`greenerthumb` Design

`greenerthumb` is implemented with a subprogram approach. This allows each sensor or client to be implemented with the least extra code and the most reuse of tests. It also simplifies using different languages where they make sense. An example is that this facilitates using Python for the sensors where library support is excellent, Go for infrastructure where efficiency is important, and c++ for GUIs where OpenGL bindings are mature.

Some downsides of this architecture are working with the paths to the subprograms, remembering all the necessary subprograms for a task, and messaging. An activate.sh/deactivate.sh pair is provided to mitigate the first problem. The scripts create aliases to all programs such that they can be run from anywhere. The second problem is addressed by giving composite scripts for some useful combinations of subprograms. Finally, internal messaging is handled by passing JSON lines from STDIN to STDOUT. This allows subprograms to not have to understand the messages, just how to handle JSON

Subprograms are described in later sections. Some of the subprograms also fulfill requirements themselves. An example of this is the `process` subprograms. Several major composite programs are provided:

`run-air`

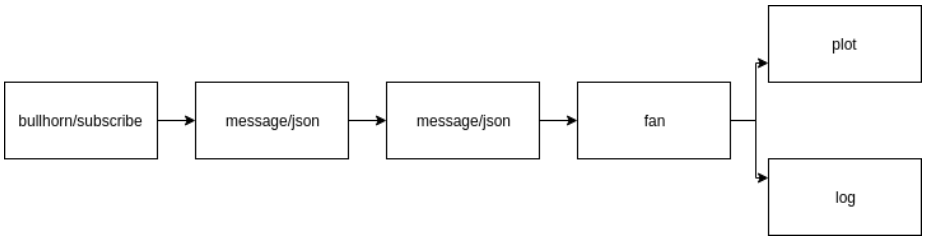
`run-soil`

`run-disclosure`

The disclosures are broadcasted on port 35053 by default. This is something all devices need to agree on by convention.

`run-logger`

`run-plotter`



`bullhorn` Design

`bullhorn` contains program pairs for networked communication. Included methods are pub/sub, broadcast, and listening.

pub/sub

pub/sub allows messages to be sent from publishers to subscribers.

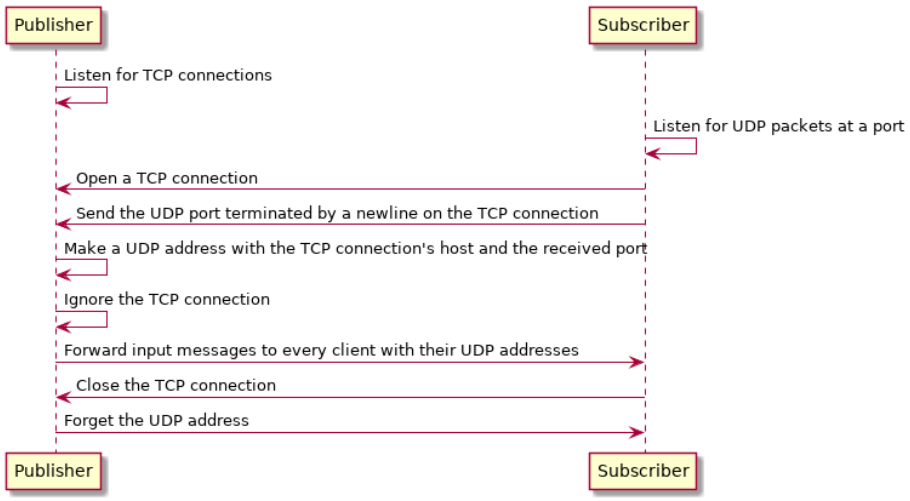


Figure 1: Sequence Diagram

This is done via a pub/sub system which operates over UDP with a TCP trunk. The TCP trunk allows the publisher to know when to stop publishing to subscribers and lets the subscriber know when it needs to try to reconnect to a publisher if reconnect is enabled. The unreliable UDP connection is fine because mostly periodic statuses are sent through the system.

The publisher will publish all newline-separated lines it receives over STDIN to every subscriber until STDIN is closed.

The subscribers print all newline-separated lines they receive from the publisher until the publisher is closed if reconnect isn't enabled. Subscribers never close if reconnect is enabled and will just periodically attempt reconnects. Subscribers will always exit with a failure to connect unless terminated because they will either try to reconnect forever or fail to connect to a terminated publisher.

broadcast

broadcast messages to all clients.

This is done via the broadcast address.

The server will send all newline-separated lines it receives over STDIN to every client until STDIN is closed. The clients print all newline-separated lines they receive until they are terminated.

Listening

Listening allows messages to be reliably sent from talkers to a listener.

This is done via TCP.

