Flood Relay

Angella Mwangale — Sam Collet

July 2019

1 Abstract

In 2018 it was estimated some 200,000[6] Kenyans were displaced by rainwater floods, and the resulting mudslides. The coastal and river areas of Kenya are prone to flooding every year. This includes the rural villages and towns, but is a significant problem in metropolitan areas like Mombasa and Nairobi[4]. A lack of preparedness and infrastructure has only exacerbated this disaster.

In this project we present The Flood Relay Network (FRN). The FRN is an accessible, low-cost, high-efficiency network of devices that provides actionable data for local disaster preparedness efforts.

2 Project Overview

The project consists of hardware and free and open-source software(FOSS) components. Due to our two-week time constraint we were unable to engineer the hardware elements, thus our code has not been battle tested in a real scenario. Instead we illustrate a plan for implementation in highly affected areas, a grid architecture that fits our limitations, a low-barrier-to-entry device for collecting weather data, and performant software to maximize data throughput.

Our device consists of a LoRa[2] radio transmitter that can send and receive data on a 900mhz frequency, mounted to a Raspberry Pi. The Raspberry Pi is a single-board computer (SBC) and is sufficient for our needs as the cost is low and it supports several ancillary devices including: the LoRa radio, a flood stick or array of flood sticks, and a humidity/temperature sensor to name a few. Here is a listing of potential sensors[7].

The Raspberry Pi also runs on a linux operating system which can run our C code with support for posix multithreading. This makes it an ideal device for general development, but also allows engineers to optimize for the problem without having to write necessarily embedded software.

The device is meant to be mounted on a radio tower-like structure to extend its range and keep it above the affecting elements.

The FRN consists of these devices, but are not meant to be sources of data extraction. Instead, we rely on a central gateway with a dedicated power source and higher (ethernet wired) bandwidth.

3 Grid Planning

The network is not entirely self-sufficient. Although solutions towards that can be developed, it's not our intention to rely on the individual nodes for data extraction, but to instead aggregate the data from their sources, "pipe" the data back through the network to a central gateway, geolocate the data origin based on device signatures, and present a holistic data representation of the weather (for immediate analysis) and climate (for long-term study and executive action).

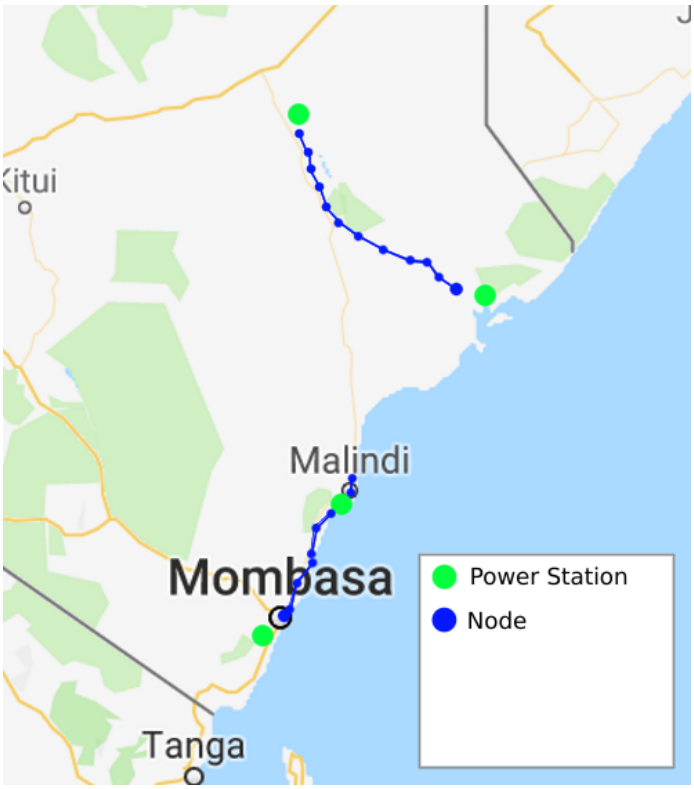

Figure 1: FloRelNet Grid Map

Displayed in Figure 1 is a map of two possible network routes accounting for Kenya's power grid[1], elevation, and the LoRa maximum LOS distance. For reference, the distance from Mombasa to Malindi is about 106km - as the crow flies. Using just six nodes, we can bridge the two cities along the coastline with each node responsible for covering about 18km of the flight journey.

