# GRASP (Generalizable, Remote, Actuation & Sensing of a Process)

# Project Report

Tanner DeKock, Lam Nguyen, Aditya Tyagi, David Vakshlyak
tdekock@berkeley.edu, lamlnguyen@berkeley.edu, adi.tyagi@berkeley.edu, david.vakshlyak@berkeley.edu

## INTRODUCTION

The overarching project goal was to create a system that allows for the remote sensing, actuation, and control of processes that is generalizable across application areas.

More specifically, this translated into the following concrete sub-goals:

- Designing communication interface between process-level microcontrollers and end-user via MQTT over WiFi, internet cloud servers, etc.
- Implementing feedback control at the process-level to maintain a desired property of a system that is set remotely
- Making the architecture unified (i.e. compatible across diverse application areas)
- Creating a viable UI that can work with all stated application areas.

## PROBLEMS ADDRESSED

GRASP addresses an important problem that will likely affect the home of the future: integration of diverse devices into one unified system. Currently, there are multiple thrusts being made into the area of smart technology: smart gardening, smart homes, smart transportation, etc. by companies such as Nest, Uber, Google Home, etc. GRASP is a proof-of-concept of a system that can integrate supervision & control of all these diverse devices into one simple cloud-based interface.

## INSPIRATION

Our inspiration for the project came from SCADA (Supervisory Control & Data Acquisition) which is a framework used to remotely supervise devices involved in industrial production processes at various factories. We reasoned that a similar approach could be used to create value in the area(s) mentioned above.

## COURSE CONNECTIONS

Our project made us think more deeply about the following topics:

- Sensing & Actuation (Calibration, etc.)
- Networking & Protocols (MQTT, HTTP)
- Feedback Control (2-Step/Bang-Bang Controller)
- Temporal Specifications (system latency, etc.)
- Cybersecurity (How to ensure system is accessed by appropriate individuals?)

## 1 DESIGN & IMPLEMENTATION

### 1.1 Networking

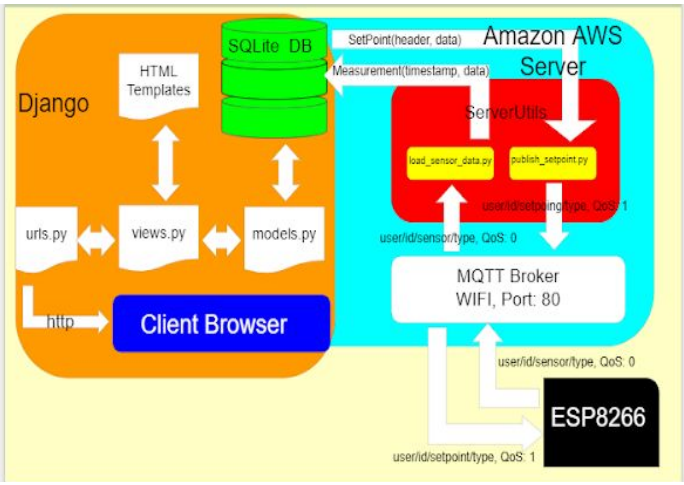



Figure 1 : **Overall System Architecture**

We used the free tier of Amazon Web Services Cloud Server to act as our central hub for sending and receiving

data. On our server, we have an MQTT broker running on port 80. MQTT is a machine-to-machine communication protocol that works off of a publish/subscribe model, with a central broker receiving data from publishing clients and sending data to subscribing clients based on topic. The broker was run on port 80 because it is the port usually used for HTTP, so it is usually not blocked. Data sent to devices and received by the server are of the following generalized JSON format:

```
{
  "device_id": Unique ID
  "device_type": Name of Application
  "data":  Array of Data_pts, for Data
  "setpoint": Array of Data_pts, for Setpoints
}
data_pt:
  {
    "d_label": Type of Data
    "d_value":
  }
```

Figure 2: **JSON Schema**

On the server, we have four scripts, in our ServerUtils folder, that handle all operations of data coming in and out over MQTT. The first two scripts, update_sensor_sub_list and update_setpoint_pub_list update the files in which we store the list of sensor topics and setpoint topics, respectively. The next script, publish_setpoint is run every time a user creates or updates a setpoint. It then publishes the user setpoint over the corresponding topic, with Qos 1 (Quality of Service) to ensure the setpoint is received by the device. Topics for each device are structured in the following way:

user/device_id/setpoint/device_type
user/device_id/sensor/device_type

The final script, load_sensor_data, is continuously run on our server. It subscribes to all sensor topics, and checks for new topics every five seconds. Upon receiving a message, if first verifies that the received json is of the correct format, verifies the setpoint sent back by the device matches the user defined setpoint decodes and parses the received json into our database. If there is mismatch between the user and device setpoints, the user setpoint is automatically published again.

