LoRa Synchronized Energy-Efficient LPWAN (SEEL)

Boyang Zhang, Yan Zhang, and Robert Dick

University of Michigan, S State St, Ann Arbor, Michigan, 48109

1 Purpose

The LoRa Synchronized Energy-Efficient LPWAN (SEEL) communication protocol is a novel Low Power Wide Area Network (LPWAN) utilizing LoRa-PHY technology. SEEL differs from LoRa-WAN and other existing work due to its implementation of an energy-efficient multi-hop tree topology. SEEL is recommended for low data rate applications that require an extensive area to be covered using battery-powered nodes; an example of this is agricultural data collection which typically requires monitoring of large areas without power distribution infrastructure.

This document explains the design and implementation of the SEEL protocol, the setup (both hardware and software) required to start a new network, suggestions on using the network in practice, and areas for future improvement. The code for SEEL is available at https://github.com/SEEL-Group/SEEL.

2 Keywords

- NODE: A physical network device containing a LoRa transceiver.
- GNODE: Gateway node, also called the root node. This device serves as the sink of messages in the SEEL network and is not energy constrained. The GNODE transfers collected messages in the network to a usable form (uploading to a server, USB logging, etc). By default, the GNODE has the ID of '0'. The entire network must have exactly one GNODE to function correctly. A GNODE is defined by its functionality, so different types of devices can serve as GNODEs with the proper code and hardware.
- SNODE: Sensor node. This device acts as the message source and forwarding component of the network, and it is typically energy constrained. The network requires no SNODEs to operate; however, it requires at least one to be useful. An SNODE is defined by its functionality, so different types of devices can serve as SNODEs with the proper code and hardware.
- Cycle: Time between GNODE broadcasts. The time of a cycle is equal to the time SNODEs are awake plus the time they are asleep. Cycle times can be dynamically adjusted through GNODE broadcasts.
- Hop: NODEs are 1-hop away from each other if they are within range to communicate with each other without relying on other NODEs; however, they may not necessarily communicate with each other. NODEs are 2-hops away from each other if their communication relies on exactly one in-between (forwarding) NODE. Thus, two NODEs are N-hops away when they require N-1 NODEs to relay messages between them.
- Parent/child: If two linked NODEs are 1-hop away from each other, the upstream (going towards the GNODE) NODE is the parent and the downstream (going further from the GNODE) NODE is the child. A NODE can have multiple children but only one parent. Parent-child relationships are dynamically configured and thus can change throughout time.
- Downstream: Transmissions traveling down the tree, away from the GNODE.
- Upstream: Transmissions traveling up the tree, toward the GNODE.

3 Protocol Description

3.1 Packet Contents

The default message packet size is 14 bytes (not including LoRa headers); however, the user can add more bytes to the USER packet field if needed. The size of the packet is statically determined at compile time, and the full packet is sent every time. The user can increase the default message packet size by increasing "SEEL_MSG_USER_SIZE" as described in Section 5.3.2. Each packet field is described below along with how many bytes they each have allocated in the message packet.

Packet

TARG	SEND	CMD	SEQ	MISC

TARG: 1 Byte. Target. ID of the NODE this packet is addressed to.

SEND: 1 Byte. Sender. ID of the most recent sender of this packet, updated before transmissions. Original sender ID can be placed in data messages, if needed.

CMD: 1 Byte. Command. Different commands change how the packet is interpreted and the MISC field contents.

0. SEEL_CMD_BCAST – Broadcast (BCAST) packet. This message is initially transmitted by the GNODE and is echoed by SNODEs that receive the message, which creates a cascading effect that continues until all the SNODEs that are in range of another NODE have joined the network. Broadcast packets have no target; thus any SNODE waiting for a broadcast will check the packet contents of all the broadcast messages it receives. SNODEs are activated (see Section 3.5 for more details) upon receiving the broadcast message, so they can start sending messages in the current cycle. Broadcast messages contain network information (such as SNODE awake time, SNODE sleep time, etc) and SNODE network entry responses, and they only travel downstream.

1. SEEL_CMD_ACK – Acknowledgement (ACK) packet. This message type can be sent by both the GNODE and SNODEs; however, only SNODEs will be the recipients of ACK messages. Acknowledgement packets have no target, instead the acknowledge target ID is embedded inside the message; this allows multiple SNODEs to be acknowledged efficiently, see Section 3.3.1 for more details. After a NODE receives a DATA or ID_CHECK message, it sends out an ACK message, but there typically is a delay before the ACK message is sent to avoid message collision. ACK messages only travel downstream.

2. SEEL_CMD_DATA – Data (DATA) packet. This message is sent by SNODEs towards the GNODE. Data messages contain information set by the user and can only travel upstream.

3. SEEL_CMD_ID_CHECK – ID check (ID_CHECK) packet. This message is sent by new (unapproved) SNODEs trying to enter the network; ID check messages contain entry responses generated by the GNODE (see Section 3.4 for more details) and only travel upstream.

SEQ: 1 Byte. Sequence number. A counter used to differentiate transmissions, since message content can be identical among transmissions. Sequence number is incremented with non-forwarding packet transmissions. This value may overflow, which results in the same sequence number being seen more than once. Overflow creates issues with the duplicate message buffer (Section 3.3.3) and the SNODE blacklist systems (Section 3.5.4) by treating new messages as old messages. More bytes can be allocated to this variable to reduce the probability of overflow.

MISC: 10 Bytes. Miscellaneous. Content changes based on the packet's command field, which also changes the number of allocated bytes to sub-fields in the MISC packet field.

MISC: Broadcast

TIME_SYNC	SLP_TIME	AWK_TIME	ID_FEEDBACK

TIME_SYNC: 4 Bytes. Contains the sender NODE's time right before it sent out this BCAST message. This value is used to synchronize the child SNODE's time with its parent NODE's time. SNODEs further down the network are less synchronized with the original GNODE time as each hop introduces latency to transmissions which delays the synchronization. Time synchronization is used by the TDMA protocol to schedule message transmission without collisions and can be used by the user to schedule device events.

SLP_TIME: 2 Bytes. Specifies how long the SNODE should stay asleep for every cycle in seconds. The specified time may be different in practice depending on the hardware clock precision (see Section 6.6).

AWK_TIME: 2 Bytes. Specifies how long the SNODE should stay awake for every cycle in seconds. This value is the theoretical time the SNODE will be awake for; the duration of the AWK_TIME starts after the SNODE processes the broadcast message. However, the SNODE will have been awake for a while waiting for the broadcast message. Thus, total awake time will be the time spent waiting for the broadcast message plus the specified AWK_TIME. The time spent waiting for the broadcast message may not be precise due to SLP_TIME not being precise (see Section 6.6 for more details).

ID_FEEDBACK: 2 Bytes. Default space allocated for SNODE entry; more space can be allocated in the USER field. Two bytes are needed for one SNODE to join: 1 byte for target SNODE ID and 1 byte for response. By default, only one SNODE can join the network at a time per BCAST. There are three different types of responses (generated by the GNODE upon receiving an ID_CHECK message):

- ID_ACCEPT (ID: x, RESPONSE: x): SNODE with ID x is accepted into the network, no change needed. This occurs when no other NODE has taken the ID x.
- ID_CHANGE (ID: x, RESPONSE: y): SNODE with ID x is changed to ID y and accepted into the network. This occurs when an existing NODE in the network has the ID x.
- ID_ERROR (ID: x, RESPONSE: 0): SNODE with ID x has a conflict with another pending SNODE's ID; both SNODEs will randomly re-generate their ID and send an ID_CHECK message to the GNODE again. This occurs when more than one unregistered SNODE with the same ID tries to join the network; this does not have to occur in the same cycle since the GNODE has a queue for pending network join requests and the broadcast message may not contain all requests due to limited ID_FEEDBACK space. The ID cannot be changed to one that is available in the network like it is in ID_CHANGE since both the requests would be then accepted so there would be two or more SNODEs in the network with the same ID; thus, the ID is flagged and all join-pending SNODEs with that ID re-generates their ID and re-tries the join process.

