2.0 Final Design

This section provides the final design of the network protocol, system-level overview and detailed description of individual software modules.

2.1 Network Protocol (author: Dixin Wu)

Our mesh networking software package uses spanning trees as the underlying network topology, where nodes in the network form parent-child relationships. The root of the tree is usually the gateway device. Each node can have at most one parent node but multiple child nodes. The following sections describe our design of the high-level protocols for network formation, data gathering, and failure handling in a spanning tree network. The software implementation of the protocol is described in detail in Section 2.3.3 Forwarding Engine and 2.3.2 Message Processor.

2.1.1 Network Formation (author: Dixin Wu)

A spanning tree starts with a root which is the gateway. New nodes join the tree by establishing connections with an existing node in the network. The first new node to join the network sends messages to the gateway and becomes a child node of the gateway.

To accomplish a spanning tree topology, we used the following three-way handshake process (Figure 5) to establish a connection.

1. When Node A wants to join the network, it broadcasts a join request to look for a suitable parent node.

2. Nodes already in the network are eligible to become a parent of Node A. Upon receiving its join requests, they each reply to Node A with a join acknowledgement indicating their addresses and costs to the gateway.

3. Node A listens for 10 seconds to receive those aforementioned join acknowledgements. After 10 seconds, among all nodes from which Node A received a join acknowledgement, it picks a parent node, which has the least cost (hop counts) to the gateway and offers an acceptable link quality. Then Node A sends a join confirmation message to the best parent node to become its child node.

