Using LoRa WAN and SWARM for an ultra-scalable sensor data gathering solution

Bill Brouwer 01/2024

Introduction

Long Range Wide Area (LoRa) Network IoT offerings have proliferated since the introduction of the technology, based on chirp spread-spectrum communication, robust under noise and allowing transmission over large distances. Coupled with the decreasing cost and increasing computational power of embedded processors, Internet of Things (IoT) connected devices offer unparalleled levels of distributed remote monitoring sophistication.

Similarly, communication methods allowing upload of remote monitoring data from point of acquisition to cloud have increased in coverage while decreasing in cost, the two most common techniques being cellular (eg., Long Term Evolution or LTE) and various satellite providers, with uplinks that may also overlap with LTE.

In the last few years, the introduction of Starlink from Space-X has put satellite internet connectively on par with cellular, in terms of both costs as well as bandwidth, while offering higher global coverage. SWARM, now also under the purview of Space-X, is a related provider of dramatically lower bandwidth and cost edge solutions.

In this work, we explore using a popular LoRa IoT device based on Wio-E5 Wireless Module (STM32WLE5JC) equipped with ARM Cortex-M4 and SX126x RF transceiver in a single embedded package, supporting EU868 and US915 frequency bands, to name two. To these devices we connect a variety of sensors whose analog response is digitized using TI ADS1115 ADCs before communication over I2C to nodes, and subsequently via LoRa WAN to host hub (based on SX1302), before transmission to cloud using SWARM.

Proof of Concept

Overview

The solution detailed in this document comprises at least five main hardware components:

- Edge (IoT) nodes that gather equipment and environmental data and communicate sensor data via radio (LoRa) link.

- An aggregator or gateway for receiving said data and managing the LoRa Wide Area Network (WAN).
- A REST API / service capable of communicating with the SWARM modem and amenable for use with microservice applications.
- The SWARM modem and transmitter itself.
- A remote or cloud-based REST service for receiving and decoding payloads

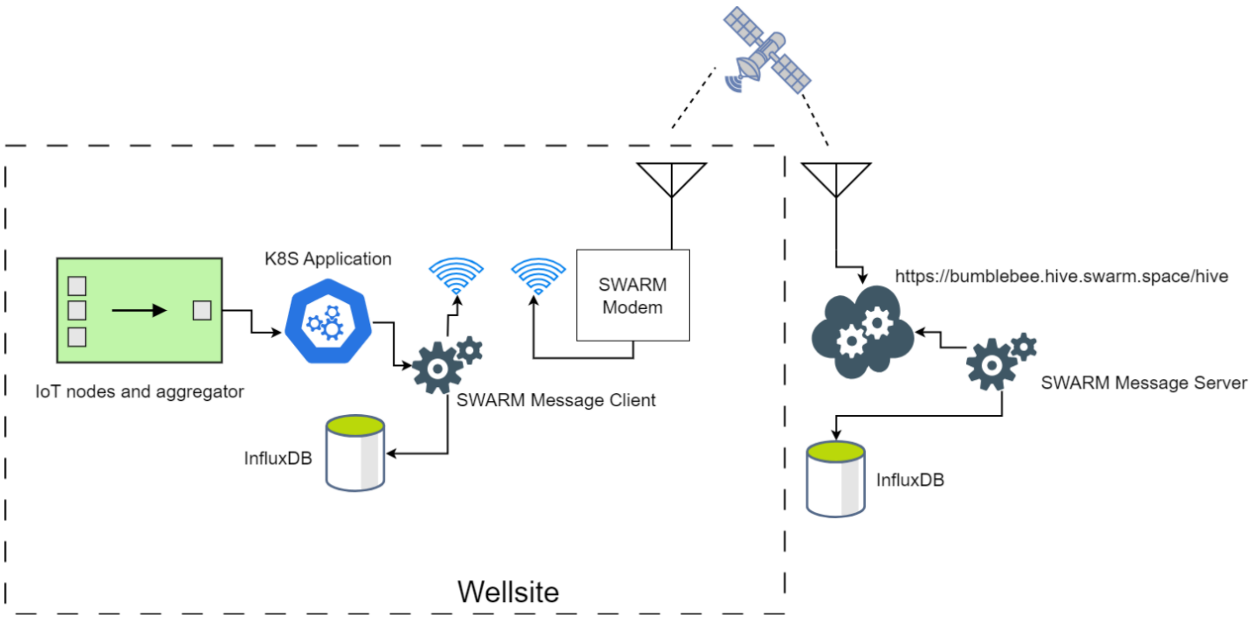


Figure 1: Proposed architecture

Hardware Components

SEEED Wio-E5 Edge node

The following table details the important features of the Wio-E5 module used for IoT nodes in the POC.

Table 1: Wio-E5 (STM32WLE5JC) Features

Feature	**Details**
Core	32-bit ARM Cortex-M4 CPU, =>48 Mhz
LoRaWAN Stack	AT Command based, frequency bands including EU868/US915/AU915/AS923/KR920/IN865
Package	12x12 mm, 2 pins SIMD
Interfaces	UART, I2C, ADC, SPI, GPIO
Sensitivity	-116.5dBm (SF5), -121.5 dBm (SF7), -136dBm (SF12)

Modulation	LoRa, G/FSK,G/MSK, BPSK
Certification	FCC,CE,TELEC
Power Supply	1.8V - 3.6V
RF Output Power	<= 20.8 dBm @ 3.3V

For development purposes, two LoRa-E5 dev boards were purchased from Mouser. As alluded to in the table, settings can be interrogated and set over UART interface using basic AT commands. For example, after connecting with a Linux (Ubuntu 22) host via USB, the device is available on /dev/ttyUSB0 and using picocom one can quickly determine important details eg., via issuing the command AT+ID

```
 AT+ID

+ID: DevAddr, 32:30:86:9A
+ID: DevEui, 2C:F7:F1:20:32:30:86:9A
+ID: AppEui, 80:00:00:00:00:00:00:06
```

The DevEui is an ID in the IEEE EUI64 address space used to identify a device, supplied by the manufacturer. During activation (connection with aggregator/gateway, to be discussed) DevAddr is assigned to the device. This DevAddr is used in the LoRaWAN protocol afterwards and the DevEui is sent unencrypted. The AppEui (now referred to as JoinEui) is a global application ID in the IEEE EUI64 address space identifying the join server (gateway) during activation.