MISC: Acknowledge

ACK_ID

ACK_ID: 10 Bytes. Contains SNODE IDs that required acknowledgement. SNODEs looking for acknowledgement will check every byte of this sub-field to check if they were acknowledged. Contents are defaulted to 0, which is not a possible SNODE ID. Additional space can be allocated by increasing "SEEL_MSG_USER_SIZE".

MISC: Data

INFO

INFO: 10 Bytes. User specified information (Set in "SEEL_sensor_node.ino"). Additional space can be allocated by increasing "SEEL_MSG_USER_SIZE".

MISC: ID_Check

GEN_ID	UNIQUE_KEY	X

GEN_ID: 1 Byte. Current tentative SNODE ID. Generated on SNODE start-up and re-generated if there is an ID conflict.

UNIQUE_KEY: 4 Bytes. Unique key generated on start-up that decreases chances of indiscernible SNODE network entry requests. Without this key, the GNODE does not know whether an ID_Check message is from an SNODE with a pending request or from a new SNODE since duplicate IDs can occur. Even with the unique key, an indiscernible situation can still occur if two or more SNODES randomly generate the same tentative ID and unique key; a total of 5 randomly generated bytes need to match. However, due Arduino's poor random number generator (see Section 6.7), this situation may occur more often than expected.

X: 5 Bytes. Ignored.

3.2 Scheduler

The SEEL scheduler ("SEEL_Scheduler.h") serves to provide the user an interface for their custom tasks to integrate with system behavior. The main loop for the SEEL protocol is in the scheduler, so both user and system tasks are run directly from the scheduler.

To prevent user tasks from taking priority over system tasks, system tasks can enable and disable user tasks through functions given by the scheduler. By default, user tasks are disabled on the SNODE until the SEEL system setup is completed. The SNODE disables user tasks on wake up and enables them when the broadcast message is processed and the SNODE has been verified in the network (when system tasks are completed for the cycle).

SEEL tasks ("SEEL_Task.h") are added to the scheduler using public methods in the scheduler. Only one-shot tasks are allowed to be added to the scheduler, so repeat tasks need to re-add themselves in their own function after they complete and before control is returned to the scheduler. Tasks can be added with a delay to allow scheduling for future events. It is important that user tasks are designated as "user" to give priority to system tasks; tasks are by default marked as user tasks. Additionally, the user functions added to the scheduler should not block and should execute quickly; otherwise, control is not returned to the scheduler and some time sensitive tasks (such as SNODE wake up) can be delayed.

To create a SEEL task, either pass a function pointer to the scheduler (see "SEEL_sensor_node.ino" and "SEEL_gateway_node.ino" for examples) or inherit the "SEEL_Task.h" class to create a child task object which can then also be passed to the scheduler (see "SEEL_GNode.h" or "SEEL_SNode.h" and their respective *.cpp files for examples). User tasks can get the current task information by calling the "get_task_info" method in "SEEL_Scheduler.cpp" which may help in scheduling tasks and debugging.

3.3 Message Transmission

Both the GNODE and SNODE share code for message sending, which is located in "SEEL_Node.h". Message transmission for NODEs aims to successfully deliver packets while reducing collisions due to multiple NODEs transmitting at the same time. While NODEs send different types of messages, every NODE is capable of sending out acknowledgement (ACK) messages to let other NODEs know their packets were successfully sent. ACK messages are prioritized over non-ACK messages in transmissions to ensure older messages go through the network before newer ones do.

3.3.1 Acknowledgement Messages

Messages with targets (DATA, ID_CHECK, and forwarded messages) require the sender NODE to wait for an acknowledgement (ACK) message before transmitting another ACK-required message. This allows the sender NODE to keep the message until it knows the message has been successfully delivered. If an ACK message was never received (due to interference, SNODEs sleeping, etc) by the sender NODE after a time-out period, the original message is re-transmitted. The ACK packet has no target, so all NODEs in range of the sender NODE sees the ACK packet and checks if the packet has their ID in it. SNODEs that are not waiting for an acknowledgement ignore the ACK message.

NODEs have an acknowledgement ID buffer (which is reset on cycle start) that lists the IDs of NODEs to acknowledge. After a NODE receives a message that requires an ACK response, the NODE records the ID

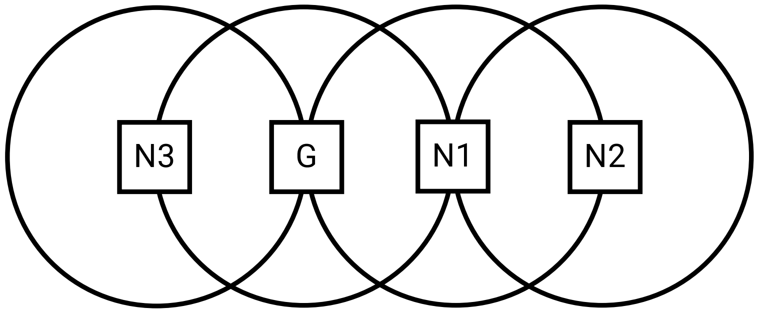

Figure 1: An example network topology. Here, the GNODE (G) is in range of SNODEs 1 and 3 (N1 and N3) while SNODE 2 (N2) must hop through SNODE 1 to communicate with the GNODE. The rings are simplified visualizations of each NODEs' transmission ranges.

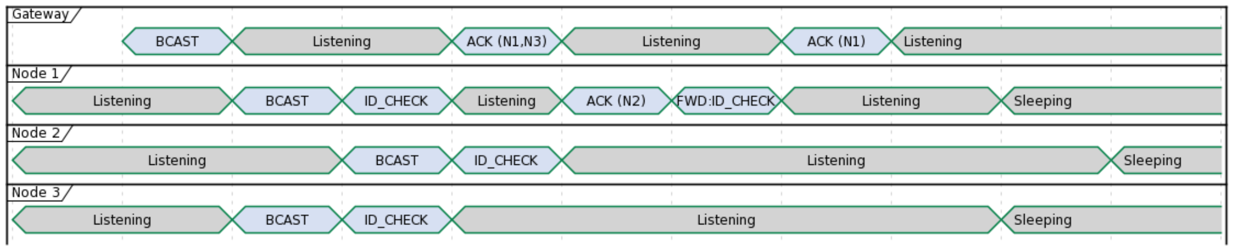

Figure 2: An example network join timing diagram with the topology depicted in Figure 1. Intervals (gaps between dotted lines) represent the time it takes between a NODE sending a message and another NODE receiving the message, the Time on Air, of a message transmission (t). See Section 6.4 for configuring t. Network variables (collisions, interference, variable transmission times, etc) are ignored for simplification. SNODEs are awake for $6t$ in this example. Listening and Sleeping represent two states of the NODE. Listening is the state when the NODE is waiting to receive messages from other NODEs. Note that listening times are dependent on system events; the example shown is an instance of what a cycle could look like. Sleep is the low-power state of the SNODE when it waits for the next cycle (note that the GNODE does not go to sleep). BCAST, ACK, ID_CHECK are transmissions corresponding to commands listed in Section 3.1. FWD:X represents the SNODE forwarding a message of type X.

of the sender NODE and adds the ID to its ID buffer. When the receiver NODE can send a message (in accordance to its collision-avoidance strategy) the NODE prioritizes transmitting ACK messages and sends as many stored IDs as it can fit in a message transmission: one ID takes up 1 byte of space in the packet. Those IDs are then immediately cleared; note this means if the ACK message fails to reach its target, then the ACK message is lost. Losing the ACK message results in the original sender NODE re-transmitting the original sent message since their ACK wait times out. Collecting multiple IDs instead of immediately sending single-target ACK messages prevents flooding the system with ACK messages; the receiver NODE must wait on their collision-avoidance send slot regardless, so no extra response latency is introduced by the collection process.