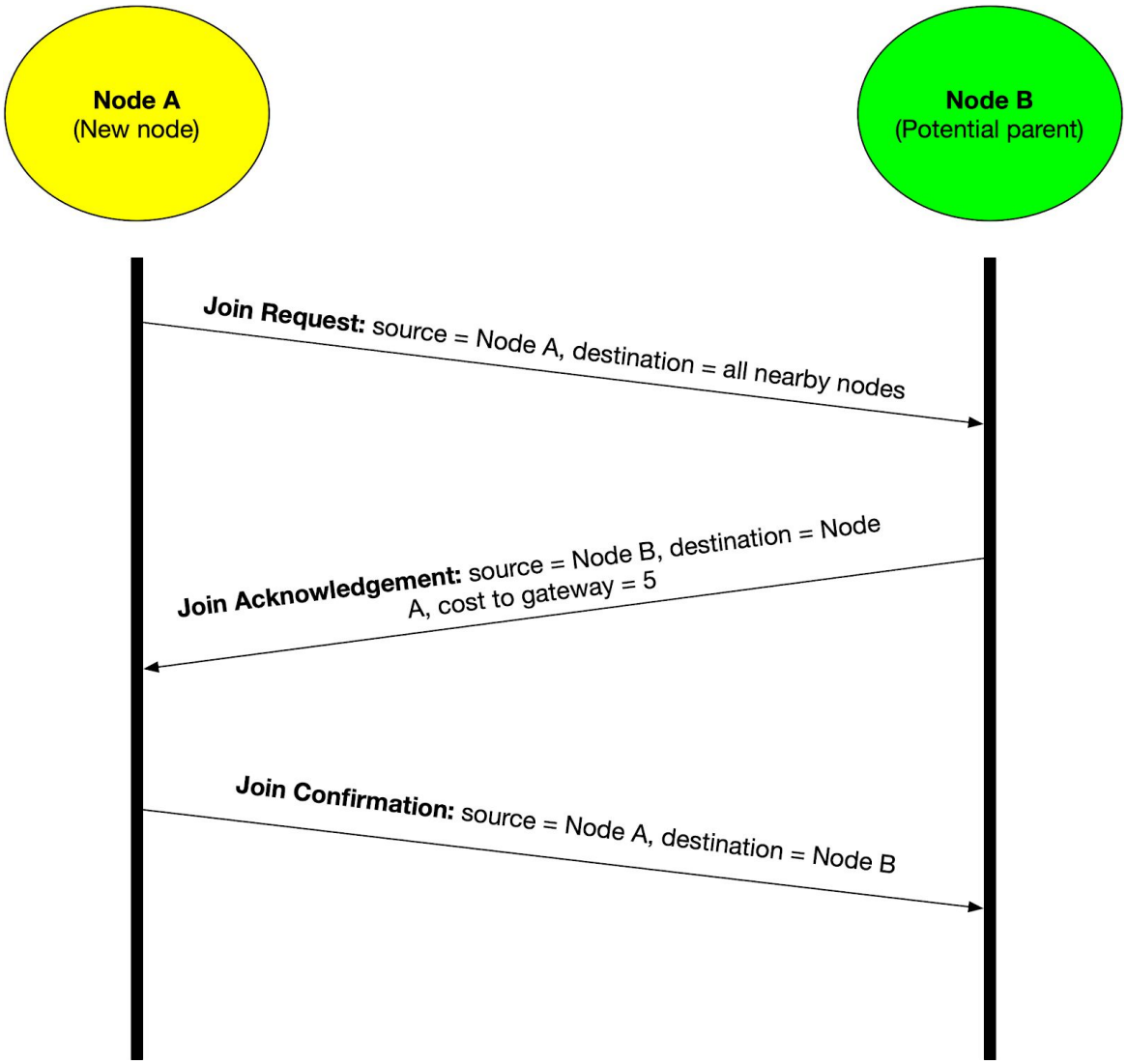

Figure 5. A typical three-way handshake process when a new node (Node A) joins the network by connecting to a parent node (Node B). After the process is complete, Node A becomes a child node of Node B.

2.1.2 Data Gathering (author: Hongyi Yang)

The software package provides a feature called Gateway Request, which allows the gateway to collect data periodically from all nodes within the network. Users of the library can set a timer value when they set up the gateway. Once the timer expires upon timeout, five steps will be performed to gather data:

1. The gateway sends out GatewayReq messages to all its children.
2. A node that receives a GatewayReq forwards this message to all its children.
3. A node that receives the GatewayReq uses a callback function defined by the user to fetch the data .
4. The nodes then put the data into NodeReply messages and send them to their respective parents.
5. All parents forward the NodeReply messages they received upward in the spanning tree.

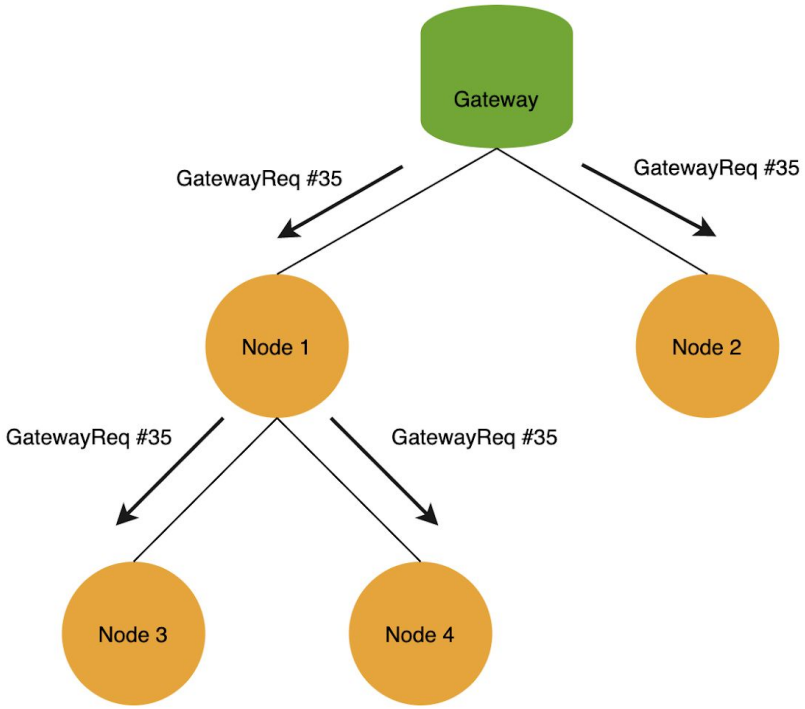

Figure 6. Gateway requests data from nodes (Steps 1 and 2)