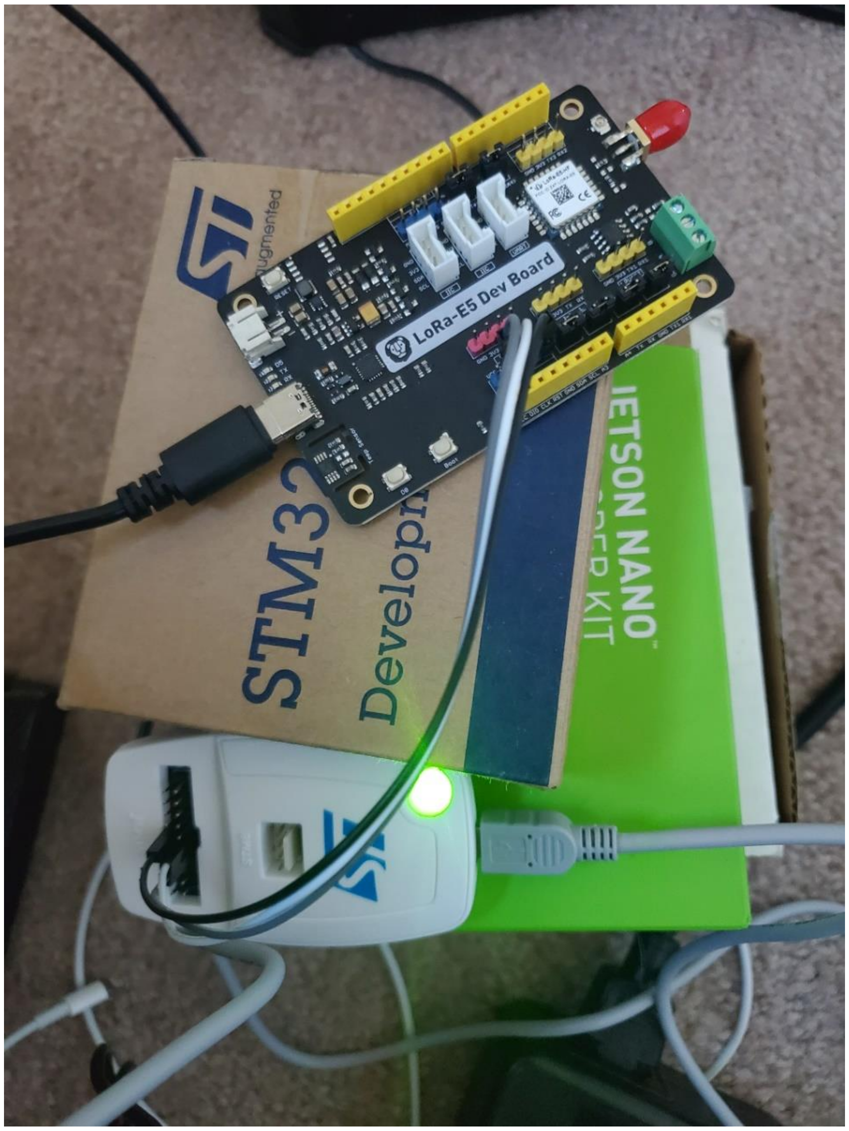


Figure 2: SEEED LoRa-E5 dev board with ST-LINK interface connected to an Ubuntu 22 machine

In order to take advantage of peripherals eg., the I2C interface, it is necessary to program and flash the MCU. There is a very large software base behind STMicroelectronics devices and associated development boards.

Unlike official dev kits, the SEEED board used in this work doesn't contain the additional hardware used to flash and debug the MCU, so an ST-LINK V2 programmer was purchased and used, using documentation to implement to correct JTAG connections.

This worked well with the open source st-link toolset, when installed on Ubuntu 22 using an official deb package eg., the output when connected to ST-LINK as per the figure (with reset held down):

```
$ sudo st-info --probe --connect-under-reset
```

```
Found 1 stlink programmers
  version:    V2000000
  serial:     38FF6F06304E4B3046122243
  flash:      0 (pagesize: 0)
  sram:       0
  chipid:     0x0497
```

The chipid (0x0497) enumerated correctly and was used in conjunction with the STM32CubeIDE to identify physical details (Appendix A), used to select appropriate hardware abstraction layer (HAL) and middleware modules for peripherals and set parameters when flashing the device.

As mentioned, a sensor array is connected to an I2C bus by way of the TI ADS111 (16 bit) device, capable of digitizing up to four inputs. While the main MCU contains an ADC, it is lower resolution and apparently a single channel.

The sensors selected for testing are from the MQ series, capable of measuring CH4, volatile gas and many more examples when calibrated correctly.

- MQ-2 (~H2)
- MQ-5 (~LPG)
- MQ-135 (~Benzene)
- MQ-9 (~CH4)

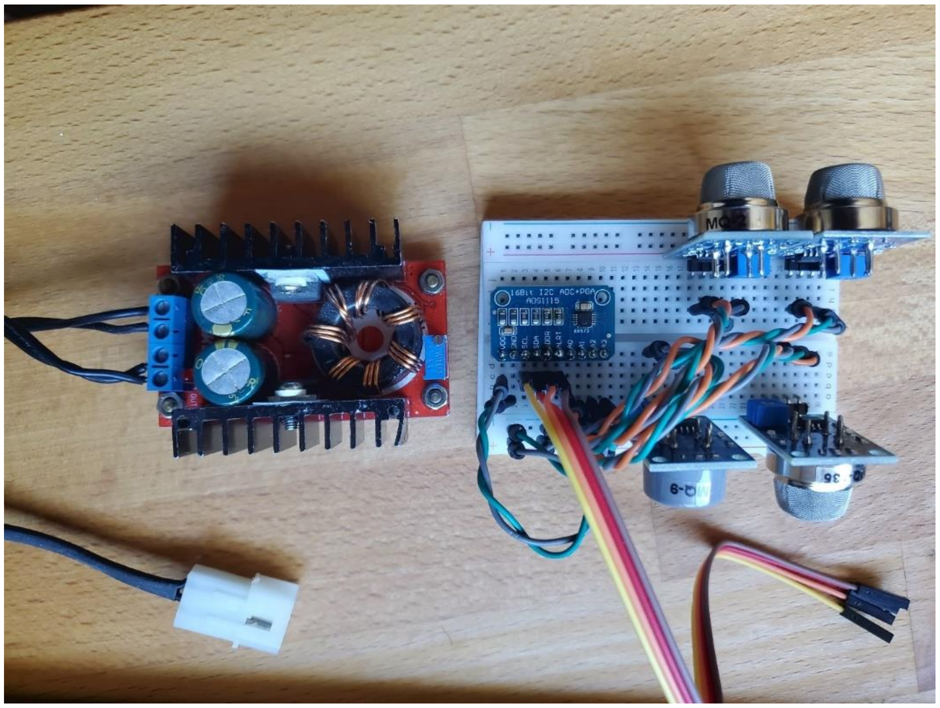


Figure 3: Four sensor array with ADC and I2C interface, powered by a lab supply for testing

The important configuration parameters used on conjunction with the boilerplate used to flash the device include the ACTIVE_REGION (LORAMAC_REGION_US915) as well as EUI from the gateway, discussed next.

Waveshare LoRa/LoRaWAN Gateway

This solution from Waveshare provides a wide area network for IoT nodes, based on the Semtech SX1302 module. The particular model purchased and used was selected for use in the US (US915).

An Nvidia Jetson Nano was used as host, based on previous successful benchmarking and testing of the device, in the context of performing video processing and machine learning tasks at the edge.

The SX1302 module is accessible using the serial parallel interface (SPI) to read/write registers, by way of the PCIe bridge. The developer kit in turn provides a Hardware At Top (HAT) interface suitable for bridging between the host and SX1302.

