reception failure (CRC error).
   - Add the message back to the pending list.
   - Update error counters:
     - Node 1: TEC = 1, REC = 0.
     - Node 2: TEC = 0, REC = 1.
5. **Round 1 - Retry Transmission:**

- Retry Node 1’s message 00000000010.
- CRC error persists, message added back to the pending list.
- Update error counters:
  - Node 1: TEC = 2, REC = 0.
  - Node 2: TEC = 0, REC = 2.

6. **Round 2 - Retry Transmission:**
   - Retry the same message again.
   - CRC error persists, message added back to the pending list.
   - Update error counters:
     - Node 1: TEC = 3, REC = 0.
     - Node 2: TEC = 0, REC = 3.
7. **Round 3 - Final Attempt:**
   - Retry transmission for Node 1’s message.
   - CRC error persists.
   - Update error counters:
     - Node 1: TEC = 4, REC = 0.
     - Node 2: TEC = 0, REC = 4.
   - Node 2 is disabled due to REC reaching 4.
8. **Round 4 - Continuation After Node 2 Is Disabled:**
   - Arbitration continues with remaining messages in the queue.
   - Skip further attempts to send messages to the disabled Node 2.
9. **Error Handling and Final State:**
   - Confirm error counters and disabled node status.

Expected Results:

1. Arbitration selects Node 1’s message due to its higher priority.
2. CRC validation fails, resulting in retransmissions and error counter increments.
3. Node 2 reaches the maximum REC value (4) and is disabled.
4. Node 1 continues attempts, but further communication to Node 2 is skipped.
5. Final node error counters:
   - Node 1: TEC = 4, REC = 0.

- Node 2: TEC = 0, REC = 4 (disabled).

Actual Results:

The actual results match the expected ones as evidenced in the log file:

- Node 1 repeatedly retransmitted the message with CRC errors.
- Node 2 incremented its REC counter to 4 and was disabled as expected.

```
ROUND: 0
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 0
.
.
Winner 00000000010, sender ID: 1, initial round: 0
Stuffed Message:
000001000010010111110100011100011111010001111101001010100100000100101000110001110010110111101100000100000100000100000100000100000100000100000100000100000100000100000100000100000100000101
CRC: 0101101111011000
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1  TEC: 1  REC: 0
- node 2  TEC: 0  REC: 1

ROUND: 1
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 0
.
.
Winner 00000000010, sender ID: 1, initial round: 0
Stuffed Message:
000001000010010111110100011100011111010001111101001010100100000100101000110001110010110111101100000100000100000100000100000100000100000100000100000100000100000100000100000100000100000101
CRC: 0101101111011000
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1  TEC: 2  REC: 0
- node 2  TEC: 0  REC: 2

ROUND: 2
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 0
.
.
Winner 00000000010, sender ID: 1, initial round: 0
Stuffed Message:
000001000010010111110100011100011111010001111101001010100100000100101000110001110010110111101100000100000100000100000100000100000100000100000100000100000100000100000100000100000100000101
```

CRC: 0101101111011000
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 3 REC: 0
- node 2 TEC: 0 REC: 3

ROUND: 3
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 0
.
.
Winner 00000000010, sender ID: 1, initial round: 0
Stuffed Message:
000001000010010111110100011100011111010001111101001010100100000100101000110001110010110111101100000100000100000100000100000100000100000100000100000100000100000100000100000100000100000100000101
CRC: 0101101111011000
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 4 REC: 0
- node 2 TEC: 0 REC: 4
Node 2 has reached the maximum REC value. It will be disabled.

▪ Test Case 3: CAN Bus Arbitration with Repeated Transmission Failures when Sender Node is Faulty

Test Objective:

To test the arbitration, retransmission, CRC validation, and error handling in a CAN bus simulation where repeated failures result in nodes being disabled due to exceeding error thresholds, when the sender node is faulty.

Preconditions:

- Nodes are registered: Node 1 and Node 2.
  - Node 2 is faulty.
- The messages with their frequencies are set up like this:
  - Node 2 sends messages to Node 1 with a frequency of 5msg/min;

Test Steps:

1. **Start Simulation:**
   - Initialize the CAN bus at a specific timestamp.
2. **Arbitration Process – Round 0:**
   - Perform bit-by-bit arbitration from the 11th to the 1st bit position.
   - Transmit the winning message.
   - Apply bit-stuffing and append CRC 0000000000000000.