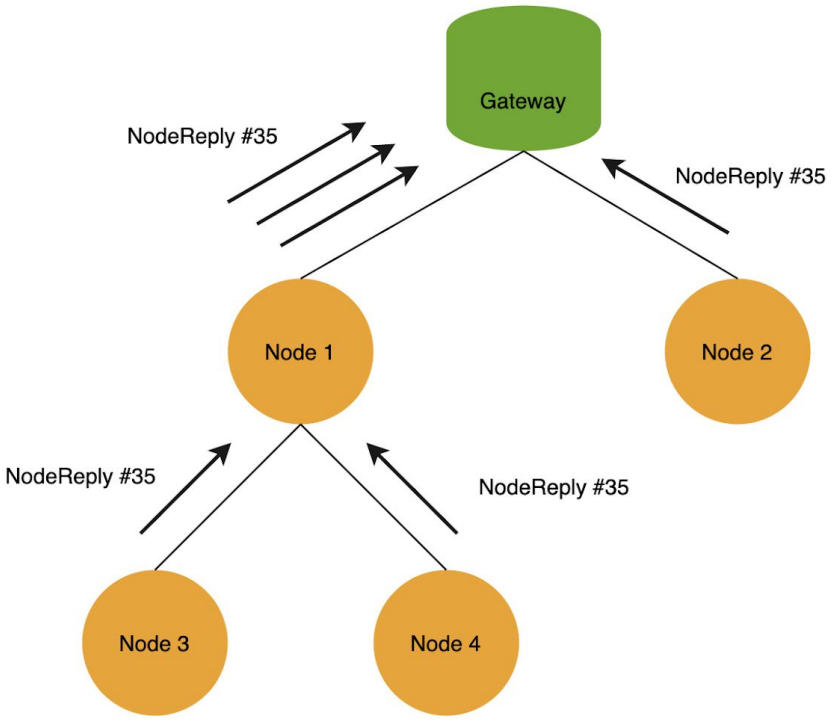

Figure 7. Nodes reply with data to the gateway (Steps 4 and 5)

To identify requests sent out at different times, the gateway adds a sequence number when it sends out GatewayRequest messages. A NodeReply message which is sent in response to a GatewayRequest message includes the sequence number of the GatewayRequest message. In this way, we can prevent delayed messages from causing confusion to the users of the library.

2.1.3 Failure Detection and Recovery (author: Yizhi Xu)

The network has a self-healing feature. When a connection between two nodes fails unexpectedly, the node performs the following steps.

A child monitors its uplink status by sending a “CheckAlive” message periodically to its parent. If the parent fails to send back a “ReplyAlive” message within 30 seconds, the child assumes a link failure and tries to recover a parent. The node will try to rejoin the network by restarting the discovery process (described in 2.1.1) and looking for a parent (path to gateway implied). Once the connection is established between the node and the new parent, the node can communicate with the gateway, hence the connectivity is restored.