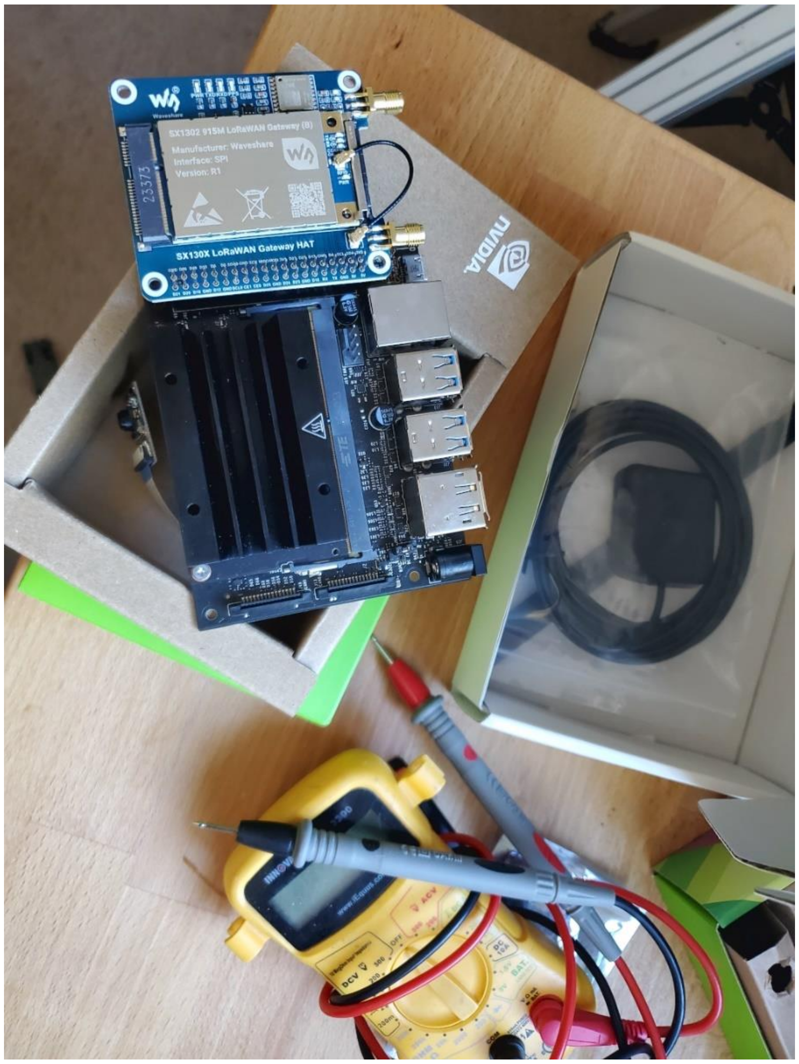


Figure 3: Waveshare LoRa gateway attached to Nvidia Jetson host by way of the HAT interface.

Testing in this way provided an easy way to debug hardware without need of oscilloscope, by checking pin state with multimeter or by way of LED to GND.

By studying the SX1302 reference architecture, it was discovered that while the Pulse Per Second (PPS) input from Global Navigation Satellite System (GNSS) is listed as optional input, the Waveshare implementation apparently requires PPS and the GNNS device even without satellite connection must function.

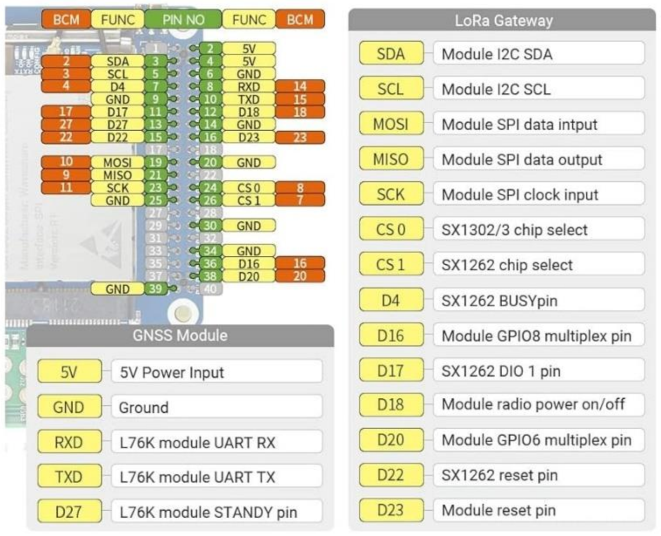


Figure 4: Pinout for Waveshare implementation of SX1302 w/ L76K GNNS

Pin states that were incorrect on boot were manipulated using the GPIO fs method on the host eg., for setting the L76K standby pin HIGH:

```
>sudo echo 14 > /sys/class/gpio/export
>sudo echo out > /sys/class/gpio/gpio14/direction
>sudo echo 1 > /sys/class/gpio/gpio14/value
```

As mentioned, the SX1302 and companion radio SX1262 (spectral scan tasks) require serial or more commonly SPI communication, and this was configured (SPI0) using the Jetson IO tools.

Drivers once available in the linux kernel were tested using a simple loopback test (shorting MOSI/MISO pins) :

```
$ ./spidev_test -D /dev/spidev0.1 -v -s 1000000

spi mode: 0x0
bits per word: 8
max speed: 1000000 Hz (1000 kHz)
TX | FF FF FF FF FF FF 40 00 00 00 00 95 FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF F0 0D  |......@.........................|
RX | FF FF FF FF FF FF 40 00 00 00 00 95 FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF F0 0D  |......@.........................|
```

Aside from correctly functioning SPI, the SX1302 apparently depend strongly on being able to reset the radio via a specific GPIO pin. This was deduced using the Jetson device tree source (DTS) file for hardware architecture, again by examining pin state using multimeter. Thereafter, SX1302 drivers and test programs appeared to work well.

SWARM

SWARM is a relatively new company that proclaims itself to be the world’s lowest cost, global connectivity for IoT devices, with data plans beginning at 5 USD / month. SWARM relies on a growing constellation of ultra-small satellites, with up/down links operating in the VHF range and plans to introduce direct to cellular network communication as well. At the time of writing, the transmitter provides three interfaces; one direct USB/serial connection, one a basic web client and the third is telnet over WiFi network originating from the device. The modem itself and telnet command/responses follow the NMEA communication specification, Table 2 list details of key features.

Table 2: SWARM transmitter characteristics

Feature	**Details**
Components	GPS, VHF/LTE, ARM Cortex-M4 processor, indicator LEDs, 3.3V serial UART interface, 3.3V GPIO
Sensors	Onboard GPS (lat/lon/alt), CPU temperature
Dimensions	51.0 mm x 30.0 mm x 5.3mm
Mass	9.6 kg
Power	Sleep mode (3.3V): 80 μA (max) Receive mode (3.3V) 26 mA (typ), 40 mA (max) Transmit mode (3.3V) 850 mA (typ), 1000 mA (max)
Environment	Operational: -40 C to +85 C Storage: -40C to +85 C
Command Interface	3.3V Serial UART
Bit Rate	1 kbps
Frequency	137-138 MHz (down) 148-150 MHz (up)