- Simulate reception failure (CRC error).
- Add the message back to the pending list.
- Update error counters:
  - **Node 1:** TEC = 0, REC = 1.
  - **Node 2:** TEC = 1, REC = 0.

5. **Round 1 - Retry Transmission:**
   - Retry **Node 2's message 00100000001**.
   - CRC error persists, message added back to the pending list.
   - Update error counters:
     - **Node 1:** TEC = 0, REC = 2.
     - **Node 2:** TEC = 2, REC = 0.
6. **Round 2 - Retry Transmission:**
   - Retry the same message again.
   - CRC error persists, message added back to the pending list.
   - Update error counters:
     - **Node 1:** TEC = 0, REC = 3.
     - **Node 2:** TEC = 3, REC = 0.
7. **Round 3 - Final Retry Attempt:**
   - Retry transmission for **Node 2's message**.
   - CRC error persists again.
   - Update error counters:
     - **Node 1:** TEC = 0, REC = 4 (disabled).
     - **Node 2:** TEC = 4, REC = 0.
   - **Node 1 is disabled** as REC reaches the threshold of 4.
8. **Round 4 - Continuation After Node 1 Is Disabled:**
   - Arbitration continues with **Node 2** attempting transmission.
   - Transmission fails again due to no receivers.

Expected Results:

1. Arbitration selects **Node 2's message** due to no contention.

2. CRC validation repeatedly fails, resulting in retransmissions and error counter increments.
3. **Node 1** reaches the maximum REC value (4) and is **disabled** in **Round 3**.
4. **Node 2** continues attempts until it finds no other receivers.
5. Final node error counters:
    - **Node 1:** TEC = 0, REC = 4 (**disabled**).
    - **Node 2:** TEC = 4, REC = 0.

Actual Results:

The actual results correspond to the expected ones as evidenced in the log file:

- **Node 1** incremented its REC counter to 4 and was **disabled** in **Round 3**.
- **Node 2** incremented its TEC counter to 4, stopped transmission due to no receivers.

```
ROUND: 0
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00100000001, sender ID: 2, initial round: 0
.
.
Winner 00100000001, sender ID: 2, initial round: 0
Stuffed Message:
00100100100110110100101011100001110111100110101111000101101000010110010011110000010000010000
00100000100000100000100000100000100000100000100000100000100000100000100000100000100000100010
CRC: 0000000000000000
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1  TEC: 0  REC: 1
- node 2  TEC: 1  REC: 0

ROUND: 1
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00100000001, sender ID: 2, initial round: 0
.
.
Winner 00100000001, sender ID: 2, initial round: 0
Stuffed Message:
00100100100110110100101011100001110111100110101111000101101000010110010011110000010000010000
00100000100000100000100000100000100000100000100000100000100000100000100000100000100000100010
CRC: 0000000000000000
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1  TEC: 0  REC: 2
- node 2  TEC: 2  REC: 0
```

ROUND: 2
  Arbitrating at bit position: 11
    Messages still in the race at the beginning:
      - message: 00100000001, sender ID: 2, initial round: 0
.
.
Winner 00100000001, sender ID: 2, initial round: 0
Stuffed Message:
001001001001101101001010111000011101111100110101111000101101000001011001001111000001000001000
0010000010000010000010000010000010000010000010000010000010000010000010000010000010000010000010
CRC: 0000000000000000
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 0 REC: 3
- node 2 TEC: 3 REC: 0

ROUND: 3
  Arbitrating at bit position: 11
    Messages still in the race at the beginning:
      - message: 00100000001, sender ID: 2, initial round: 0
.
.
Winner 00100000001, sender ID: 2, initial round: 0
Stuffed Message:
001001001001101101001010111000011101111100110101111000101101000001011001001111000001000001000
0010000010000010000010000010000010000010000010000010000010000010000010000010000010000010000010
CRC: 0000000000000000
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 0 REC: 4
Node 1 has reached the maximum REC value. It will be disabled.
- node 2 TEC: 4 REC: 0

ROUND: 4
  Arbitrating at bit position: 11
    Messages still in the race at the beginning:
      - message: 00100000001, sender ID: 2, initial round: 0
.
.
Winner 00100000001, sender ID: 2, initial round: 0
Stuffed Message:
001001001001101101001010111000011101111100110101111000101101000001011001001111000001000001000
0010000010000010000010000010000010000010000010000010000010000010000010000010000010000010000010
CRC: 0000000000000000
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 2 TEC: 4 REC: 0

- **Test Case 4: CAN Bus Arbitration with Repeated Transmission Failures and Transmission Succes, when Multiple Nodes (faulty or not) are involved – Realistic Simulation**

Test Objective:

To test the arbitration, retransmission, CRC validation, and error handling in a CAN bus simulation involving multiple nodes, including scenarios where repeated failures lead to node disabling.