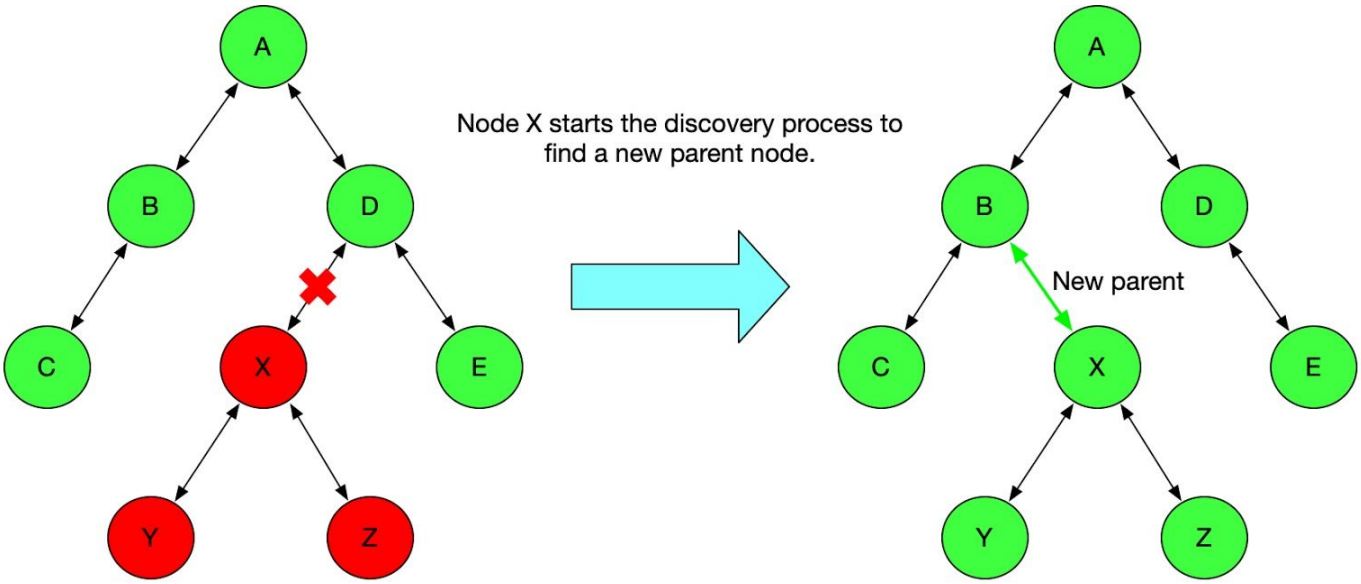

Figure 8. Node X detects a link failure between itself and Node D, then it starts the discovery process and finds a new parent node B.

2.2 System-level Overview (author: Yizhi Xu)

As a network software library, our project provides a set of functions, algorithms, and protocols for users to create nodes in a mesh network for IoT applications. It consists of an API (application programming interface) frontend, a Forwarding Engine, a Message Processor and a Generic Driver Interface (Table 6).

Table 6. Descriptions of modules

Module Name	Role	User Visibility
API frontend	Provides high-level functions for application developers to perform network operations and manipulate resources.	Users have full access to functions and variables in the module.
Forwarding Engine	Manages the network topology and makes forwarding decisions.	Internal operations in the module are not visible to users.
Message Processor	Provides message serialization (encoding) and deserialization.	Predefined message formats in the module are not visible to users.
Generic Driver Interface	Provides software abstraction of wireless communication hardware. The application developer needs to provide his or her implementation of the driver interface depending on the specific transceiver.	Some abstract functions in the module are visible to users and require users to override them.

A typical workflow (marked in blue in Figure 9) for sending a message consists of the following four steps:

1. The developer provides the application logic by calling functions offered by the API frontend. The API frontend then calls functions of the Forwarding Engine to execute user commands. To send a string to a remote node, the API frontend takes the message payload and destination address to the Forwarding Engine to find an available path.

2. Upon receiving the message from the API front end, the Forwarding Engine determines the next hop in the path. If the next hop is available, the message is sent to the Message Processor for serializing.

3. In the Message Processor, a protocol header which contains the sender information will be appended to the message. Next, the entire message including the header is serialized to a stream of bytes and passed to the driver module.

4. The Generic Driver Interface has a specific driver implementation provided by the developer. As the byte stream arrives, the user-defined sending function will pass them to the underlying hardware to transmit them as an electromagnetic wave.