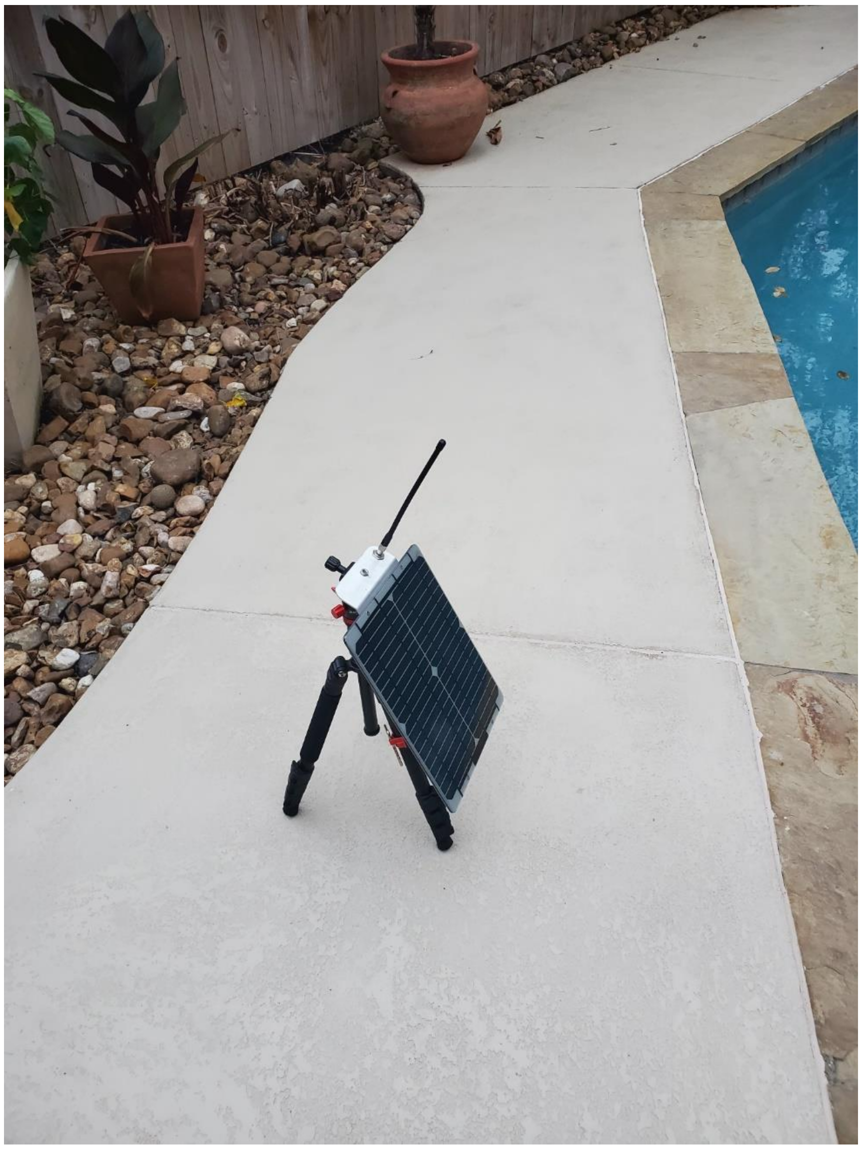
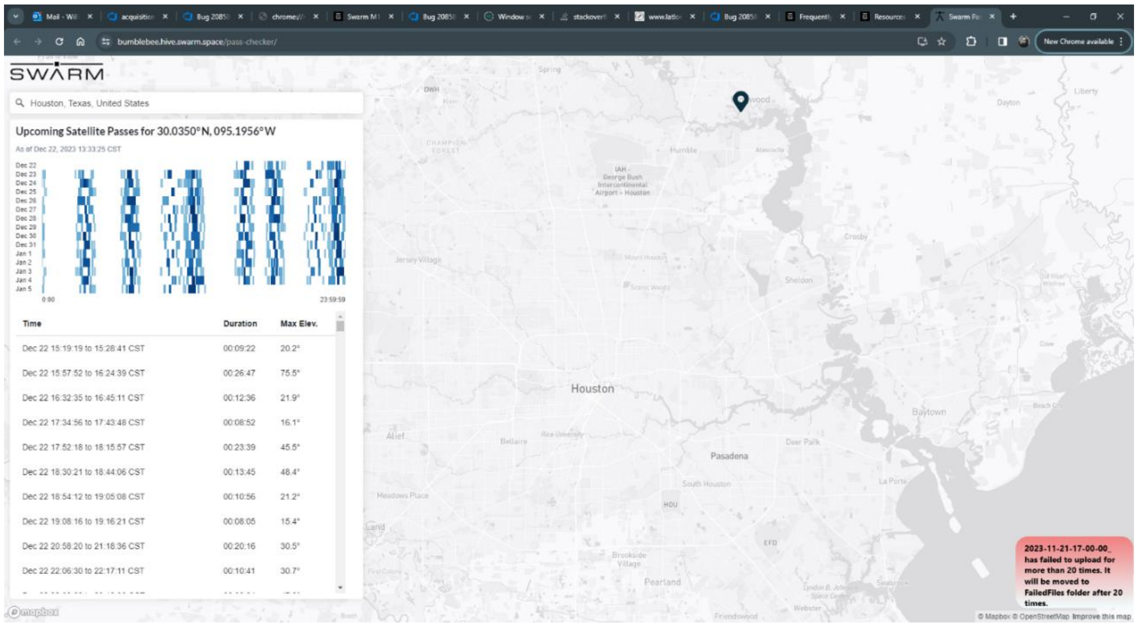


Figure 5: SWARM transmitter unit

Reception time of packets at cloud obviously a function of satellite pass-over and frequency, but buffered on the device and a maximum of 192 bytes in size, provides sufficient bandwidth to report dozens of measurements hourly.

The figure below is taken from https://bumblebee.hive.swarm.space/ (requires account/login) eg., for the day in question (12/23/23) there were at least ten independent satellite passes for the Houston area, of different durations.


bumblebee.hive.swarm.space/pass-checker/
SWARM
Houston, Texas, United States
Houston
Pasadena
Baytown
Pearland
0:00
23:59:59

Upcoming Satellite Passes for 30.0350°N, 095.1956°W

As of Dec 22, 2023 13:33:25 CST

Time	Duration	Max Elev.
Dec 22 15:19:19 to 15:28:41 CST	00:09:22	20.2°
Dec 22 15:57:52 to 16:24:39 CST	00:26:47	75.5°
Dec 22 16:32:35 to 16:45:11 CST	00:12:36	21.9°
Dec 22 17:34:56 to 17:43:48 CST	00:08:52	16.1°
Dec 22 17:52:18 to 18:15:57 CST	00:23:39	45.5°
Dec 22 18:30:21 to 18:44:06 CST	00:13:45	48.4°
Dec 22 18:54:12 to 19:05:08 CST	00:10:56	21.2°
Dec 22 19:08:16 to 19:16:21 CST	00:08:05	15.4°
Dec 22 20:58:20 to 21:18:36 CST	00:20:16	30.5°
Dec 22 22:06:30 to 22:17:11 CST	00:10:41	30.7°

2023-11-21-17-00-00_ has failed to upload for more than 20 times. It will be moved to FailedFiles folder after 20 times.

Figure 6: Dashboard showing SWARM satellite passes over Houston on 12/23

At the time of writing, SWARM is licensed for usage in several dozen countries and international waters.

Software Solutions

SWARM Message Client

The interfaces provided to interact with the SWARM modem are good for quick testing and periodic use, but not for integration with software applications.

For this reason, a client service was developed, for inclusion in service-oriented architecture (SOA) software solutions.