3.3.2 Collision Avoidance

There are two collision avoidance modes implemented by the SEEL protocol: Time-Division Multiple Access (TDMA) and Exponential Back-off (EB). Correctly configured TDMA ensures collision reduction, but it requires users to manually assign slots to NODEs and may extend the time a NODE takes to send out a message by forcing it to wait for its scheduled TDMA slot. On the other hand, EB doesn't require manual assignment, but its stochastic natures leads to unpredictable results. TDMA is more effective than EB as transmission times get longer (over 100 milliseconds) as the EB wait interval to ensure few collisions becomes too long to be practical. LoRaWAN airtime calculators [1] give a good estimate for transmission times, so one

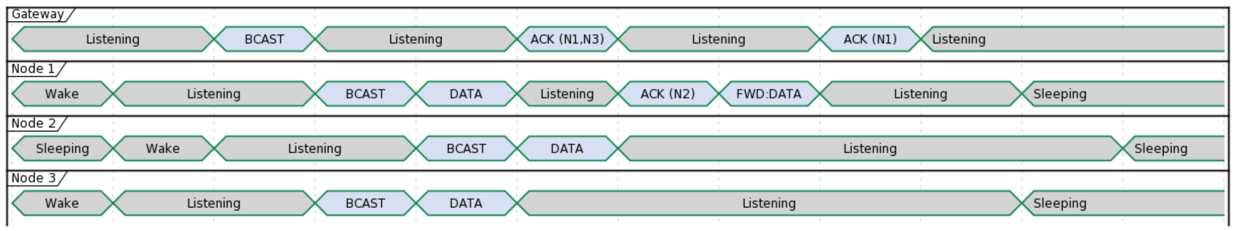

Figure 3: An example data transmission timing diagram with the topology depicted in Figure 1, with similar network assumptions as Figure 2. Note the timing diagram looks very similar to the one in Figure 2 but ID_CHECK messages are replaced with DATA messages and SNODEs wake up at the beginning of the cycle to receive the BCAST message. This example would have multiple collisions in practice since no collision avoidance technique is used; see Figure 4 for a similar example with TDMA collision avoidance implemented.

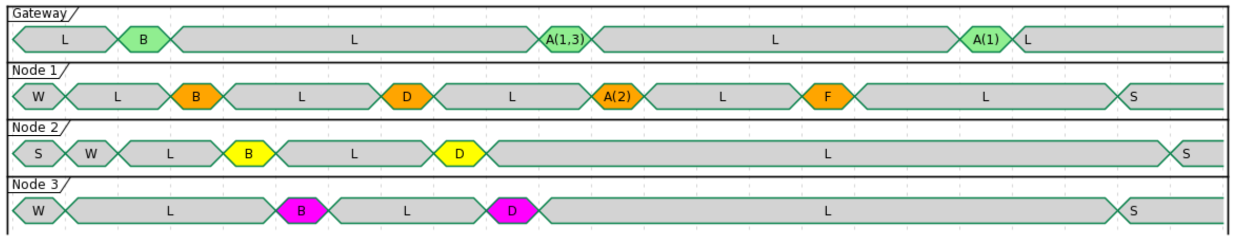

Figure 4: An example data transmission timing diagram (similar to Figure 3, with condensed notations) using TDMA. Note the duration that SNODEs need to stay awake for is increased from the previous scenario to ensure all messages are sent during the cycle.

can decide which collision avoidance approach is more practical. The variable "SEEL_TDMA_USE_TDMA" (which defaults to TDMA mode) in "SEEL_Params.h" allows users to toggle between TDMA and EB; also, the same header file contains additional parameters for both modes.

Time-Division Multiple Access (TDMA)

TDMA requires the user to specify a number of slots ("SEEL_TDMA_SLOTS"), provide a maximum transmission time for all NODEs ("SEEL_TDMA_TRANSMISSION_DURATION_MILLIS") and buffer time ("SEEL_TDMA_BUFFER_MILLIS"), and assign each NODE a slot ID. With these parameters, the strategy is to give each slot ID a slot time of the max transmission time plus the buffer time and NODEs would have to wait until their slot time to transmit; thus, no two NODEs with different slot IDs would be transmitting at the same time. A buffer time is needed to account for transmission duration variability and system-related computational delays. An illustration of data transmission using TDMA is shown in Figure 4.

The TDMA strategy places burden on the user to assign slot IDs. NODEs should be assigned TDMA slot IDs according to the following rules.

1. NODEs within transmission range of each other cannot have the same TDMA slot. If NODEs are in transmission range of each other, their messages can collide with each others'; thus, they need to have separate TMDA slots.

2. NODEs that communicate with a common NODE cannot have the same TDMA slot. For example, imagine a linear topology where A is B's parent and B is C's parent, but A and C are not in transmission range of each other. Then A and C cannot have the same TDMA (B also cannot have either A or C's TMDA slot due to Rule 1). If NODEs share a common NODE, there are instances where the NODE-in-common needs to listen to messages from both upstream and downstream (example, ack messages and data messages). When both upstream and downstream sources talk at once, the NODE-in-common can only listen to one of them, therefore losing one of the messages. Thus, the upstream

(parent) and downstream (children) NODEs of the NODE-in-common need to have different TDMA slots.

Exponential Back-off (EB)

The EB strategy is much simpler than TDMA. NODEs wait a bounded random period of time, lower-bounded by "SEEL_EB_MIN_MILLIS" and upper-bounded by "SEEL_EB_EXP_SCALE" raised to the power of the number of unacknowledged transmissions at the time multiplied by "SEEL_EB_INIT_MILLIS" before transmissions. Hence the name exponential back-off since NODEs wait exponentially longer before transmitting again. The unacknowledged transmissions count is set to zero after the NODE is acknowledged. EB requires no manual schedule configuration; however, increasing NODE density or reducing the random wait time bounds will lead to more, unpredictable collisions.

3.3.3 Duplicate Buffer

Multiple instances of the same message may be received by a NODE due to multi-path effects (signal reflections, deflections, scattering, etc). Were it not for sequence numbers, messages intended as distinct might be identical. The duplicate buffer contains a queue (the amount held determined by "SEEL_DUP_MSG_SIZE") of previous received messages. If a new received message has the same sender ID, command, and sequence number of a recorded transmission in the buffer, it is marked as a duplicate and ignored.

3.4 GNODE Behavior

The GNODE acts as the root node of the network where all upstream messages end up. There must be one and only one GNODE in the SEEL network for proper functionality. ID '0' is exclusively reserved for the gateway node; using ID '0' for SNODE circumstances (such as ID entry responses) signals an error.

At the beginning of every SEEL cycle, the GNODE sends out a broadcast message, which is repeated periodically (where the period of the broadcast is controlled by the parameter "SEEL_BCAST_PERIOD_MILLIS"). The broadcast contents are described in Section 3.1 under "MISC Broadcast". This message is received by the SNODEs 1-hop away from the GNODE, and those SNODEs will set the GNODE as their parent, so they know who to send upstream messages to during the current cycle; however, the GNODE, by default, will not keep track of these child SNODEs. After processing the broadcast message, the SNODEs send out a modified version of the broadcast message to SNODEs without parents 1-hop away. This is process is elaborated in Section 3.5.