Reversely, a typical workflow (marked in orange in Figure 9) for receiving a message consists of the following three steps:

1. The Forwarding Engine spends most time blocking on receiving incoming messages. In other words, it invokes its receiving function periodically to catch a message. If an incoming message arrives at the transceiver hardware, a stream of bytes will be read by the driver module and passed to the Message Processor.

2. The Message Processor parses the received bytes to extract the header and payload. The extracted network information and data are then stored in data structures for the Forwarding Engine to process.

3. The Forwarding Engine analyzes the sender information and determines if the network status has changed. After processing the network information, it returns the data to the API frontend where the user application can use it.

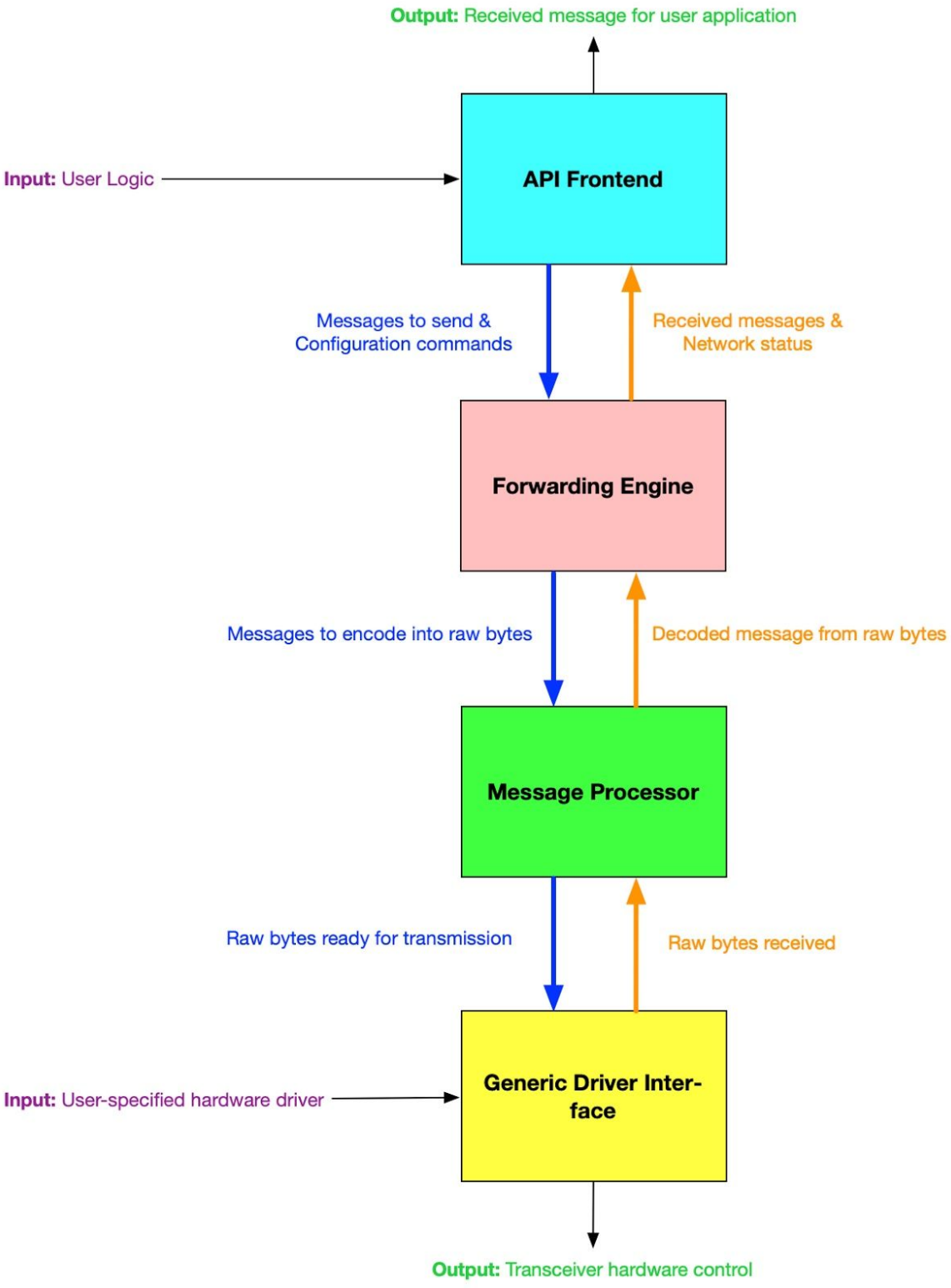

Figure 9. System-level design overview.