4 Architecture

4.1 Hardware

4.1.1 Base Hardware

The Raspberry Pi[5] is a single-board computer running a special Linux distro, Raspbian. This makes development almost platform independent as software prototyping can be done on a laptop running similar versions of Linux and then airdrop the binary for hardware prototyping/deployment.

The LoRa radio module[2] is a long-rage radio transceiver generally operating at about 900mhz and capable of up to 20km(line-of-sight) broadcast distances. Increasing elevation improves the LOS distance. This can be used to our advantage by creating "beacon" nodes to be placed on taller hills to bypass uneven terrain interference.

The LoRa is connected to the RasPi's GPIO(General-Purpose Input/Output) pins and our program compiles with a C library to process these IO operations. There is plenty of OSS available to even develop a custom library wrapper for the IO operations, but we prefer to use pre-built and tested libraries.

Isolation of devices presents a major hurdle in terms of maintenance, observation, and most importantly, sustainability. Power is very difficult to provide, especially in remote or hard-to-reach areas. We suggest a solar power solution to provide continuous power for these always-on nodes.

4.1.2 Models

There are multiple approaches to how a "node" or "radio tower" on the network can be structured. Here we describe at least three, opting for the last one in its simplicity.

A Two-Module Split Radio System contains two devices. One is a dedicated radio receiver. One is a dedicated radio sender. Data collection can arbitrarily be split across the devices and bussed across the ethernet ports to dedicated threads.

A significant problem with this approach is timing the receiver to stop listening when the sender is active and vice-versa. It becomes a much more complex problem at that point, but allows for a much more sophisticated data collection system.

A Two-Module Single Radio System contains two devices as well. This time, only one radio transceiver is present for sending and receiving meaning the tick rate is entirely dependent on one process. The data collection, however,

is delegated to the second device. This frees up the ancillary device's ports for complete dedication to weather data collection.

A One-Module System contains a single device responsible for data collection, processing, and transmission. This is the device we optimized our program for because it's simple, straightforward, and runs on a single, easily managed process.

4.2 Software

The core conceit of the software is that is acts as a data processing pipeline. Our software is written in C for speed, but yields to unsafe memory if poorly implemented. Beyond this, we can achieve a fast and reliable, multi-threaded application to harness data from the device itself as well as its peers.

The first version of our software is built on top of two notable libraries: pigpio[3] and LoRa-RaspberryPi[8]

HOWEVER, you'll notice the LoRa library we picked is running on an sx1278 LoRa which broadcasts a smaller range at 137mhz to 525mhz. In this case, we chose to base our code on the simplicity and readability of this library while following the same setup and callback procedures as necessarily effective libraries (for sx1272 devices). The tutorial linked above offers a library with a suite of functions to manage receive and send states.

4.2.1 Code

Our program is designed to maximize throughput and take advantage of multithreading in a meaningful way. On initialization, we introduce four thread groups to the runtime.

Two of the threads describe our receiving and sending process. In the sx1278 model, we simply initiate a callback sequence when we've received some data. This is a limited-lifetime process as it has to exit without leaving a zombie process, whereas in the sx1272 model, we simply describe a singular loop. This loop captures both the sending and the receiving in one go and handles them at a pace described at the software level. More efficient approaches might set a fixed ring buffer for the send queue and wait until it's full to stop receiving and dump the queue. In our examples, we simply receive once, send once.

Networking is an essential part of the relay system. Without it, data extraction becomes impossible. We are constrained by the device limits but gain a boost from its long-range capabilities. In our implementation, we choose not to worry about the embedded software controlling the LoRa module because it is an expensive problem to worry about.

Conceptually, everything should be framed through two vectors:
Receive (here, a callback function),

```
1            void rx_callback(rxData *data) {
2              /*
3              rx->buf is the buffer we're
4              looking for;
5
6              we can strip other header
```

```
            information from the rx
            protocol, but we just want
            the buffer for this example

            we take the buffer and
            transform it into a data
            model we can manipulate;

            we convert to a result, strip
            the header, strip the message,
            and send it to the correct
            bucket so that bucket's
            thread can handle it;

            we don't care if the thread
            fails, we just want the
            message to go somewhere;

            and we stop when everything
            else stops;
            */

            int index;

            t_node *node;
            t_queue *queue;
            char *rx_buffer;

            t_header *header;
            t_result *result;
            t_message *message;

            rx_buffer = (char*)data->buf;
            if ((header = strip_header(&rx_buffer))) {
              node = (t_node*)data->userPtr;
              index = *((int*)get(node->neighbor_map, header->id)
);

              if ((queue = (t_queue*)get(node->receive_hash,
index))) {
                if ((message = strip_message(&rx_buffer))) {
                  result = new_result(header->id);
                  memcpy(&result->header, header, sizeof(t_header
));
                  memcpy(&result->message, message, sizeof(
t_message));
                  push_back(&queue, (void**)&result, sizeof(
t_result));
                }
              }
            }
          }