There are two possible message types the GNODE can receive: ID_CHECK and DATA messages, differentiated by the message command. Both of these message types require the GNODE to acknowledge the immediate sender SNODE. DATA messages are exposed to the user to handle (upload to server, USB log, etc). ID_CHECK messages are to verify an SNODE's entry into the network, which is tracked by the GNODE. The GNODE checks its internal array tracking all possible network IDs. If the requested ID is free or taken and expired, the GNODE okays the ID; if the ID is taken but not expired, the GNODE finds an available ID to assign to the requesting SNODE (scanning its internal array from the last element to the first, as users are more likely to manually assign SNODEs with lower ID values). After an ID is accepted, its expiration timer is reset when the GNODE receives a DATA message from that ID. The GNODE also sends out acknowledgement messages to alert immediate child SNODEs that their messages were successfully received by the GNODE (see Section 3.3.1).

3.5 SNODE Behavior

Useful networks contain one or more SNODEs. Due to congestion and device memory constraints, the SEEL network can only practically handle a limited number of NODEs. The maximum number of NODEs in the network can be adjusted via the "SEEL_MAX_NODES" parameter. An SNODE in the SEEL network both sends data messages and forwards messages from other NODEs (BCAST messages from the GNODE and DATA/ID_CHECK messages from other SNODEs). SNODEs start out in the join phase (Section 3.5.2) and transition to the data phase (Section 3.5.3) once they are accepted into the network.

SNODEs can be placed anywhere within 1-hop range of a NODE already in the SEEL network; they are battery powered which means they are not limited by power infrastructure. To maximize NODE-to-NODE communication range, one should pay attention to antenna orientation, surrounding obstacles, and interfering devices (such as other LoRa networks). The message throughput of an SNODE during a cycle mainly depends on the SNODE's awake time and the number of messages the SNODE needs to send in a cycle, which depends on the number messages the SNODE itself generates plus the number of forwarding messages from nearby NODEs. One should be cognizant of SNODE arrangements to prevent bottlenecks. The tree network structure increases the communication load of SNODEs closest (in hops) to the GNODE.

3.5.1 General Behavior

SNODEs complete the following general tasks regardless of the phase they are in:

1. Wait for a BCAST message. Once received, the SNODE records the original sender of the BCAST message and sets its own ID as the new sender ID in the packet field. Then the SNODE re-transmits the modified BCAST message when possible according to its collision avoidance scheme.
2. Synchronize immediately-used network parameters contained in the BCAST message: the corrected system time, how long the SNODE should be awake for "SEEL_SNODE_AWAKE_TIME_SECONDS", and how long the SNODE should sleep for "SEEL_SNODE_SLEEP_TIME_SECONDS". Additionally, the SNODE sets the sender of the received broadcast message (the sender ID is contained in the BCAST message) to be its own parent.
3. Set a sleep task to occur after the SNODE has been awake for the specified time. Note this means the awake time only applies to the time the SNODE is awake after receiving the broadcast message.
4. Send out phase specific messages.
5. Forward received messages designated for its ID. The parent NODE does not have a list of its child SNODEs; thus, the SNODE forwards all messages targeted towards itself, assuming they are from child SNODEs.
6. Sleep in the low-powered state for the time specified by "SEEL_SNODE_SLEEP_TIME_SECONDS" when the sleep task occurs. Note that the SNODE should wake up before the next BCAST message arrives in order to receive it, this should be configured using network parameters, as described in Section 6.6.

3.5.2 Join Phase

SNODEs start off in the join phase; they are constantly in a high-powered state waiting for a broadcast message. In addition to the general tasks, an SNODE in the join phase checks its ID with the GNODE by sending out an ID_CHECK message with its current ID. Figure 2 showcases an example network where SNODEs are in the join phase.

The SNODE receives a response from the GNODE in a BCAST message regarding its join request. If the GNODE accepted the SNODEs ID request, the SNODE keeps its current ID and moves on to the data phase; if the GNODE suggests an ID for the SNODE, the SNODE takes the suggested ID and moves on to the data phase. However, if the GNODE flags the requested ID as an error, then the SNODE re-generates an ID and restarts the join phase.

3.5.3 Data Phase

SNODEs reach the data phase after joining the network and receiving confirmation of that through a BCAST message. SNODEs in the data phase stay in the data phase until a system reset (power cycle, new code upload, etc). In the data phase, SNODEs complete the general tasks and also send data messages generated by the user. Figure 3 showcases an example network where SNODEs are in the data phase.

3.5.4 Blacklisting Parents

Child SNODEs may not receive responses from their parents during a cycle; this issue occurs if the parent NODE is too busy to respond (overloaded NODE) or the bidirectional communication between parent and child NODEs is either asymmetrical (the child can receive messages from the parent but the parent cannot receive messages from the child) or unstable. Thus, the parent never receives messages from the child and the child's messages are lost; future messages are also likely to be lost if the same parent-child link were to be established again. However, since there may be additional NODEs around, a better connection may be available.

To solve this problem, SEEL uses a blacklist system on SNODEs. If the parent of an SNODE does not respond during a cycle, the SNODE blacklists the parent's ID and will not set that ID to be the parent (however it still receives the network parameters from the broadcast) thus enabling other nearby NODEs to become the parent. However, if the SNODE receives no available parents while it is awake, then the next cycle it clears the blacklist, trying the previous parent-child links again. Although the SNODE will not set its parent to be a NODE in its blacklist; if the SNODE receives a broadcast message from a blacklisted parent, the SNODE takes the system synchronization parameters (awake time, sleep time, etc) from the broadcast message to stay synchronized with the network and sleep (otherwise the SNODE may stay awake the entire time looking for a non-blacklisted parent).

4 Hardware Setup

The SEEL protocol is optimized for LoRa transceivers, but is otherwise hardware independent. The hardware components in our implementation were selected based on popularity and ease of use. Switching hardware platforms (from Arduino) would require significant code modification.

4.1 GNODE Setup

SEEL is tested with the Dragino LG01-P (customized Arduino Yún) as the GNODE. The LG01-P includes a LoRa transceiver, wireless card, and an USB interface which are useful for SEEL. Other Arduino devices can readily function as GNODEs; a device needs at minimum a LoRa transceiver.

An off-the-shelf Dragino LG01-P can directly be deployed to and used as a GNODE. Consult [2] for general Dragino LG01 setup instructions. See Appendix A for Dragino LG01 logging setup instructions.

4.2 SNODE Setup

SNODEs require LoRa transceivers. Our SNODE implementation consists of an Arduino Pro Mini 328 – 3.3 V/8 MHz, an RFM95W-915S2 LoRa transceiver chip [3], and a CR123 battery holder. Both the battery holder and LoRa transceiver are directly soldered to the Arduino Pro Mini.

The Arduino Pro Mini serves as the microcontroller platform to control the SNODE. Arduino is a popular choice for embedded projects since it is beginner friendly and has many libraries for different platforms; additionally, the Dragino LG01 has an underlying Arduino Yún. Originally, the SEEL protocol utilized the Arduino Uno; however, the Arduino Uno has high sleep power consumption, so the board was changed to the Arduino Pro Mini, largely motivated by [4]. The base Arduino Pro Mini is modified by desoldering the power jumper pad next to the GND and RST pins to the deactivate the power LED and voltage regulator to achieve ultra low power consumption (some clone Arduino Pro Mini's may not contain this jumper).

The LoRa transceiver is required to send and receive LoRa packets. There are RFM95 breakout boards sold that make soldering easier; however, they add power LED's and voltage regulators which drives up the power consumption of the entire SNODE system and increase transceiver price. LoRa transceivers have different frequency bands; 915 MHz was appropriate for our tests in North America. Our LoRa transceivers are outfitted with whip antennas that are cut to fit its 1/4 wavelength (7.8 cm). Additionally, 10 µF ceramic capacitors are soldered to the transceiver's power supply pins to act as decoupling capacitors as instructed by the RFM95W specifications found at [3].