The new service is based on node.js and a small number of packages that abstracts away the SWARM telnet interface, provided over the local WiFi network of the transmitter, and offers two simple REST API methods:

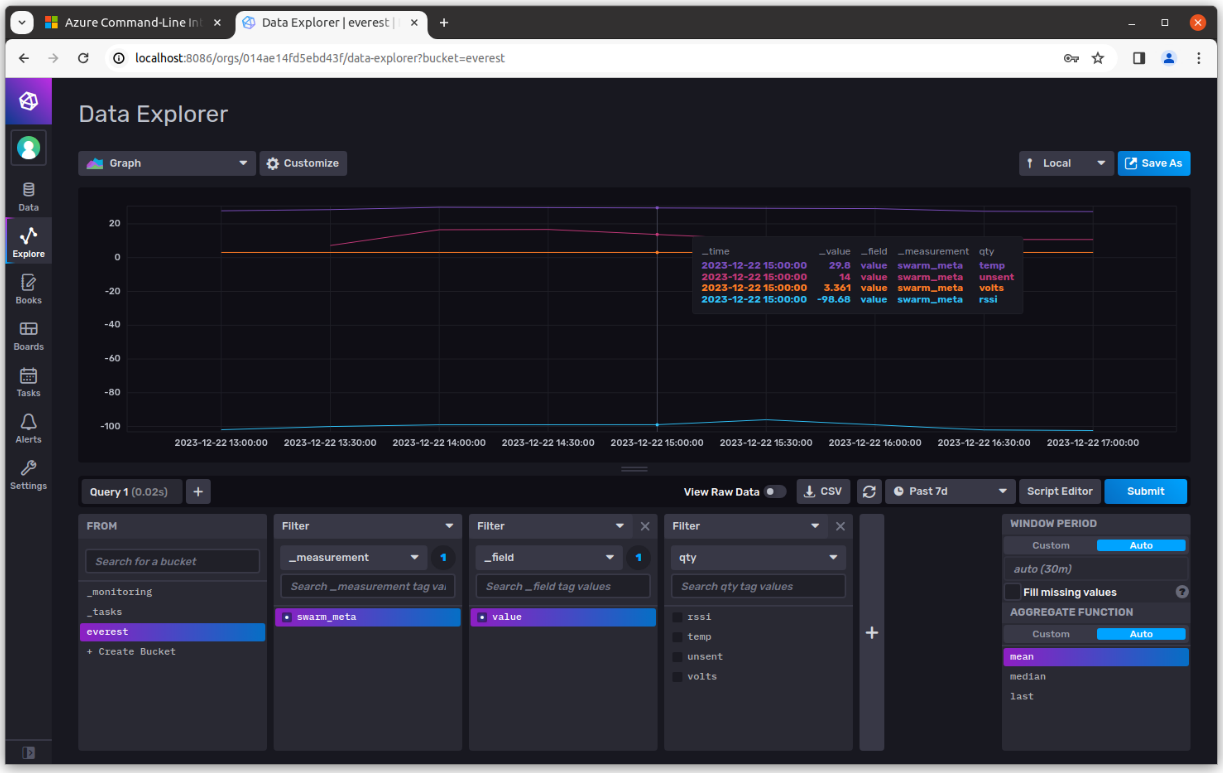
- GET /meta
  - Get the current meta data for the modem, JSON fields:
    - Voltage
    - CPU temperature
    - Latitude and Longitude
    - UNIX time
- POST /upload
  - Upload data to local modem and eventually cloud, fields:
    - Array of name,value pairs as JSON

The meta (data) route is important for understanding the state of the modem; without a GPS lock for example, transmission to satellites won't take place.

Voltage and temperature serve as zeroth order measurements of device health.

This data as well as the number of unsent messages (buffered at the modem) and Received Singal Strength Indicator (RSSI) are also logged in InfluxDB, figure 5.

Referring to the figure, note that over time the number of unsent messages increases first as they are pushed to the modem, then decreases as they are transmitted by VHF uplink to a satellite during a pass over event.

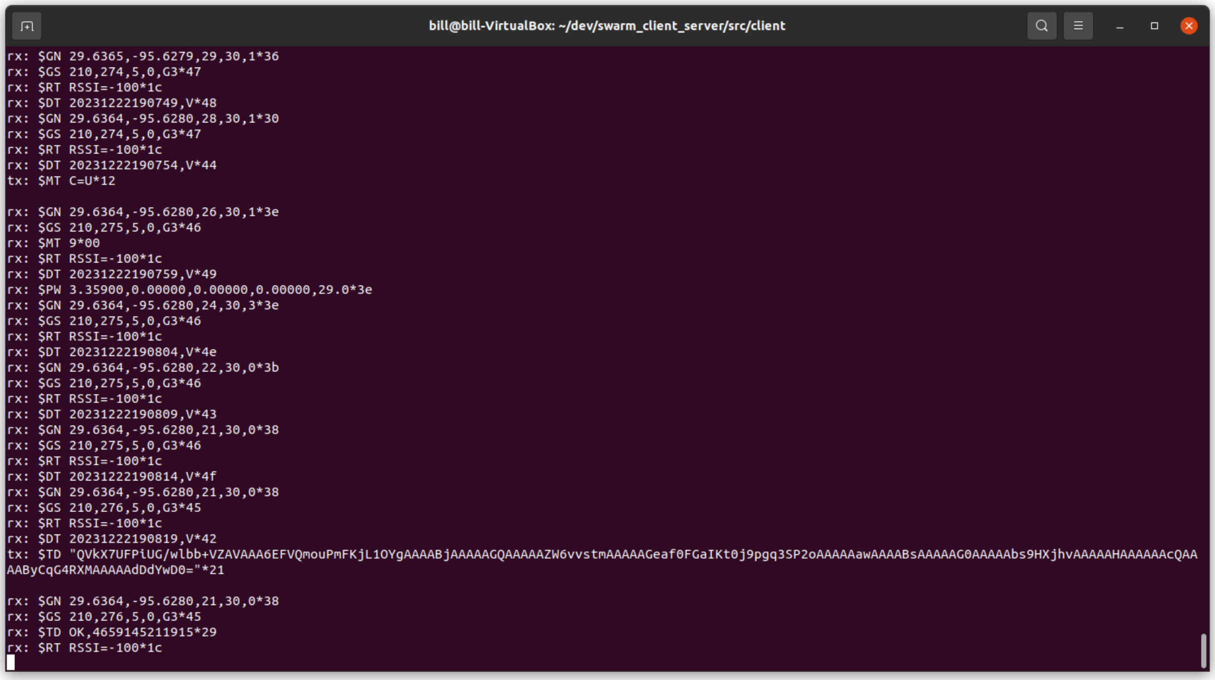


_time	_value	_field	_measurement	qty
2023-12-22 15:00:00	29.8	value	swarm_meta	temp
2023-12-22 15:00:00	14	value	swarm_meta	unsent
2023-12-22 15:00:00	3.361	value	swarm_meta	volts
2023-12-22 15:00:00	-98.68	value	swarm_meta	rssi

Figure 6: Screenshot of InfluxDB dashboard showing data series captured from SWARM transmitter, showing RSSI, number of unsent messages, CPU temperature and device voltage.

As mentioned, the main purpose of the new client service is to provide a simple method for uploading data, taking care of telnet (NMEA) commands, payload compression and chunking etc. Once a telnet connection is established with the modem, NMEA format messages arrive every second, providing information including location and RSSI. The message processing loop is also used to check the (threadsafe) outbound message queue, populated when other applications use the upload API method of the service.