Preconditions:

- Nodes are registered: Node 1, Node 2, Node 3, and Node 4.
- Message Frequency Configuration:
  - Node 1 sends messages to Node 3 at 5 msg/min.
  - Node 2 sends messages to Node 3 at 5 msg/min.
  - Node 3 sends messages to Node 1 at 1 msg/min.
  - Node 4 sends messages to Node 1 at 5 msg/min.

Test Steps:

1. **Start Simulation:**
   - Initialize the CAN bus at a specific timestamp.
2. **Arbitration Process – Round 0:**
   - Perform bit-by-bit arbitration from the 11th to the 1st bit position.
   - Confirm message ID 00000000110 from Node 1 wins arbitration due to the highest priority.
   - Transmit the winning message.
   - Node 3 successfully receives the message and sets the ACK bit.
   - Remove the message from pending messages.
   - Update error counters (no need here)
3. **Round 1 - Arbitration for Remaining Messages:**
   - Arbitration prioritizes Node 2's message 00100000100.
   - CRC validation fails.
   - Update error counters:
     - Node 2: TEC = 1, REC = 1.
     - Node 3: TEC = 0, REC = 1.
4. **Rounds 2-4 - Retransmission Attempts:**
   - Retry Node 2's message 00100000100 multiple times.

- CRC failures persist, leading to:
    - Node 2: TEC = 4, REC = 1.
    - Node 3: TEC = 0, REC = 4 (disabled).

5. **Round 5 - Retry with Remaining Nodes:**
    - Arbitration prioritizes Node 4's message 01100000101.
    - Node 1 successfully receives the message and sets the ACK bit.
    - Remove the message from pending messages.
    - Error counters:
        - Node 1: TEC = 0, REC = 0.
        - Node 4: TEC = 0, REC = 0.
6. **Rounds 6-48 - Continuing Arbitration and Transmission:**
    - Repeat arbitration and transmissions for remaining messages.
    - Node 2 continues attempts until it is disabled in Round 14 after REC = 4.
    - Node 4 continues successful transmissions with Node 1 receiving valid messages.
7. **Final Error Handling and State:**
    - Confirm error counters and disabled nodes:
        - Node 1: TEC = 3, REC = 0.
        - Node 2: TEC = 4, REC = 4 (disabled).
        - Node 3: TEC = 0, REC = 4 (disabled).
        - Node 4: TEC = 0, REC = 0.

Expected Results:

1. Arbitration prioritizes messages based on ID values, starting with Node 1's message 00000000110.
2. CRC validation failures lead to retransmissions, and error counters increment appropriately.
3. Node 3 is disabled after reaching REC = 4 in Round 4.
4. Node 2 is disabled after reaching REC = 4 in Round 14.
5. Node 4 continues to send valid messages received by Node 1 without errors.
6. Final node error counters reflect expected states:
    - Node 1: TEC = 3, REC = 0.

- Node 2: TEC = 4, REC = 4 (disabled).
- Node 3: TEC = 0, REC = 4 (disabled).
- Node 4: TEC = 0, REC = 0.

Actual Results:

The actual results match the expected ones as evidenced in the log file:

- Arbitration prioritized Node 1's message followed by others based on IDs.
- CRC validation errors resulted in retransmissions, incrementing error counters.
- Node 3 and Node 2 were disabled after reaching REC thresholds.
- Node 4 continued sending valid messages received successfully by Node 1.

```
ROUND: 0
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00000000110, sender ID: 1, initial round: 0
       - message: 00100000100, sender ID: 2, initial round: 0
       - message: 01000000001, sender ID: 3, initial round: 0
       - message: 01100000101, sender ID: 4, initial round: 0

  Arbitrating at bit position: 10
     Messages still in the race at the beginning:
       - message: 00000000110, sender ID: 1, initial round: 0
       - message: 00100000100, sender ID: 2, initial round: 0
       - message: 01000000001, sender ID: 3, initial round: 0
       - message: 01100000101, sender ID: 4, initial round: 0

  Arbitrating at bit position: 9
     Messages still in the race at the beginning:
       - message: 00000000110, sender ID: 1, initial round: 0
       - message: 00100000100, sender ID: 2, initial round: 0

  Arbitrating at bit position: 8
     Messages still in the race at the beginning:
       - message: 00000000110, sender ID: 1, initial round: 0
.
.
Winner 00000000110, sender ID: 1, initial round: 0
Stuffed Message:
000001000110011111000110011001111000001011101001101100000110001001000011111001100010011001111
110000001000001000001000001000001000001000001000001000001000001000001000001000001000000100000101
CRC: 0100110011111000
Message was not received.
Node 3 received the message, CRC verification was valid.
 - ack bit was set to valid by node: 3
Winning message was received by at least one node. Removing it from the pending messages.
NODE ERROR COUNTERS :
- node 1  TEC: 0  REC: 0
- node 2  TEC: 0  REC: 1
- node 3  TEC: 0  REC: 0
```

- node 4 TEC: 0 REC: 0

ROUND: 1
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00100000100, sender ID: 2, initial round: 0
       - message: 01000000001, sender ID: 3, initial round: 0
       - message: 01100000101, sender ID: 4, initial round: 0

  Arbitrating at bit position: 10
     Messages still in the race at the beginning:
       - message: 00100000100, sender ID: 2, initial round: 0
       - message: 01000000001, sender ID: 3, initial round: 0
       - message: 01100000101, sender ID: 4, initial round: 0

  Arbitrating at bit position: 9
     Messages still in the race at the beginning:
       - message: 00100000100, sender ID: 2, initial round: 0
.
.
Winner 00100000100, sender ID: 2, initial round: 0
Stuffed Message:
00100100110001111100011001100111100000101110100110110000011000100100001111001100000100000100
000010000010000010000010000010000010000010000010000010000010000010000010000010000100000110
CRC: 0000000000000000
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 0 REC: 0
- node 2 TEC: 1 REC: 1
- node 3 TEC: 0 REC: 1
- node 4 TEC: 0 REC: 0

ROUND: 2
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00100000100, sender ID: 2, initial round: 0
       - message: 01000000001, sender ID: 3, initial round: 0
       - message: 01100000101, sender ID: 4, initial round: 0

  Arbitrating at bit position: 10
     Messages still in the race at the beginning:
       - message: 00100000100, sender ID: 2, initial round: 0
       - message: 01000000001, sender ID: 3, initial round: 0
       - message: 01100000101, sender ID: 4, initial round: 0

  Arbitrating at bit position: 9
     Messages still in the race at the beginning:
       - message: 00100000100, sender ID: 2, initial round: 0
.
.
Winner 00100000100, sender ID: 2, initial round: 0
Stuffed Message:
00100100110001111100011001100111100000101110100110110000011000100100001111001100000100000100
000010000010000010000010000010000010000010000010000010000010000010000010000010000100000110
CRC: 0000000000000000

Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 0 REC: 0
- node 2 TEC: 2 REC: 1
- node 3 TEC: 0 REC: 2
- node 4 TEC: 0 REC: 0

ROUND: 3
  Arbitrating at bit position: 11
    Messages still in the race at the beginning:
      - message: 00100000100, sender ID: 2, initial round: 0
      - message: 01000000001, sender ID: 3, initial round: 0
      - message: 01100000101, sender ID: 4, initial round: 0

  Arbitrating at bit position: 10
    Messages still in the race at the beginning:
      - message: 00100000100, sender ID: 2, initial round: 0
      - message: 01000000001, sender ID: 3, initial round: 0
      - message: 01100000101, sender ID: 4, initial round: 0

  Arbitrating at bit position: 9
    Messages still in the race at the beginning:
      - message: 00100000100, sender ID: 2, initial round: 0
.
.
Winner 00100000100, sender ID: 2, initial round: 0
Stuffed Message:
0010010011000111110001100110011110000010111010011011000001100010010000111100110000010000010
0000100000100000100000100000100000100000100000100000100000100000100000100000100000100000110
CRC: 0000000000000000
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 0 REC: 0
- node 2 TEC: 3 REC: 1
- node 3 TEC: 0 REC: 3
- node 4 TEC: 0 REC: 0

ROUND: 4
  Arbitrating at bit position: 11
    Messages still in the race at the beginning:
      - message: 00100000100, sender ID: 2, initial round: 0
      - message: 01000000001, sender ID: 3, initial round: 0
      - message: 01100000101, sender ID: 4, initial round: 0

  Arbitrating at bit position: 10
    Messages still in the race at the beginning:
      - message: 00100000100, sender ID: 2, initial round: 0
      - message: 01000000001, sender ID: 3, initial round: 0
      - message: 01100000101, sender ID: 4, initial round: 0

  Arbitrating at bit position: 9
    Messages still in the race at the beginning:
      - message: 00100000100, sender ID: 2, initial round: 0
.