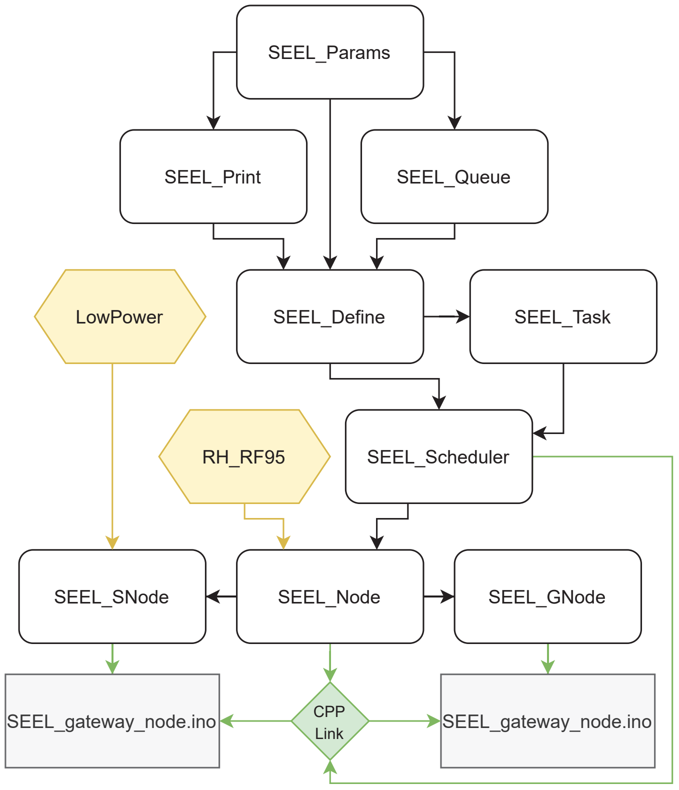

Figure 5: Header File Hierarchy.

The CR123 battery holder serves to enable CR123 batteries to power the SNODE system. Other sources of power can be used, as long as they fall within the chosen components' supply power constraints. CR123 batteries were selected because of their battery voltage characteristics, long shelf-life, and low price.

5 Software Setup

5.1 External Libraries

FileIO.h: Manages files on Arduino. Default Arduino library.

Console.h: Enables communication between Arduino Yún and the computer. Default Arduino library.

LowPower.h: Places the Arduino into low power state by disabling board functionalities. The repository can be found at [5].

RH_RF95.h: Radiohead LoRa RFM95 transceiver library. The repository can be found at [6].

5.2 Header File Hierarchy

See Figure 5 for a visual depiction of the SEEL header file hierarchy. Keeping the SEEL files linked locally instead of through the Arduino library system allows for fast development. To link them locally, the SEEL *.cpp files are included at the beginning of the Arduino (*.ino) files. This is denoted in Figure 5 with green lines. Otherwise, the SEEL header files are placed in the Arduino library folder.

5.3 Parameters

5.3.1 Frequently Used Parameters

Frequently used parameters for both the GNODE and SNODEs are placed in their respective *.ino files to allow for easy access and fast tuning. Most SNODE parameters apply to all SNODEs that are uploaded with the settings; however, SNODE settings broadcast by the GNODE (like SNODE awake times) apply to all SNODEs in the network.

GNODE ("SEEL_gateway_node.ino")

- SEEL_BCAST_PERIOD_MILLIS: Duration of a cycle in the SEEL network. The GNODE sends a broadcast message every SEEL_BCAST_PERIOD_MILLIS milliseconds after initialization.
- SEEL_SNODE_AWAKE_TIME_SECONDS: Duration SNODEs in the network should be awake for. This parameter (and the sleep time parameter below) is embedded in the broadcast message and is checked by SNODEs every cycle start, so SNODE sleep/wake timers can be dynamically adjusted even after SNODEs have joined the network. Actual awake time may be different due to the SEEL scheduler blocking in tasks when it's time to sleep. Units are in seconds (same with the sleep time parameter below) because of limited space in the broadcast packet.
- SEEL_SNODE_SLEEP_TIME_SECONDS: Duration SNODEs in the network should be asleep for. This should be a multiple of eight seconds since the Arduino Watchdog setting used is eight seconds; if more precise times are required, see Section 7.9. In practice, the SNODE will typically be asleep for longer than the specified duration (See Section 6.6).

SNODE ("SEEL_sensor_node.ino")

- SEEL_SNODE_ID: Manual ID assignment for the SNODE ID. Use the value of '0' to randomly generate an ID.
- SEEL_RFM95_CS: CS pin for the LoRa device. Passed to the RFM95 initialization function.
- SEEL_RFM95_INT: INT pin for the LoRa device. Passed to the RFM95 initialization function.
- SEEL_RFM95_RST: RST pin for the LoRa device. The RST pin is triggered in the SEEL initialization code to perform a reset on the LoRa device.

5.3.2 SEEL Library Parameters

- SEEL_PRINT_USE_CONSOLE: Whether to use the Console (true) or Serial (false) for SEEL debugging messages.
- SEEL_PRINT_DEBUG: Whether SEEL debugging messages should be printed.
- SEEL_SERIAL_BAUD_RATE: Serial initialization baud rate.
- SEEL_BRIDGE_BAUD_RATE: Console initialization baud rate.
- SEEL_RFM95_FREQ: LoRa parameter, frequency specified in MHz. Applies to both the GNODE and SNODE. See Section 6.4 for more information on LoRa parameter selection.
- SEEL_RFM95_SF: LoRa parameter, spreading factor. Ranges from 6 to 12. Applies to both the GNODE and SNODE.
- SEEL_RFM95_BW: LoRa parameter, Bandwidth specified in Hz. Possible values are: 7800, 10400, 15600, 20800, 31250, 41700, 62500, 125000, 250000. Applies to both the GNODE and SNODE.
- SEEL_RFM95_GNODE_TX: LoRa parameter, GNODE transmission power. Ranges from 5 (min) to 23 (max) specified by the RFM95 library.
- SEEL_RFM95_GNODE_CR: LoRa parameter, GNODE coding rate. Ranges from 5 to 8 which represents the denominator of the coding rate specified by the RFM95 library. Thus, setting this parameter to 5 would represent a coding rate of 4/5.