The clients will connect to the servers, write all their input from STDIN to the connections, and the server will echo the messages to STDOUT. Clients run until STDIN is closed or the connection is closed. The servers run until they're terminated.

`fan` Design

`fan` connects its STDIN to the STDINs of listed out-programs and STDOUTs from listed out-programs to STDINs of listed in-programs.

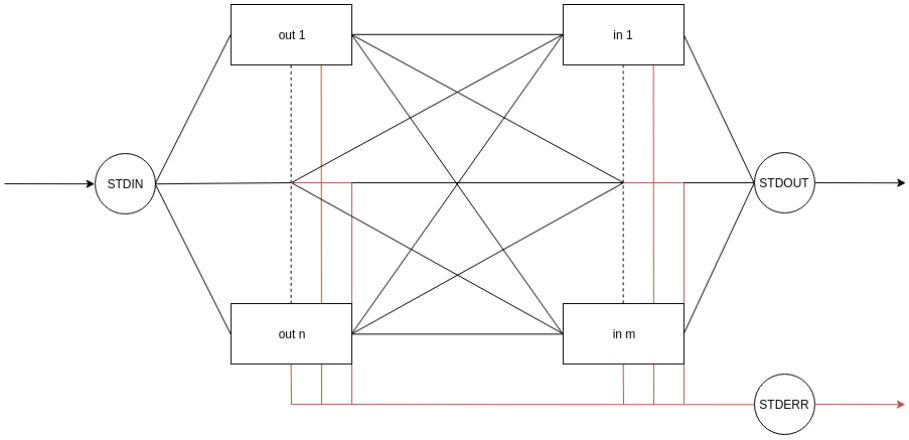


Figure 1: Diagram

Once data is written to `fan`'s STDIN, it gets written to the STDIN of all the in-programs. All of the in-programs' STDOUTs is copied into all of the out-programs' STDIN. All of the out-programs' STDOUT is then recombined and copied to `fan`'s STDOUT. Every program's STDERR is copied to `fan`'s STDERR. An in or out program failing just filters out that program and the rest of `fan` keeps running. `fan` exits once all of its programs exit.

`log` Design

`log` messages from STDIN to a file.

This just copies each line of STDIN to a file. Logs have the time appended to the file-name and are rotated each day.

`message` Design

`message` is where `greenerthumb` messages from the ICD are defined and the bytes-JSON conversion is implemented. Converters are provided for both directions. The converters run for continuous input instead of one message at a time so it is easier to pipe with other programs. Errors in input are ignored with a log so the converter can continue running.

`plot` Design

`plot`s `greenerthumb` JSON messages from STDIN.

UI

`plot` will convert input messages to points on a line-graph. The expected message format is:

```
{
  "Name": <message_name>,
  "Timestamp": <timestamp>,
  <name>: <value>,...
}
```

Messages don't necessarily need to be `greenerthumb` messages. They just need to fit this format.

Each non-ID and non-timestamp field will become a line in the graph. The name of the line will be determined by concatenating the `<message_name>` and the field's `<name>`. The line's will be assigned unique random colors which will be displayed in a legend with the message name's on the right side of the plot.

Each line will be overlayed to allow trend comparison. To do this, each line will have units normalized to eachother. The x-axis will have units of hours scaled to the period of all the received messages. Ranges of units are presented in the legend to account for the normalization.

If received messages have the same timestamp, the newest message will overwrite the older messages.

A save button makes screenshots.

`plot` only closes once commanded to close instead of once STDIN is closed.

Performance

`plot` is expected to be able to render 2-weeks worth of 5 kinds of data at a sample rate of 1 instance per second in less than 1 second per frame.

`process` Design

`process greenerthumb` data.

Programs

All programs accept `greenerthumb` ICD JSON messages from STDIN and report results to STDOUT. Each program terminates with a message printed to STDERR if any JSON message is malformed.

`summarize`

`summarize` reads all input until STDIN is closed and then reports a 5-number-summary for each data-type along with how many instances of that data-type were included.

`flatten`

`flatten` smooths data by keeping a sliding window of 3 instances of a data-type and replacing it with a weighted average of the 3 instances biased towards the middle instance. The first instance and last instance have a copy of themselves used as the instance to the left and right of them.

The left and right values are weighted by 1/6 each while the middle value is weighted 2/3.

`filter`

`filter` instances of data-types by specifying a list of ANDing conditions in the set of less than or equal to, less than, equal, greater than, and greater than or equal to and filtering STDIN according to the conditions.

An epsilon value for comparisons can also optionally be passed. The system epsilon is used otherwise.

`clean`

`clean` reads all input until STDIN is closed and filters instances that are more than a passed number of standard deviations away from the mean.

`select`

`select` messages from STDIN with names in an included set.