Visible in the figure below is the push of a message ($TD command) which has been chunked and encoded by the service.



```
rx: $GN 29.6365,-95.6279,29,30,1*36
rx: $GS 210,274,5,0,G3*47
rx: $RT RSSI=-100*1c
rx: $DT 20231222190749,V*48
rx: $GN 29.6364,-95.6280,28,30,1*30
rx: $GS 210,274,5,0,G3*47
rx: $RT RSSI=-100*1c
rx: $DT 20231222190754,V*44
tx: $MT C=U*12

rx: $GN 29.6364,-95.6280,26,30,1*3e
rx: $GS 210,275,5,0,G3*46
rx: $MT 9*00
rx: $RT RSSI=-100*1c
rx: $DT 20231222190759,V*49
rx: $PW 3.35900,0.00000,0.00000,0.00000,29.0*3e
rx: $GN 29.6364,-95.6280,24,30,3*3e
rx: $GS 210,275,5,0,G3*46
rx: $RT RSSI=-100*1c
rx: $DT 20231222190804,V*4e
rx: $GN 29.6364,-95.6280,22,30,0*3b
rx: $GS 210,275,5,0,G3*46
rx: $RT RSSI=-100*1c
rx: $DT 20231222190809,V*43
rx: $GN 29.6364,-95.6280,21,30,0*38
rx: $GS 210,275,5,0,G3*46
rx: $RT RSSI=-100*1c
rx: $DT 20231222190814,V*4f
rx: $GN 29.6364,-95.6280,21,30,0*38
rx: $GS 210,276,5,0,G3*45
rx: $RT RSSI=-100*1c
rx: $DT 20231222190819,V*42
tx: $TD "QVkX7UFPiUG/wlbb+VZAVAAA6EFVQmouPmFKjL1OYgAAAABjAAAAAGQAAAAAZW6vvstmAAAAAGeaf0FGaIKt0j9pgq3SP2oAAAAAawAAAABsAAAAAG0AAAAAbs9HXjhvAAAAAHAAAAAAcQAA
AAByCqG4RXMAAAAAdDdYwD0="*21

rx: $GN 29.6364,-95.6280,21,30,0*38
rx: $GS 210,276,5,0,G3*45
rx: $TD OK,4659145211915*29
rx: $RT RSSI=-100*1c
```

Figure 7: Screenshot of SWARM client service log showing messages received (rx) from the modem, as well as a message pushed (tx) to the modem for upload via satellite to cloud.

An example upload POST request input supplied by an application:

[{"name":"AcqTime","value":1590043848},{"name":"MRUDATA_PITCH","value":0},{"name":"CT_SPEED","value":0},{"name":"CT_OD_MIN","value":0},{"name":"WH_PRESS","value":-24992800},{"name":"CT_WALL_MIN","value":0},{"name":"CT_TENSION","value":12370.5},{"name":"CT_MEAS_DEPTH","value":1.64592},{"name":"CT_CORR_DEPTH","value":1.64592},{"name":"UTLM_CALCULATED_DEPTH","value":0},{"name":"CT_OD_OVALITY","value":0},{"name":"CT_WALL_AVG","value":0},{"name":"N2_VOLUME","value":0},{"name":"TOTAL_PUMP_RATE","value":0.0000529958},{"name":"UTLM_CALCULATED_SPEED","value":0},{"name":"TOTAL_N2_RATE","value":0},{"name":"TOT_FLUID","value":0},{"name":"CT_WEIGHT","value":5891.23},{"name":"CT_OD_MAX","value":0},{"name":"MRUDATA_ROLL","value":-0.05885435},{"name":"CT_WALL_MAX","value":0},{"name":"CIRC_PRESS","value":182711}]

The client takes the numerical values supplied as text and converts values to single precision binary, interleaved in a longer base-64 binary string by single character (ASCII) header characters. As such, header mappings are created and supplied as separate strings, all of which conform to the 192 byte independent message limit of the SWARM modem, below.

```
{
  payload:
'Qd4h7UFPQY6+wlYzM1NAVGZmIEJVW5cuPmFNjL1OYgAAAABjTisFO2QAAAAAZRCuvstmAAAAAGea2XdGaPjfyj9p+N/KP2oAAAAAawAAAABsAAAAAG0AAAAAbs9HXjhvAAAAAHAAAAAAcQAAAAByPH8RRnMAAAAAdLsIrLs=',
  headers: [
'{"A":"lat","O":"lon","V":"volts","T":"temp","U":"unix_time","a":"AcqTime","b":"MRUDATA_PITCH"
,"c":"CT_SPEED","d":"CT_OD_MIN","e":"WH_PRESS","f":"CT_WALL_MIN","g":"CT_TENSION"}',

'{"h":"CT_MEAS_DEPTH","i":"CT_CORR_DEPTH","j":"UTLM_CALCULATED_DEPTH","k":"CT_OD_OVALITY","l":
"CT_WALL_AVG","m":"N2_VOLUME","n":"TOTAL_PUMP_RATE","o":"UTLM_CALCULATED_SPEED"}',
    '{"p":"TOTAL_N2_RATE","q":"TOT_FLUID","r":"CT_WEIGHT","s":"CT_OD_MAX","t":"MRUDATA_ROLL"}'
  ]
}
```

SWARM Message Server

Once received at the SWARM cloud server and subsequently downloaded, original messages must be reconstructed, using header information arriving as independent messages.

Below is an example of a message downloaded from the SWARM server using the REST API, showing an encoded data payload.

```
  {
    packetId: 6972754268233472,
    messageId: 6972754268233472,
    deviceType: 1,
    deviceId: 15562,
    direction: 1,
    dataType: 6,
    userApplicationId: 0,
    organizationId: 65822,
    len: 168,
    data:
'UWZBVzdVRlBpVUcvd2xiYitWWkFWQUFBNEVGVlAyb3VQbUZLakwxT1lnQUFBQUJqQUFBQUFHUUFBQUFBWlJDdXZzdG1BQ
UFBQUdjQVNrRkdhSUt0MGo5cGdxM1NQMm9BQUFBQWF3QUFBQUJzQUFBQUFHMEFBQUFBYnM5SFhqaHZBQUFBQUhBQUFBQUF
jUUFBQUFCeTF4bTRSWE1BQUFBQWRFSVJjYjA9',
    ackPacketId: 0,
    status: 0,
    hiveRxTime: '2023-12-22T21:07:49'
  }
```

Once original name-value pairs have been reproduced, the SWARM message service pushes received data to InfluxDB, including relevant meta data such as coordinates and UNIX time of data gathering at the client application.

Conclusions and Proposed Work

In this document we have briefly laid out the details of a proof of concept, that seeks to provide a radically more cost effective and scalable solution for environmental and equipment monitoring data gathering and transmission.

We used widely available components including industry standard devices for LoRa wide area networks, as well a computationally powerful, low energy host device (Nvidia Jetson Nano) capable of sensor gathering and sophisticated data processing if needed.

We assembled and tested an edge device capable of digitizing four analog sensor inputs before transmission to a SX1302 based gateway device. In the proposed solution, we completely eliminate the need for an attached cellular backhaul, and replace it instead with a SWARM uplink.