2.3 Software Modules

This section describes detailed designs of each software module from the low level driver interface to the API frontend.

2.3.1 Generic Driver Interface (author: Yizhi Xu)

The Generic Driver Interface is a software abstraction of LoRa transceivers. It decouples the Message Processor and the actual hardware driver. Table 7 shows the function defined in the interface. Parameters to those functions include destination address, which is a 2-byte address uniquely assigned to each node (Details in 2.3.2 Message Processor).

Table 7. Interface functions.

Function	Parameters	Return Value	Description
byte recv( )	None	Return one byte of message received.	Read one byte of data from the transceiver that is sent by other nodes to this node.
int send( )	*destination address* *message payload* *message length*	Return the number of bytes sent.	Send a message to the destination.
int available( )	None	Return the number of bytes that are available to receive.	Get the number of available bytes in the receive buffer.
int getRSSI( )	None	Return the RSSI value for the last message the node has received.	Read the RSSI value from the transceiver for the last message received.

Generic Driver Interfaces assumes that the underlying driver uses unicasting for packet transmission and receiving. For example, the recv() function should only return bytes in a packet which has a destination address of the node or a broadcasting address. By using unicasting, a node can discard packets that are sent to other nodes at the driver level. Therefore, it avoids unnecessary decoding at the Message Processor and saves power.

Some transceivers such as Ebyte E22 900MHz have already provided hardware support for unicasting. For other transceivers, their hardware drivers which implement the Generic Driver Interface should provide address filtering in the implementation. For example, a developer can append a destination address at the beginning of an outgoing packet. Receivers of the message examine the destination address and decide whether to accept the packet.

Hardware Drivers

Hardware drivers implement the functions defined in the Generic Driver Interface specifically for each transceiver. Developers are required to use their own hardware drivers to use the library.

In our project, we developed hardware drivers for Ebyte and Adafruit transceivers for testing. For Ebyte E22, the hardware driver implements the Generic Driver Interface using UART (Universal Asynchronous Receiver/Transmitter) to interact with the transceiver. For Adafruit, the hardware driver uses an open source "arduino-LoRa" library [11] to implement the interface. However, unlike Ebyte E22, the firmware of Adafruit transceivers does not support address filtering. Therefore, in order to implement the recv() function, the driver needs to filter the messages according to the target address after it receives the messages from the transceiver.

2.3.2 Message Processor (author: Hongyi Yang)

The Message Processor is a component that connects the high-level Forwarding Engine to the low-level hardware driver. This module is responsible for serializing outgoing messages to bytes and passing them to the underlying driver for transmission. Meanwhile, upon receiving messages, it processes raw bytes from the driver into structured packets which are readable by the Forwarding Engine.

Table 8. Key functions in the Message Processor.

Function	Parameters	Description
receiveMessage( )	*driver,* *timeout*	Reads raw bytes from the Generic Driver module using the argument *driver*, converts them to a decoded message, and returns the message. The process should be finished within the time limit specified by the argument *timeout*. **Return:** A message object
send( )	*driver,* *destAddr*	Uses the Generic Driver Interface provided by the argument *driver* to send a message to the address *destAddr*. **Return:** The number of bytes sent

All communication in the network uses messages with our predefined structure. As shown in Figure 10, we have seven different message types. The message structure takes memory usage into consideration. For example, numerical values such as *hopsToGateway*, *seqNum*, and *datalength* are represented by one byte instead of a 4-byte integer. This saves memory and time in transmission. Besides, we decided to assign a 2-byte address to each node in the network. The reason is that it can save memory usage. Two bytes provide enough addresses to make the network scalable to up to 65534 ($2^{16} - 2$) nodes and gateways. Address 0xFFFF is reserved for the broadcast address. To distinguish gateways from regular nodes, gateways use addresses which have their first bit set to 1.