`sense` Design

`sense` programs write `greenerthumb` JSON messages from sensors to STDOUT. These can be `fan`ned into `message/bytes` piped into `bullhorn/publish`.

Sensors

`air`

`air` senses the 'Air Status Message' at 0.1 hertz.

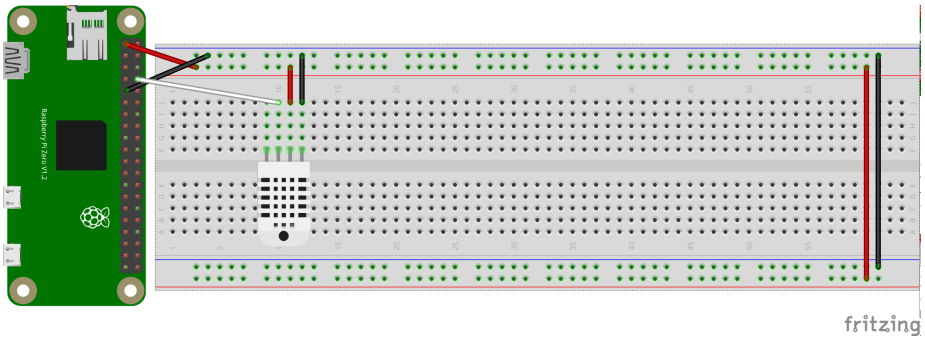


Figure 1: Air Schematic

`soil`

`soil` senses the 'Soil Status Message' at 0.1 hertz.

Emulators

Emulators are provided for all programs and each accepts an optional rate flag.

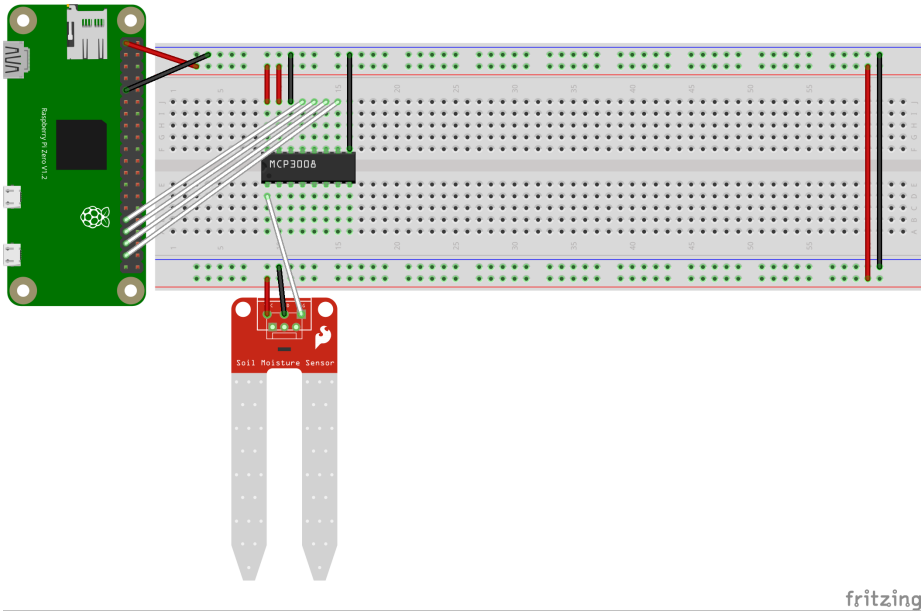


Figure 2: Soil Schematic

`store` Design

`store` messages with unique names.

`read` will read messages from a store and `write` will write unique messages to a store updating repeats.

`read` will write all messages in the store to STDOUT. `write` will accept all messages from STDIN and put them in the store. Duplicated messages are detected based on the name.

`disclosure` Design

`disclosure` prints the disclosure message with the passed values periodically at the given rate in hertz.

The default rate is 5 hertz.

`app` Design

`app` shows devices on the network.

The major portion of the work the app does is deserializing messages from the network. To accomodate this, a layered approach is taken where base receivers are defined for the different protocols and converters and converting receivers can get the received data all the way to the correct message with high code reuse. Higher level receivers can share lower level receivers to improve resource utilization. Testing is also simplified because most of the testing can occur at the highest level of data. Each receiver only receives one kind of message. This allows models to modularly choose which receivers they need and further simplifies testing.

One con of the layered approach is that it's difficult to construct the actual instances used for the app. Factories are provided in the app package to facilitate the full construction.

The mock, network, and app packages aren't tested. The mock package isn't tested because the gains would be marginal. The network package would be difficult to test and can be reasonably verified just by running the app since all its receivers are immutable and don't have branches. The app package is difficult to test because it uses lots of real resources. The Android app itself isn't tested because it is difficult, even though it probably should be.