- SEEL_RFM95_SNODE_TX: LoRa parameter, SNODE transmission power. Ranges from 5 (min) to 23 (max) specified by the RFM95 library.
- SEEL_RFM95_SNODE_CR: LoRa parameter, SNODE coding rate. Ranges from 5 to 8 which represents the denominator of the coding rate specified by the RFM95 library. Thus, setting this parameter to 5 would represent a coding rate of 4/5.
- SEEL_MSG_USER_SIZE: Number of user allocated bytes in the message packet. Increasing this allows for more data to be sent, but increases transceiver transmission time as more bytes need to be sent.
- SEEL_DUP_MSG_SIZE: Number of messages to hold in the duplicate message buffer. Increasing this decreases the chances of allowing a duplicate message through but increases the memory usage and computational effort of the SNODEs.
- SEEL_MAX_NODES: Max number of NODEs allowed to join the network. This parameter takes a significant toll on memory usage in the SNODE; the Arduino platform does not warn on most out-of-memory issues. If cryptic errors occur, first try reducing the memory usage of the device (for example, by lowering this parameter).
- SEEL_MAX_CYCLE_MISSES: Max number of cycles an SNODE can miss before getting kicked out of the network. This kick-out is only executed when a new SNODE tries to join the network and a slot is needed.
- SEEL_SEND_TIMEOUT_MILLIS: Transceiver time-out. The duration to wait for the transceiver to send out a packet before aborting and re-trying the transmission when possible.
- SEEL_ACK_TIMEOUT_MILLIS: Acknowledgement time-out. The duration to wait for an acknowledgement after sending out a acknowledgement-needed packet (any packet types besides BCAST and ACK) before re-transmitting the packet when possible.
- SEEL_WD_TIMER_SECS: The duration of a Watchdog sleep. Possible values are: SLEEP_8S, SLEEP_4S, SLEEP_2S, SLEEP_1S, SLEEP_500MS, SLEEP_250MS, SLEEP_120MS, SLEEP_60MS, SLEEP_30MS, SLEEP_15MS. Increasing this parameter lets the SNODE be more energy-efficient since it does not have to wake up as often but reduces the precision of the sleep duration.
- SEEL_PSEL_MODE: SNODE parent selection mode. Sets the criteria the SNODE (if the it received multiple broadcasts) uses to determine which NODE is set as its parent. Selection modes are defined in "SEEL_PARENT_SELECTION_MODE" in "SEEL_Params.h".
- SEEL_PSEL_DURATION_MILLIS: Duration to wait before selecting a parent, in order to find a potentially better parent. This value is a lower-bound on the wait duration; SNODEs will continue collecting possible parents until they are able to transmit their own broadcast packet. Thus, setting this value to 0 guarantees the SNODE will only collect possible parents until its broadcast message can be transmitted, and values greater than 0 forces the SNODE to wait on the specified duration and an available transmit time (based on the collision avoidance strategy) before transmitting its own broadcast message.
- SEEL_TDMA_USE_TDMA: Boolean toggle on whether to use the TDMA or the EB collision avoidance scheme. See Section 3.3.2 for more details on both schemes.
- SEEL_TDMA_SLOTS: The number of TDMA slots used by the TDMA algorithm. This value should be set to the smallest value possible (to reduce transmission latency) while having enough slots to follow the TDMA slot assignment strategy described in Section 3.3.2.
- SEEL_TDMA_TRANSMISSION_DURATION_MILLIS: Estimate of the transmission duration when NODEs transmit messages. This is typically the Time on Air of the transmitted message which can be estimated by measuring the time NODEs spend in their transmit state. Note if the NODE has variable-length transmissions, this value should cover the longest possible transmission duration.
- SEEL_TDMA_BUFFER_MILLIS: Buffer duration for transmissions. This value accounts for miscellaneous delays associated with transmitting messages such as transceiver delays, CPU delays, etc. The

State	Transmit (mA)	Listen (mA)	Sleep (µA)
Mean	126	15.1	5.41
Std Dev	2.65	0.147	0.709

Table 1: Mean current draw results from five different SNODEs running in the SEEL system with a controlled environment (no interference); the purpose of this test was to measure current on different SNODE devices. The test set up consisted of one SNODE and one GNODE. The SNODEs had a duty cycle of approximately 30%. Figure 6 shows the recorded current draw for one of the measured SNODEs.

total TDMA slot duration is equal to this value added to the previous estimated transmission duration parameter.

- SEEL_EB_INIT_MILLIS: Starting max duration of the transmission exponential back-off. Longer exponential back-offs reduce the chances of message collision but also increases the average delay duration before sending out a message.
- SEEL_EB_MIN_MILLIS: Constant minimum value of the exponential back-off algorithm.
- SEEL_EB_EXP_SCALE: Scaling factor for the exponential back-off algorithm.
- SEEL_RANDOM_SEED_PIN: Which Arduino pin is used for random seeding. Choose a pin that is not in use. See Section 6.7 on why Arduino's random number generator is problematic.
- SEEL_QUEUE_ALLOCATION_SIZE: How large the array size allocated for the SEEL_Queue class is. The SEEL_Queue class is used in all of the queues in SEEL (SNODE send_queue, SNODE blacklist, GNODE pending_broadcast_queue, Scheduler task_queue, etc). This means the array allocation must be large enough to satisfy the needs of all of the mentioned subsystems. Allocating too much unneeded space results in wasted memory usage. Static memory allocation is used since dynamic memory allocation is discouraged on low SRAM systems.

6 Notes

The following subsections contain useful setup and debugging information for deploying a SEEL network.

6.1 Network Performance

Data collection for a sample SEEL network is currently on-going. The results of the experiment will be added to this section.

6.2 Energy Measurements

We ran current draw measurements on five SNODEs with the hardware mentioned in Section 4.2 and with the following parameters: Centre Frequency = 915 MHz SF = 12, BW = 250 kHz, CR = 4/5, TX = 20 dBm, Payload size = 16 Bytes (plus 4 Bytes for explicit header). Results are shown in Figure 6 and Table 6.1. Based on the average sleep current of 5.41 µA and assuming a 1,500 mAh battery (typical mAh for CR123 batteries), we calculate the maximum lifetime (permanently staying in the sleep state) of the average SNODE to be $1,500\,\text{mAh}/5.41\,\mu\text{A} = 277,264$ hours, or 32 years. See Appendix B for more lifetime analysis.

6.3 Software Warnings

Currently the SEEL system has a warning of "Deletion of non-virtual pointer" stemming from a missing "virtual" keyword in the Radiohead library. This issue is unlikely to interfere with system functionality. The issue has been reported to the Radiohead library's developers.

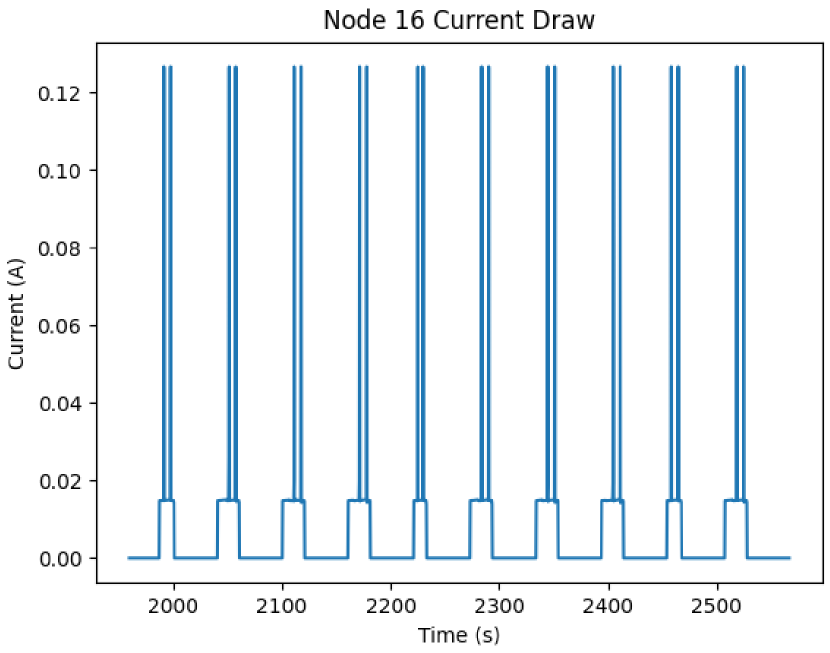

Figure 6: Example snapshot of current draw for an SNODE in the SEEL system. The SNODE is at receive mode when the current draw is around 15 mA; current spikes to ∼130 mA represent transmissions and current drops to ∼5 µA represent sleep mode. Note this scenario depicts the SNODE making two transmissions during each cycle: one for re-transmitting the broadcast message and one for ID_CHECK or DATA messages.

6.4 LoRa Parameter Selection

LoRa parameter selection is an important part of the SEEL system, as different combinations of LoRa parameters trade off node-to-node communication range and data rate, burst interference resistance, and power consumption. The key parameters are Transmission Power, Carrier Frequency, Spreading Factor, Bandwidth, and Coding Rate. Some other parameters are dependent on these key parameters; for example, the Time on Air is dependent on the data rate (SF, BW) and the number of bytes to be transmitted (which is dependent on the CR).

Transmission Power influences the output power on the LoRa transceiver. Higher output power increases the signal-to-noise ratio of transmissions but also increases the power consumption of the device. Transmission levels are hardware dependent and may have regional constraints.