Note that this also provides a significant security benefit, in the sense that the attack surface over network is reduced or eliminated altogether.

The SWARM modem interfaces are unwieldy, hence we developed client and server solutions that make integration with SLB SOA solutions easier.

The hardware components used in this work were as follows:

- SWARM evaluation (modem/transmitter) kit ~ 400 USD
- Waveshare SX1302 concentrator/LoRa gateway ~ 120 USD
- 4Gb NVIDIA Jetson Nano ~ 200 USD
- 2xSEEED E5 LoRa Dev Kits ~ 60 USD
- MQ Sensors and ADS111 ADC ~ 20 USD

Some observations developed during testing:

- The SWARM transmitter requires a particular threshold RSSI before data transmission to satellites is possible ie., areas where there is significant radio interference exists, it may not be possible to use the uplink. This will likely no longer be a concern when SWARM switches to LTE uplinks.
- The SWARM dev kit was susceptible to vibration eg., a display panel dislodged at least once, and the solar panel was insufficient in size for charging during cloudy days, the latter easily rectified.

Further work could be dedicated to calibrating sensors correctly, and understanding the exact measurements of interest to stakeholders, as well as business systems that might be targets for data ingestion.

As with the SWARM dev kit, some hardening of IoT edge and gateway devices should be conducted, as well as an extended study of power requirements under load and in the target operating environments.

References

- Open source stlink toolset : https://github.com/stlink-org/stlink
- SEEED LoRa E5 dev (edge) board: https://wiki.seeedstudio.com/LoRa_E5_Dev_Board/
- STM32WLE* datasheet: https://www.st.com/resource/en/datasheet/stm32wle5jc.pdf
- SEEED LoRa-E5 AT command manual : https://files.seeedstudio.com/products/317990687/res/LoRa-E5%20AT%20Command%20Specification_V1.0%20.pdf
- ADS111 ADC datasheet: https://www.ti.com/lit/ds/symlink/ads1115.pdf
- I2C bus specification: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

- SWARM modem manual : https://swarm.space/wp-content/uploads/2022/08/Swarm-M138-Modem-Product-Manual.pdf
- NMEA reference manual: https://www.sparkfun.com/datasheets/GPS/NMEA%20Reference%20Manual-Rev2.1-Dec07.pdf
- ARM GNU toolchain: https://developer.arm.com/downloads/-/gnu-rm
- STM32 MCUs and MPUs programmer manual: https://www.st.com/resource/en/programming_manual/pm0214-stm32-cortexm4-mcus-and-mpus-programming-manual-stmicroelectronics.pdf
- ST-LINK V2 programmer specifications : https://www.st.com/resource/en/user_manual/dm00026748-st-link-v2-in-circuit-debugger-programmer-for-stm8-and-stm32-stmicroelectronics.pdf
- ST-LINK pinout details: https://stm32-base.org/guides/connecting-your-debugger.html
- Waveshare LoRa gateway wiki: https://www.waveshare.com/wiki/SX1302_LoRaWAN_Gateway_HAT
- Semtech SX1302 LoRa baseband gateway transceiver : https://files.waveshare.com/upload/d/df/DS_SX1302_V1.0.pdf
- Nvidia Jetson datasheet : https://slb001-my.sharepoint.com/:b:/g/personal/wbrouwer_slb_com/EeIyPNmOuiNOtStJF0_Q79QBtNHMBCfhemxN4LbelppWew?e=HBD005
- Nvidia Jetson developer guide: https://docs.nvidia.com/jetson/archives/r34.1/DeveloperGuide/index.html

Appendix A

```
<mcu>
  <name>STM32WLE5JCIx</name>
  <status>Active</status>
  <dieId>0497</dieId>
  <parent>stm32wlsingle</parent>
  <cpus>
    <cpu>
      <id>0</id>
      <svd>
        <name>STM32WLE5_CM4.svd</name>
        <version>v1r4</version>
        <url></url>
      </svd>
      <cores>
        <core>
          <id>0</id>
          <apnum>0</apnum>
          <type>arm cortex-m4</type>
```

```xml
        <fpu>None</fpu>
      </core>
    </cores>
  </cpu>
</cpus>
<package>UFBGA73</package>
<!--XMLfilename = STM32WLE5JCIx-->
<!--STVPdevicename = STM32WLExxC-->
<memories>
  <memory>
    <type>RAM</type>
    <name>RAM</name>
    <address>0x20000000</address>
    <size>0x10000</size>
    <!--end = 0x2000FFFF-->
  </memory>
  <memory>
    <type>ROM</type>
    <name>FLASH</name>
    <address>0x8000000</address>
    <size>0x40000</size>
    <!--end = 0x803FFFF-->
  </memory>
</memories>
<CDefines>
  <CDefine>-DSTM32WLE5JCIx</CDefine>
</CDefines>
</mcu>
```

Appendix B

1. Node.js SWARM data set/get methods for client/server

```javascript
const alpha = 'abcdefghijklmnopqrstuvwxyz'.split('');


function set_mapping(label,name,map){

  const new_entry={};
  new_entry[label]=name;

  let dict = map[map.length-1];
  let dict_copy =JSON.parse(JSON.stringify(dict));
  dict_copy[label]=name;

  if (JSON.stringify(dict_copy).length>180){
    map.push(new_entry);
  }else{
    map[map.length-1] =dict_copy;
  }
}

function get_data(payload,dict){

  let output = {};

  if (!payload || !dict) return output;
  const buffer =  Buffer.from(payload, 'base64');

  let index=0;

  for (;;){

    let k = buffer.toString('utf-8',index,index+1);
    if (dict[k]) k=dict[k];
    index++;
    output[k] = buffer.readFloatLE(index);
    index+=4;

    if (index >= buffer.length) break;
  }

  output.unix_time*=1e13;

  return output;
}

function set_data(msg,meta){

  const buffer = Buffer.alloc(192);
  let msg_array=[];

  try{
    // is string input
    msg_array=JSON.parse(msg);
  }catch(err){

    for (let x of msg)
      if (x.name === undefined || x.value === undefined)
```

```
        return {headers:[],payload:''};

      msg_array = msg;
    }


    let index=0;
    let pos=0;

    buffer.write('A',index++,'utf-8');
    buffer.writeFloatLE(meta.lat,index);
    index+=4;

    buffer.write('O',index++,'utf-8');
    buffer.writeFloatLE(meta.lon,index);
    index+=4;

    buffer.write('V',index++,'utf-8');
    buffer.writeFloatLE(meta.volts,index);
    index+=4;

    buffer.write('T',index++,'utf-8');
    buffer.writeFloatLE(meta.temp,index);
    index+=4;

    const t = new Date();
    buffer.write('U',index++,'utf-8');

    const ts = Number(t.getTime()) / 1e13;
    meta.unix_time=ts;
    buffer.writeFloatLE(meta.unix_time,index);
    index+=4;

    let
  header_map=[{'A':'lat','O':'lon','V':'volts','T':'temp','U':'unix_time'}];

    for (const x of msg_array) {

      const label = alpha[pos++];
      buffer.write(label,index++,'utf-8');
      buffer.writeFloatLE(x.value,index);
      index+=4

      set_mapping(label,x.name,header_map);

      if (pos>=20) break;
    }

    const payload = buffer.toString('base64',0,index);

    let headers = [];

    for (const x of header_map){
      headers.push(JSON.stringify(x));
    }

    let output={};
    output.payload=payload;
    output.headers=headers;

    return output;

  }


  module.exports = {set_data,get_data};
```