.
Winner 00100000100, sender ID: 2, initial round: 0
Stuffed Message:
00100100110001111100011001100111100000101110100110110000011000100100001111001100000100000100000010000010000001000000100000010000001000000100000010000001000000100000010000001000000100000010000001000000100000010000001000000100000110
CRC: 0000000000000000
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 0 REC: 0
- node 2 TEC: 4 REC: 1
- node 3 TEC: 0 REC: 4
Node 3 has reached the maximum REC value. It will be disabled.
- node 4 TEC: 0 REC: 0

ROUND: 5
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00100000100, sender ID: 2, initial round: 0
       - message: 01100000101, sender ID: 4, initial round: 0

  Arbitrating at bit position: 10
     Messages still in the race at the beginning:
       - message: 00100000100, sender ID: 2, initial round: 0
       - message: 01100000101, sender ID: 4, initial round: 0

  Arbitrating at bit position: 9
     Messages still in the race at the beginning:
       - message: 00100000100, sender ID: 2, initial round: 0
.
.
Winner 00100000100, sender ID: 2, initial round: 0
Stuffed Message:
00100100110001111100011001100111100000101110100110110000011000100100001111001100000100000100000010000010000001000000100000010000001000000100000010000001000000100000010000001000000100000010000001000000100000110
CRC: 0000000000000000
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 0 REC: 0
- node 2 TEC: 4 REC: 1
- node 4 TEC: 0 REC: 0

ROUND: 6
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 01100000101, sender ID: 4, initial round: 0
.
.
Winner 01100000101, sender ID: 4, initial round: 0
Stuffed Message:
01100000110101111100011001100111100000101110100110110000011000100100001111001100010011001110010000010000001000000100000010000001000000100000010000001000000100000010000001000000100000010000001000000100100
CRC: 0100110011100110
Node 1 received the message, CRC verification was valid.
 - ack bit was set to valid by node: 1
Winning message was received by at least one node. Removing it from the pending messages.

NODE ERROR COUNTERS :
- node 1 TEC: 0 REC: 0
- node 2 TEC: 4 REC: 1
- node 4 TEC: 0 REC: 0

No messages for the current round: 7
…

ROUND: 12
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 12
       - message: 01100000001, sender ID: 4, initial round: 12

  Arbitrating at bit position: 10
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 12
       - message: 01100000001, sender ID: 4, initial round: 12

  Arbitrating at bit position: 9
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 12
.
.
Winner 00000000010, sender ID: 1, initial round: 12
Stuffed Message:
0000010000100111110001100110011110000010111010011011000001100010010000111100110001111000010
0110000010000010000010000010000010000011100000100000100000100000100000100000100001
CRC: 0111100001001100
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 1 REC: 0
- node 2 TEC: 4 REC: 2
- node 4 TEC: 0 REC: 0

ROUND: 13
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 12
       - message: 01100000001, sender ID: 4, initial round: 12

  Arbitrating at bit position: 10
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 12
       - message: 01100000001, sender ID: 4, initial round: 12

  Arbitrating at bit position: 9
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 12
.
.
Winner 00000000010, sender ID: 1, initial round: 12
Stuffed Message:
0000010000100111110001100110011110000010111010011011000001100010010000111100110001111000010
0110000010000010000010000010000010000011100000100000100000100000100000100000100001

CRC: 0111100001001100
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 2 REC: 0
- node 2 TEC: 4 REC: 3
- node 4 TEC: 0 REC: 0

ROUND: 14
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 12
       - message: 01100000001, sender ID: 4, initial round: 12

  Arbitrating at bit position: 10
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 12
       - message: 01100000001, sender ID: 4, initial round: 12