Carrier Frequency sets the centre frequency of transmissions, programmable in steps of 61 Hz. The frequency range depends on the hardware used.

Spreading Factor (SF) and Bandwidth (BW) trades off range and data rate. Shorter range influences the network topology; the network may require more intermediate SNODEs to hop to SNODEs far from the GNODE. Slower data rates not only result in slower network traffic, but also increase message collision chance when using random send methods. When using TDMA, slower data rates result in longer TDMA slots, which increases message transmission latency that scales with the number of TDMA slots.

Coding Rate (CR) provides protection against burst interference. Increasing the coding rate adds more protection against decoding errors but results in larger packet sizes due to the extra error-checking bits.

6.5 Parameter Adjustment

Changing variable type sizes may require additional code modification. To accommodate the low SRAM capacity of the Arduino, variables in the SEEL code are often set to uint8_t or uint16_t. Thus, when changing parameter types, one must be careful of related variables in the code that interact with the target parameters. For example, many of the existing variable types related to NODE IDs are set to uint8_t; thus, if more NODEs are needed in the network, many variables types would need to be adjusted.

6.6 Awake/Sleep Time Discrepancies

Since the TDMA collision avoidance scheme relies on time synchronization among NODEs, NODE clock drift is an important aspect of the system. The Arduino Pro Mini Watchdog Timer (WDT) is imprecise; this problem is common in cheaper embedded systems. Per the spec sheets, the Arduino Watchdog Timer is adjustable in steps, with a maximum sleep time of 8 seconds. However, due to the low-end hardware components on-board (RC oscillator), the sleep times deviate up to 10% (according to online forums), and times are dependent on local temperature and supply voltage.

In our tests, we found our boards' Watchdog sleep times skew towards the higher end of the range, so when set to 8 seconds, the sleep times ranged from 8–9 seconds. Due to this variation in sleep time, we used an estimated upper-bound when calibrating sleep, awake, and cycle times for NODEs in the system.

For example, suppose we wanted an SNODE to be awake for 10 seconds and asleep for 24 seconds (since sleep times with the default WDT must be a multiple of 8 seconds). Assuming a single WDT sleep can range from 8-9 seconds (one would measure this in advance to get a better estimate), the SNODE would sleep for 24 to 27 seconds. Thus, instead of setting the cycle time to be 34 seconds, this SNODE would require a cycle time of at least 37 seconds to not miss broadcast messages to due WDT variation. Additionally, one would want to add in a small buffer to account for system response times and other factors.

Using a cautious upper-bound estimate ensures the SNODE wakes up and receives the broadcast message (which is critical otherwise the SNODE stays awake the whole cycle) but wastes energy if the upper-bound estimate is far from the actual wake-up time. One method to help calibrate the upper-bound estimate is to measure the time from when the SNODE wakes up to when it receives a broadcast message, termed the Wake-up To Broadcast (WTB) time. By measuring the distribution of the WTB, one can re-adjust the cycle time to a safe and more energy-efficient value. For example, sampling the minimum WTB and adjusting the cycle time by that value. The WTB is provided to the user in the load message callback function inside the struct “SEEL_Info”.

6.7 Random Number Generator

The Arduino Random Number Generator (RNG) fails to provide adequately random results; there were many empirical instances of non-uniform draws using the default RNG in our tests, even with the recommendation to random seed with an unused analog pin. This causes a problem in the SEEL network as randomness can be used to re-assign network IDs, and the more non-uniform the ID selection is, the more collisions would arise from assigning SNODE IDs from the RNG. This problem has also been noted by others in the Arduino online forums.

6.8 Limited Memory

The Arduino Pro Mini and Dragino Gateway (LoRa interface) has 2 KB of SRAM. Because of this limited memory, we had to reduce the number of possible SNODEs in the network since tracking more SNODEs requires more SRAM usage on the GNODE. Previously, debugging messages had to be limited since string constants occupy Flash and SRAM space which caused stack heap collisions since memory was already limited; however, the F() macro moved string literals to PROGMEM, which frees up SRAM. As an additional warning, avoid using dynamic memory where possible since there is the possibility of heap fragmentation. See [7] for additional tips on optimizing memory for Arduino.

7 Future Work

Although the SEEL protocol meets the basic criteria of an LPWAN, there are several aspects of the protocol that can be improved to make the network more efficient or to suit different types of real-world applications; these pending improvements are left as future work and are described in this section.

7.1 Additional GNODE Controlled Parameters

Mechanisms for re-adjusting network parameters grant control over the system even after deployment. SEEL currently allows for adjustment of SNODE awake and sleep times through data embedded in BCAST messages. Additional parameters, such as TDMA slot assignments, can be added as part of the network formation to increase post-deployment system control.

7.2 Improved Parent Selection

Currently, parent selection modes are limited to using local NODE information to determine the best suitable parents. Using accumulated path information, such as passing on the average or worst-case RSSI along a path, can lead SNODEs to make smarter choices on parent selection; however, this would require adding more bytes of information to the broadcast message.

7.3 Adaptive Data Rate

The present SEEL system only allows for compile-time data rate selection. Adjusting data rates for NODEs dynamically based on link condition gives greater optimization freedom to the network. However, existing TDMA scheduling would encounter problems because TDMA slots are assumed to be constant (upper bound on Time on Air for longest transmission); this assumption would be broken if data rates were to be adjusted dynamically. If the maximum Time on Air were to be used for TDMA, TDMA bottlenecks would prevent increasing message throughput with adjusting data rates. There may be more sophisticated algorithms with time synchronization that lead to personalized TDMA slots for local NODEs and neighbors to overcome this.

7.4 Variable-length Packet Transmission

Messages in the SEEL network currently have fixed length. Unspecified fields contain garbage. For example, by default, ID_CHECK messages require 5 bytes. One way to fix this inefficiency is to send variable-length packet sizes, based on the size of the number of bytes needed for the message. By reducing the number of bytes in the packet, the Time on Air duration of a message transmission decreases, allowing higher throughput of messages in the network. However, when using TDMA, the TDMA slot duration is tied to the longest possible message transmission duration (a network parameter), so using variable-length packets would not improve the network throughput when using the current TDMA algorithm; however, a more sophisticated TDMA implementation may resolve this.

7.5 Frequency Hopping and Fat Trees

The SEEL network currently only allows message transmission on the same channel across all NODEs. However, LoRa allows for orthogonal communication across different frequencies and spreading factors. Fat trees [8] are trees where SNODEs closer to the root of the tree (the GNODE) have more communication channels than SNODEs further from the root; this lessens the bottleneck issue where NODEs closer to the root handle more messages on average than NODEs further away. In addition to having additional communication channels, NODEs would have to frequency hop among these channels to detect messages pending on the channels. However, the downside is that NODEs, unless they have multiple transceivers, can still only receive one message at a time. Thus, without hardware upgrades where NODEs are equipped multiple transceivers, frequency hopping is limited to a collision avoidance technique.

7.6 Multiple GNODEs

The SEEL system only supports one GNODE as the sink node in the network. This can lead to problems if SEEL systems are used adjacent to each other or a GNODE is overloaded with messages. Having the ability to introduce multiple GNODEs into the SEEL system would allow load balancing and further scaling. The current issue is that SNODEs send their messages towards ID 0 which is designated as the GNODE ID, so using multiple GNODEs would require a set of designated IDs as sinks. Additionally, the network formation process would need to become more sophisticated to ensure even load balancing among GNODEs. Policy issues would come up regarding whether SNODEs should always send to the same GNODE or load balance dynamically. There is also the issue when different parties are using the SEEL system with different GNODEs; there needs to be a unique way GNODEs in the same network differentiate from GNODEs in other networks; for example, embedding a unique (on the network level) identifier in the packet header.