- **Gateway address:** 0x8000 to 0xFFFE (32768 to 65535)
- **Broadcast address:** 0xFFFF
- **Node address:** 0x0000 to 0x7FFF (0 to 32767)

Nodes are given a timeout value when receiving messages in order to limit the time spent on waiting. Since the devices we use do not have interrupt functionality, we actively check the buffer of the device to read data.

When the device is not ready or the message is lost, having this value can prevent the program from getting stuck so it can jump out and continue to process other tasks.

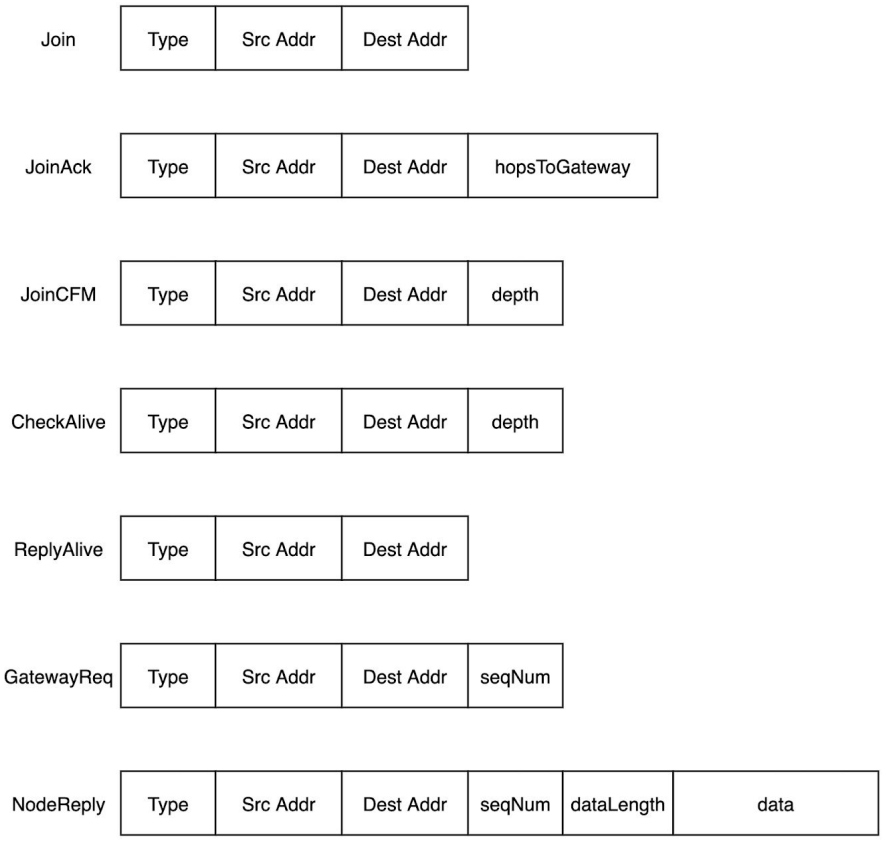

Figure 10. Message Structure Design

Table 9. Messages used in network operations

Message Type	**Network Operation**	**Description**
Join, JoinAck, JoinCFM	Network Formation	Messages for the three-way handshake joining process
CheckAlive, ReplyAlive	Self-healing	Messages to detect failure nodes
GatewayReq, NodeReply	Data Gathering	Messages that contain the data collected by nodes in the network

2.3.3 Forwarding Engine (author: Dixin Wu)

The Forwarding engine is the major component in the software package, which is responsible for executing the spanning tree network protocol (Section 2.1).