  Arbitrating at bit position: 9
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 12
.
.
Winner 00000000010, sender ID: 1, initial round: 12
Stuffed Message:
000001000010011111000110011001111000001011101001101100000110001001000011110011000111100001
011000001000001000001000001000001000001110000010000010000010000010000010000010001
CRC: 0111100001001100
Message was not received.
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 3 REC: 0
- node 2 TEC: 4 REC: 4
Node 2 has reached the maximum REC value. It will be disabled.
- node 4 TEC: 0 REC: 0

ROUND: 15
  Arbitrating at bit position: 11
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 12
       - message: 01100000001, sender ID: 4, initial round: 12

  Arbitrating at bit position: 10
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 12
       - message: 01100000001, sender ID: 4, initial round: 12

  Arbitrating at bit position: 9
     Messages still in the race at the beginning:
       - message: 00000000010, sender ID: 1, initial round: 12
.
.
Winner 00000000010, sender ID: 1, initial round: 12

Stuffed Message:
000001000010011111000110011001111000001011101001101100000110001001000011110011000111100001
011000001000001000001000001000001000001110000010000010000010000010000010000010001
CRC: 0111100001001100
No nodes received the winning message. Adding it back to the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 3 REC: 0
- node 4 TEC: 0 REC: 0

ROUND: 16
  Arbitrating at bit position: 11
      Messages still in the race at the beginning:
        - message: 01100000001, sender ID: 4, initial round: 12
.
.
Winner 01100000001, sender ID: 4, initial round: 12
Stuffed Message:
011000001001011111000110011001111000001011101001101100000110001001000011110011000111100001
100100000100000100000100000100000100001100000100000100000100000100000100000100100
CRC: 0111100001010010
Node 1 received the message, CRC verification was valid.
 - ack bit was set to valid by node: 1
Winning message was received by at least one node. Removing it from the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 3 REC: 0
- node 4 TEC: 0 REC: 0

(same thing happens)

ROUND: 48
  Arbitrating at bit position: 11
      Messages still in the race at the beginning:
        - message: 01100000001, sender ID: 4, initial round: 48
.
.
Winner 01100000001, sender ID: 4, initial round: 48
Stuffed Message:
011000001001011111000110011001111000001011101001101100000110001001000011110011000111100001
100100000100000100000100000100000100110000010000010000010000010000010000010000010000100
CRC: 0111100001010010
Node 1 received the message, CRC verification was valid.
 - ack bit was set to valid by node: 1
Winning message was received by at least one node. Removing it from the pending messages.
NODE ERROR COUNTERS :
- node 1 TEC: 3 REC: 0
- node 4 TEC: 0 REC: 0

VII. Conclusions

This CAN bus simulation provides a detailed and interactive representation of the key principles behind the Controller Area Network (CAN) protocol. By implementing core components like nodes, messages, and the CAN bus, the simulation replicates the primary operations of a real-world CAN system, such as message transmission, arbitration, and error detection. Developed in C++ for its high performance and low-level memory control, the simulation is optimized to handle the real-time characteristics of CAN communication, allowing for accurate and efficient message handling.

The inclusion of a graphical user interface, built using the Qt framework, enhances the user experience by offering visual representations of the CAN network. This GUI allows users to configure nodes, initiate message transmissions, and track the behaviour of messages across the CAN bus. Users can easily interact with the system, visualizing the flow of data and observing how different nodes communicate with each other in real-time. The system also highlights essential aspects like error detection mechanisms, including CRC verification and bit stuffing, which are integral to ensuring data integrity in a real CAN system.

The modular design of the simulation ensures that it can be customized, making it a versatile tool for both educational and experimental purposes. By simulating real-world CAN bus operations in a controlled environment, this system offers valuable insights into the functioning of distributed systems and the underlying communication protocols. In summary, this CAN simulation serves as an accessible yet powerful tool for understanding and experimenting with CAN protocol concepts in a hands-on, interactive manner.

BIBLIOGRAPHY

[1] Jesal Shah, “Understanding CAN: A Beginner's Guide to the Controller Area Network Protocol” https://www.circuitbread.com/tutorials/understanding-can-a-beginners-guide-to-the-controller-area-network-protocol

[2] Steve Corrigan, Texas Instruments, “Introduction to the Controller Area Network (CAN)”, August 2002–Revised May 2016
https://www.ti.com/lit/an/sloa101b/sloa101b.pdf

[3] Stephen St. Michael, “Introduction to CAN (Controller Area Network)”, February 19, 2019
https://www.allaboutcircuits.com/technical-articles/introduction-to-can-controller-area-network/