7.7 Event-Driven SNODE Behavior

NODEs uses a polling system to check if messages are available. This wastes energy that could be saved if the SNODE were awake only when necessary. Switching to an event-driven system that fired an interrupt on message reception would allow SNODEs to stay in sleep state (potentially a lighter sleep that can still receive LoRa messages) longer. An event-driven system would wake on message reception and sleep when there are no tasks to be run.

7.8 Server and Phone Application for Deployment

The deployment of the system currently requires knowledge of what the GNODE is logging; thus, current deployment efforts are much easier with at least two people: one to deploy the SNODEs and one to relay GNODE information. Deployment efforts would be greatly reduced if the GNODE were able to log information onto a server, so field agents could view the network traffic as they are deploying SNODEs.

7.9 More Accurate Sleep Time

For the our hardware implementation, sleep times are currently multiples of eight seconds: the longest single sleep duration supported by the Arduino's Watchdog timer. However, the Watchdog timer also supports shorter times (four seconds, two seconds, ect.). A more sophisticated sleep system could use a greedy approach to assign sleep counts per Watchdog timer duration to allow the user to specify more accurate sleep times. For example, if the user designated a 50 second sleep time, the system would sleep 6 times for 8-seconds, and 1 time for 2-seconds.

7.10 Custom LoRa-PHY Library

For our hardware implementation, we currently uses the Dragino Radiohead library to interface with our LoRa transceiver. However, this library forces the use of explicit header mode on LoRa which sends four bytes of header data that is ignored by the SEEL protocol since SEEL sends this info in the data payload. Implementing a new transceiver interface or modifying the existing Radiohead library would allow for greater flexibility and efficiency in LoRa message transmissions. Another solution would be to change the SEEL protocol to use the explicit header data instead of sending its own in the data payload.

8 Acknowledgements

We thank Brian Oo, Pranav Karra, Kevin Tactac for their work in assisting the SEEL protocol development and helping test SEEL deployments. We thank Brian Oo and Pranav Karra for their work in revising this document. We thank Deepen Solanki and Albert Anwar for their assistance during the early-stage development of this protocol.

Please email zboyang@umich.edu or dickrp@umich.edu for clarifications or questions regarding the SEEL protocol.

9 Version

Version	Date	Author	Change Log
1.0	06/2020	Zboyang	Documentation started
1.1	02/2021	Zboyang	Initial draft completed
1.2	04/2021	Zboyang	Documentation revised and published

References

[1] *LoRaWAN Airtime Calculator*. 2021 [Online]. URL: https://www.thethingsnetwork.org/airtime-calculator.

[2] *Dragino LG01 LoRa Gateway*. Dragino. Apr. 2018. URL: https://www.dragino.com/downloads/downloads/UserManual/LG01_LoRa_Gateway_User_Manual.pdf.

[3] *HopeRF RFM95W Low Power Long Range Transceiver Module*. HopeRF. 2018. URL: https://cdn.sparkfun.com/assets/learn_tutorials/8/0/4/RFM95_96_97_98W.pdf.

[4] *How To Run ATmega328P For a Year On Coin Cell Battery*. 2021 [Online Archive]. URL: https://www.programmersought.com/article/95203698332/.

[5] rocketscream. *Low-Power*. 2018. URL: https://github.com/rocketscream/Low-Power.

[6] dragino. *Radiohead*. 2018. URL: https://github.com/dragino/RadioHead.

[7] Bill Earl. *Memories of an Arduino*. 2013 [Online]. URL: https://learn.adafruit.com/memories-of-an-arduino.

[8] *Fat Tree*. 2021 [Online]. URL: https://en.wikipedia.org/wiki/Fat_tree.

Appendix

A Dragino LG01 Logging Setup

Data logging is an optional step in the protocol handled by the user. Though there are other ways to log received data, or alternatives like uploading the data to a server via the Internet, we chose to save data on a local USB drive; local logging is secure, reliable, and easily implementable.

To enable USB logging on the Dragino LG01-P, follow these hardware instructions and look at the example code in "SEEL_sensor_node.ino" (change the log path appropriately). A further step (not mentioned here) is to create scheduled backups of the log files on the Dragino device via Cron Jobs.

1. Insert USB device into the Dragino LG01-P.
2. SSH into the Dragino device (its address can be found by viewing the Arduino Ports) via the format "root@X.X.X.X", the default password is "dragino".
3. Check if the USB device is mounted (if the files are visible). If they are, ignore the following instructions; the USB is ready to be written to.
4. The USB device needs to be mounted before use, and this process should be automatic in case the Dragino device restarts. Test mounting the device. Create a folder for the device in the root directory. Find the USB device in "root/../dev/" (likely "sda1"). Move to the root directory and mount using the command "mount ../dev/< device> <created_folder>/". If successful, add the command to "/etc/rc.local", or follow the next steps to change "rc.local" via the Dragino UI.
5. Access the Dragino device's system via web browser using the IP Address: "10.130.1.1". The default username is "root" and the default password is "dragino". More detailed instructions can be found in [2].
6. Navigate to System → Startup and scroll down to Local Startup. Paste the following command "mount ../dev/<device> ../root/<created_folder>". Note the USB device may change names, find the name assigned right after starting up the device (unplugging and plugging the USB device changes the assigned device name).
7. Verify the USB device is automatically mounted on the Dragino device's startup.

B SNODE Lifetime Analysis

The generic expression for calculating NODE lifetime (L) is given by:

$$L = \frac{B}{\Sigma(S_i * A_i)}$$

Where B represents the battery capacity, S_i represents the percentage ($\Sigma(S_i) = 1$) of time the system spends in state i, and A_i represents the average current while in state i. Note the units of A_i should match the units of B. This equation does not account for factors such as Watchdog wake-up costs, battery self-discharge, and that the typical capacity of the battery (the estimate of 1,500 mAh) may not match the SNODEs' operational voltage limit. Thus, treat the lifetime estimate from the equation given as an upper-bound.

In Section 6.2, we calculate the maximum lifetime of a SEEL SNODE (using our hardware) to be around 32 years if it is permanently sleeping. However, SNODEs will realistically not be permanently sleeping but also receiving and transmitting data. The following example demonstrates how to calculate the upper-bound SNODE lifetime based on several assumptions and past results.

- Similar to the assumption made in Section 6.2, we assume the typical battery capacity of a CR123 to be 1500 mAh.

- We assume the SNODE will wake for five minutes per day (0.35% duty cycle). Thus, the SNODE will be awake for 0.35% of the time and asleep for 0.9965% of the time.
- Taking results from Table 6.1, we set the average sleep state current to be 5.41 µA and the average listening state current to be 15.1 mA.
- Assuming no packet loss and no packet forwarding, the SNODE transmits two messages per cycle. We measure the average time spent in the transmit state to be 660 ms. Thus every cycle, the SNODE spends on average 1.32 s in transmission mode. Note the average transmission state current and average time spent in the transmit state will depend on LoRa parameters; our parameters for the measurements used here are listed in Section 6.2.

The awake state includes the listening state and the transmitting state. Since SNODEs spend 1.32 s in the transmission state, the remaining awake time (298.68 s) is spent in the listening state. There are 86400 s in a day. Therefore, the SNODE in this example is in the transmitting state for $1.32\,\text{s}/86400\,\text{s} * 100\% \approx 0.0015\%$ of its life, in the listening state for roughly 0.3457% of its life, and in the sleeping state for roughly 99.6528% of its life. Putting everything together, we calculate the maximum lifetime of the SNODE in this example to be $1500\,\text{mAh}/(0.000015 * 126\,\text{mA} + 0.003457 * 15.1\,\text{mA} + 0.996528 * 0.00541\,\text{mA}) = 25217.75$ hours, or roughly 2.88 years.