```

... and send (here, a dedicated loop),

```
        bool transmit_result(t_node *node, t_result *result) {
          (void)node;
```

```
3
4           if (result) {
5             #ifndef TESTING
6               // manually stoping RxCont mode
7               LoRa_stop_receive(&node->modem);
8
9               result->header.flags.transmission = 1;
10
11              // copy data we'll sent to buffer
12              memcpy(node->modem.tx.data.buf, result, sizeof(
    t_result));
13
14              LoRa_send(&node->modem);
15              sleep(1);
16              LoRa_receive(&node->modem);
17            #endif
18
19            return (true);
20          }
21
22          return (false);
23        }
24
```

Receive and send, conceptually, provide a base assumption with which to work. Portability in the future can be versioned and pre-processed to custom build binaries for any permutation of our node devices.

Efficiency gains are a focus point to keep a steady stream of reliable data entering and leaving the system without blocking up.

Multithreading allows us many opportunities to maximize on performance in parallel by making each thread group responsible for a fraction of the data's life time.

Here is an example of a thread initialization loop in where we index a hash table with any results we receive from our callback.

```
1               ...
2               bool initialize_receive_buffers(t_node *node) {
3             t_thread_watcher *watcher;
4
5             for (unsigned int i = 0; i < node->neighbor_count; i
    ++) {
6               if ((watcher = new_thread_watcher(node))) {
7                 if (!set(
8                   &node->receive_hash, i,
9                   (void**)watcher->results, sizeof(t_queue)
10                )) {
11                  printf("Failed to create results queue\n");
12                } else {
13                  if (pthread_create(
14                    &watcher->thread, NULL,
```

```
                        listen_for_data, watcher
                      )) {
                        printf("Pthread failed to create\n");
                      } else {
                        sleep(1);
                      }
                    }
                  }
                }

                return (true);
              }
                  ...

```

Here is where we monitor those results and, optionally, perform any transformations on the data that we need to before sending it to the global results queue.

```
                void *listen_for_data(void *arg) {
                  /*
                  And endless loop, registered to
                  the thread watcher's results queue;

                  we want to keep checking if the
                  receive callback is depositing
                  data so we can do interesting things
                  like immediately hand the data to the
                  send node or segfault when the data
                  is the wrong size;
                  */
                  t_item *data;
                  t_status *parent_status;
                  t_thread_watcher watcher;

                  if (arg) {
                    memcpy(&watcher, (t_thread_watcher*)arg, sizeof
    (t_thread_watcher));

                    while (watcher.status.running) {
                      if ((data = pop_front(watcher.results))) {
                        push_back(
                          &watcher.node->global_results,
                          &data->data, sizeof(t_result)
                        );
                        sleep(1);
                      }

                      parent_status = get_status(watcher.node);
                      if (parent_status)
                        if (parent_status->running == false)
                          watcher.status.running = false;
                    }
                  }

                  pthread_exit(0);
                  return (NULL);
                }
```

We decided, due to the limited bitrate and bandwidth, that the majority of data points don't need to be entirely precise to N decimals, but instead can describe a much less precise data point with an incredible efficiency gain. Introduce: float compression.

In our C programs we use a custom float with only 21 bits.

```
...
typedef struct s_float21 {
  unsigned int sign: 1;
  unsigned int value: 20;
} float21;
...
```

That is 10 bits for the fraction, 10 for the decimal, and 1 for the sign. This saves us space but may lose us time on all those conversions that have to happen in between.
We also happen to use 10 bits for the device ids meaning that at any point, a network can only have

$$2^{10} - 1$$

neighbors. This leads us to our next efficiency gain.

Instead of worrying about collision with indexing our neighbors, we can just use a fixed-sized hash table with

$$2^{10}$$

elements and index directly to the device's ID. Since we only need two of those tables, our data models end up being 2048 times the size of a struct pointer. Thankfully, modern hardware is not wanting, and in the most extreme of cases, we are likely to only have around 5-10 neighbors.