2. Node.js SWARM connection (client) class

```
'use strict'

const internal = {};
const CircularBuffer = require('circular-buffer');
const asyncLock = require('async-lock');
const {
  Telnet
} = require('telnet-client');

const StringDecoder = require('string_decoder').StringDecoder;
const decoder = new StringDecoder('ascii');
const influx = require('@influxdata/influxdb-client');
const InfluxDB = require("@influxdata/influxdb-client").InfluxDB;
const Point = require("@influxdata/influxdb-client").Point;

const magicWord = 'swarm';
const influxDb = new InfluxDB({
  url: `http://127.0.0.1:8086`,
  token: magicWord,
  timeout: 60 * 1000
});
```

```
const influxWriteApi = influxDb.getWriteApi(magicWord, magicWord);

const payloadMethods = require('../common/payload_methods.js');

const params = {
  host: '192.168.4.1',
  port: 23,
  shellPrompt: null,
  timeout: 20000,
  encoding: 'binary'
}

module.exports = internal.swarmConnection = class {


  constructor() {

    this._lock = new asyncLock();
    this._last_headers = '';
    this._last_headers_published = new Date();

    this._lon = null;
    this._lat = null;

    this._volts = null;
    this._temp = null;

    this._response = '’;
    this._can_transmit = false;

    this._msg_queue = new CircularBuffer(20);
    this._connection = new Telnet();

    this._connection.on('timeout', () => {
      //this._connection.end();
      throw ('lost telnet connection with swarm modem');
    })

    this._connection.on('close', () => {})

    this._connection.on('data', d => {


        const payload = d.toString('utf-8');
        const chunks = payload.split('\n');
        for (let x of chunks) {

          if (!x) continue;

          if (x[0] == '$') {

            const msg = JSON.parse(JSON.stringify(this._response));
            console.log('rx: '+msg);
            this.response_handler(msg);
            this._response = x;

          } else {
            this._response += x;
          }

        }


      if (!this._can_transmit) return;

      this._lock.acquire('key', () => {
        try {
          const msg = this._msg_queue.pop();
          console.log('tx: '+msg);
          if (msg) {
            this._connection.write(msg);
          }

        } catch (err) {};
      });
    })


    this._connection.connect(params);

    this.write_modem('$PW 30');

    setInterval(() => {
      this.write_modem('$MT C=U');
    }, 30000);

  }

  get_meta() {
    let meta = {};

    meta.lat = this._lat;
    meta.lon = this._lon;
    meta.volts = this._volts;
    meta.temp = this._temp;
    const t = new Date();

    meta.unix_time = String(Number(t.getTime()) / 1000.);

    return meta;
  }

  async upload_msg(msg) {

    if (!this._can_transmit) return null;

    const p = payloadMethods.set_data(msg, this.get_meta());

    if (!p.headers || !p.payload) return null;

    const t = new Date();

    if (JSON.stringify(p.headers) === JSON.stringify(this._last_headers)) {
      this.write_modem('$TD "' + p.payload+'"');
    } else {

      for (const x of p.headers)
        this.write_modem('$TD "' + x+'"');

      this.write_modem('$TD "' + p.payload+'"');
      this._last_headers_published = t;

      this._last_headers = p.headers;
    }


    if (Number(t.getTime() - this._last_headers_published.getTime()) /
      1000. > 24 * 3600) {

      for (const x of p.headers)
        this.write_modem('$TD "' + x+'"');

      this._last_headers_published = t;
    }
    return p;
```

```
}

async write_modem(msg) {

  const cs = this.get_chksum(msg);
  msg = msg + '*' + cs + '\r\n';
  this._lock.acquire('key', () => {
    try {
      this._msg_queue.push(msg);
    } catch (err) {};
  });

}

async write_db(field, value) {

  try {
    const p = new Point('swarm_meta')
      .tag('qty', field)
      .floatField('value', value);

    influxWriteApi.writePoint(p);
  } catch (err) {

    console.log(err);
  }

}

get_chksum(msg) {
  const buf = Buffer.from(msg);
  let cs = 0;

  for (let v = 1; v < msg.length; v++)
    cs ^= buf[v];

  return cs.toString(16);
}

async get_mt(a) {

  const b = a[1].split('*');
  await this.write_db('unsent', b[0]);

}
```

```
get_gn(a) {

  // example
  // $GN 29.6363,-95.6280,28,274,1*05

  const b = a[1].split(',');

  this._lat = b[0];
  this._lon = b[1];

  this._can_transmit = true;

}

async get_pw(a) {


  // example
  // $PW 3.28700,0.00000,0.00000,0.00000,28.0*3d

  const b = a[1].split(',');

  this._volts = b[0];

  await this.write_db('volts', b[0]);
  const c = b[4].split('*');
  this._temp = c[0];
  await this.write_db('temp', c[0]);

}

async get_rt(a) {

  // example
  // $RT RSSI=-88*2d

  const b = a[1].split('=');
  const c = b[1].split('*');
  await this.write_db('rssi', c[0]);
}


response_handler(msg) {

  if (msg.length < 3)
    return;
    const a = msg.split(' ');

    if (a.length<2)
      return;

    if (a[1].includes('OK')) return;

    if (a[0].includes('$GN')) this.get_gn(a);
    else if (a[0].includes('$RT')) this.get_rt(a);
    else if (a[0].includes('$PW')) this.get_pw(a);
    else if (a[0].includes('$MT')) this.get_mt(a);
    else return;

  }
}
```

3. Node.js SWARM data (server) function

```
'use strict'


const axios = require('axios');
const hiveBaseURL = 'https://bumblebee.hive.swarm.space/hive';

const influx = require('@influxdata/influxdb-client');
const InfluxDB = require("@influxdata/influxdb-client").InfluxDB;
const Point = require("@influxdata/influxdb-client").Point;

const magicWord = 'swarm';
const influxDb = new InfluxDB({
  url: `http://127.0.0.1:8086`,
  token: magicWord,
  timeout: 60 * 1000
});

const influxWriteApi = influxDb.getWriteApi(magicWord, magicWord);
const payloadMethods = require('../common/payload_methods.js');

let headers = null;

async function getSwarmData() {

  const body = {
    'username': '',
    'password': ''
  }

  let url = hiveBaseURL + '/login'

  const result = await axios.post(url, body, {
    headers: {
      'Content-Type': 'application/json',
    },
  });


  const config = {
    headers: {
      Authorization: `Bearer ${result.data.token}`
    },
  };

  url = hiveBaseURL + '/api/v1/messages'

  const output = await axios.get(url, config);

  for (const x of output.data) {
    const buffer = Buffer.from(x.data, 'base64');
    const d = buffer.toString('utf8');

    if (d[0] == '{' && d[d.length - 1] == '}') {

      if (!headers) headers = {};
      headers = Object.assign(headers, JSON.parse(d));
    } else {

      if (!headers) continue;
      const out = payloadMethods.get_data(d, headers);

      for (let key in out) {
        await write_db(key, out[key]);
      }
    }
  }
}
```