Network Join

We implemented a function join() to handle the three-way handshake described in Section 2.1.1 Network Formation (implementation flowchart in Appendix D). In our algorithm, we keep a record of the current best parent candidate and update the record only if a better candidate node is discovered. The metric is based on the link RSSI (received signal strength indication) and hops to the gateway according to the following algorithm. RSSI value greater than -70 is considered to be a reliable link quality [12].

```
# After receiving a join acknowledgement from new_candidate
if best_candidate is not empty:

    # The new candidate is better only if it has an acceptable RSSI over -70 and
    # a closer distance to the gateway. If it is indeed better than the current
    # best candidate, update the current best candidate

    if new_candidate.RSSI >= -70 and new_candidate.hops < best_candidate.hops:
        best_candidate = new_candidate
else:

    # This is the first parent candidate available, so we always record it.

    best_candidate = new_candidate
```

Algorithm 1. Algorithm for a new node to update the current best parent candidate.

A special case occurs when the node has only one parent candidate with poor signal strength (less than -70) to connect to. With our metrics, the node will refuse to connect to such a parent candidate. From a short-term perspective, our metric seems too strict and can jeopardize the connectivity at the beginning. However, by placing some guarantees on the reliability of the link, it can prevent possible link failures in the future.

Event Loop

After a node successfully joins the network using join(), it starts the core event loop in the function run(), where data gathering (Section 2.1.2) and fault handling (Section 2.1.3) are implemented.

Table 10. A description of the event loop in run(). For better clarity, we assume that it is running on Node A.

Message Received	Description	Action taken by Node A
Join	A new node is looking for a parent node to join the network.	Send back a JoinACK if it has the capability for an additional child.
JoinCFM	A new node (as a child node) has established a connection with Node A (as a parent node).	Record the sender node as a child node.
GatewayReq	The gateway requires all nodes to send the data they collected. Node A's parent forward the request to Node A.	Forward the gateway requests to Node A's children. Then use a callback function to read the data and send it to Node A's parent.
NodeReply	Node A's child sends the data it collected to Node A.	Forward the message to its parent.
CheckAlive	A child node wants to confirm the status of Node A.	Send back a ReplyAlive message indicating that the link is up.
ReplyAlive	The parent node of Node A replies that it is not down.	Mark the parent node as alive and restarts the timer for aliveness checking.

In the event loop, we use a random backoff timer to avoid transmission collisions. Before a node sends a message, it will first wait for a random amount of time from 0.1s to 3s before sending the message. The random seed uses the 2-byte long node address which is unique to every node. Therefore, every node has different random sequences of wait times. The lower bound of 0.1s is chosen based on the approximate time measured for a node to complete one transmission. The upper bound is set to 3s such that the latency introduced will be less than 3 seconds.

Without the random backoff timer, transmission collisions occur when two or more nodes simultaneously send a message. For example, if two nodes receive a Join message from a new node at the same time and reply back simultaneously, one or both replies can be lost due to a transmission collision.

2.3.4 API FrontEnd (author: Yizhi Xu)

The API FrontEnd is a wrapper layer that provides the interface of the library to users. It allows users of the library to send and handle messages received by their nodes.

Table 11. Major functions in the API Frontend

API	Required for	Description	Usage
onReceiveRequest(function)	Regular nodes	Set a callback function to handle requests received from the gateway.	Users can define the behavior of a node when it receives requests from a gateway. The data generated or collected by this function will be sent back to the gateway.
onReceiveResponse(function)	Gateway	Set a callback function to handle responses received from nodes.	Users can define the behavior of the gateway, when the gateway receives the responses back from the nodes upon its request.
setGatewayReqTime(time)	Gateway	The interval for gateway to send requests to nodes.	Users can control the period of time for the gateway to send out requests.

These APIs abstract the corresponding functions in the Forwarding Engine, and they allow users to create arbitrary applications based on our library.