5 Complications

5.1 Adverse Effects

As of now, we are unaware of the side effects of having an always-on mesh netork of radio modules. Even more specifically, we are unaware of the adversity in regards to the fauna and the local population. It is entirely possible that the disruption of this technology to everyday life is enough to justify an incredibly careful and precise alignment of devices, but may preclude them from ever being introduced to begin with.

5.2 System Faults

Working with embedded systems is hard, especially when you have a thousand of them talking to each other. Immediate faults come to find and we wanted to discuss that here.

Data Speeds are a limiting factor when it comes to low-bandwidth, low-bitrate, wireless systems. Because the bitrate of our devices are determined by their frequency, we can perform a simple calculation to illustrate the inefficiency of our method of communication.

The bitrate is described by our broadcast frequency (in this example we'll use 900mhz) meaning every second, 900 bits should be transmitted or received. This of course does not account for dropped packets or corrupt data, or interference from other devices. Even in a perfect system, however, we see that 900 bits divided by the size of our truncated floats gives us a measly 42 floats(low-precision) every second. That is insanely low!

Among these problems, we can also factor in the distance. Even though the LoRa can reach distance of *up to* 20km LOS, we must consider bad visibility, elevation variance over long distances, weather, etc... before we can assume that this communication distance should be the default. Using the devices in a mesh network only compounds the possibility of waning signal strength because with each device we add on the path to data extraction increases our data entropy.

This means we have a higher chance of dropping packets and experiencing in-flight data corruption with each step added to the transmission process. Imagine the telephone game with radios and you have a prime example of why this can be an issue.

Thankfully, because the network acts like a mesh-net, we can most likely extrapolate a generalized model of the weather with many slightly inaccurate data points, or few greatly inaccurate data.

5.3 Wear and Tear

Weather and climate, the systems we are trying to monitor, are also adverse to the delicate, exposed hardware that is supposed to be continuously running.

Shielding and hardware protectors, offering little device exposure to the elements, are an apt way of protecting circuit boards and thin wiring. Heat sinks and UV redirection/capture can help with reducing the strain on the board itself.

Even though there are several inventive solutions available, mother nature is still relentless. The tower and the small devices themselves are subject to high wind speeds, fast-moving water or mudslides. Significant tectonic movement, rain, and wildlife interaction might all possibly degrade the device over time and lead to increased maintenance costs.

5.4 Better Solutions

Here we present a few solutions that may end up being more efficient and cost-effective.

Satellite imagery and communications is a high consideration for data extraction and extrapolation.

A cell network is another possible solution but has its own problems in more rural areas. Not everyone may own a cell phone, and not everyone who owns a cell phone may have adequate coverage to maintain an always-on network of devices. Not to mention that the devices themselves work much better as transmitters and not as well as data collectors. Considering the devices' may not have the proper instruments to measure weather data or may not be as accurate as other dedicated devices, we would not want to rely on this solution for this geographical area.

References

[1] GENI. *Kenyan Electricity Grid*. URL: `http://www.geni.org/globalenergy/library/national_energy_grid/kenya/kenyannationalelectricitygrid.shtml`. (accessed: 07.27.2019).

[2] Cooking Hacks. *SX1272 LoRa Shield for Raspberry Pi - 900 / 915 MHz*. URL: `https://www.cooking-hacks.com/sx1272-lora-shield-for-raspberry-pi-900-mhz`. (accessed: 07.14.2019).

[3] joan2937. *pigpio*. URL: `https://github.com/joan2937/pigpio/tree/826ab960ede1576b445d914f8f5b7b4d41eda17d`. (accessed: 07.26.2019).

[4] Daily Nation. *Heavy rains flood Kenya's capital Nairobi*. URL: https://www.nation.co.ke/news/Heavy-downpour-floods-streets-of-Nairobi/1056-4342344-vm69dz/index.html. (accessed: 07.27.2019).

[5] Raspberry Pi. *Raspberry Pi*. URL: https://www.raspberrypi.org/. (accessed: 07.26.2019).

[6] reliefweb. *Kenya: Floods - Mar 2018*. URL: https://reliefweb.int/disaster/ff-2018-000030-ken. (accessed: 07.27.2019).

[7] Raspberry Pi Tutorials. *Raspberry Pi Tutorials*. URL: https://tutorials-raspberrypi.com/raspberry-pi-sensors-overview-50-important-components/. (accessed: 07.27.2019).

[8] YandievRuslan. *sx1278-LoRa-RaspberryPi*. URL: https://github.com/YandievRuslan/sx1278-LoRa-RaspberryPi. (accessed: 07.14.2019).