As aforementioned, sensor data is continuously loaded into the SQLite database while the server runs. Within our Django project, models.py serves as our interface to the database. There we can enumerate fields and their types for the entries of the database. We have a Measurement model that takes a timestamp, type of data (represented by a string), and the actual float. Once we save these entries in our database via the load sensor script, we may call Python functions that abstract away the underlying language that queries the database.

## 1.2 User Interface

Django operates under an architectural pattern similar to Model-View-Controller. A templates folder contains the HTML files that act as the views in MVC. There the interface of the website is laid out and allows for data within a context to be passed from views.py. The views file acts as the controller in MVC. Here, functions are declared that render an HTTP response for each request sent to server. Functions include an index function that passes relevant data (latest sensor readings) to the relevant HTML file. There are also functions that generate graphs that show sensor data within the past hour when certain buttons are pressed. In the gardens webpage, buttons act as switches for the pumps and lights. When they are pressed, a views function runs that publishes the correct setpoint over the MQTT channel. In the water heater webpage, a Django form object allows for a validated integer input that is published to the heater when the submit button is pressed, redirecting the user to the same page.

## 1.3 Smart Gardening Application

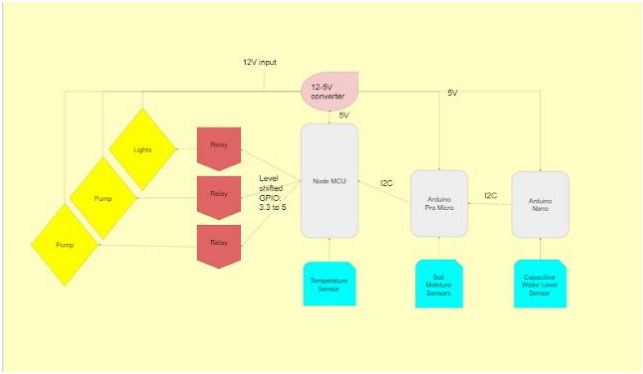



Figure 3: **Smart Gardening Wiring Layout**

The smart gardening application is powered off of a 12 volt AC adapter. The 12 volt line is connected through relays to each of the main actuators: 2 peristaltic water pumps and a set of LED grow lights. It is also connected to a 12 to 5 volt switching converter which was designed and manufactured in a different project. The 5 volt line powered the relays and the three different microcontrollers that were used: a NodeMCU, an ESP8266 based, WiFi capable microcontroller, an Arduino Nano, and an Arduino Pro Micro. The microcontrollers communicated on an I2C bus, with the NodeMCU being the master.

The NodeMCU was the main processing board, responsible for communicating with the central server,[3] collating data from the various other microcontrollers, packaging data in the JSON format,[4] and sending commands to the actuators based on received setpoints. It was also in charge of measuring the temperature. A library was used to set up the NodeMCU as an access point and display a landing page where a WiFi SSID and password could be set.[2] It was also planned that a site username or other means of authentication could be set from the device in order to differentiate devices between different users.

The Arduino Pro Micro was used to measure soil moisture from the two capacitive soil moisture sensors. The Arduino Nano was programmed to measure frequency.[1] We built in-house, a custom capacitive water level sensor and a 555-timer circuit used to convert the extremely low capacitance to a frequency. We created this in order to measure the water level in the water tank. Other means of measuring the capacitance of the water level sensor, such measuring an RC rise time, were ineffective due to the minuscule (on the order of 10s of picofarads) capacitance. The frequency was converted back into a capacitance, which was then mapped using an affine transformation to water level.

The application was split into a set of disparate tasks: soil moisture and water pumps for each plant, lights, temperature, and water level. Each task had a programmable interval at which data would be collated and published to the central server.[5] Setpoints would be set as quickly as possible, tangentially to the data publishing; a task that only updated every 5 seconds would still have its setpoint updated as soon as it arrived from the central server.

The soil moisture was designed to be in a feedback control system along with the pumps, but due to unavailability of the soil moisture sensors, the data was only published and the pumps had to be controlled manually.

### 1.4 **Smart Cooking (Water Heater) Application**:

At the plant level, our water heater consists of a relay-driven AC mains water heater, wired to a NodeMCU microcontroller, and a temperature sensor.

The temperature sensor feeds temperature data to the ESP8266, which decides whether to enable or disable the heater based on a remotely determined setpoint.

The feedback control mechanism is a simple bang-bang controller. The microcontroller turns the heater ON if the temperature is below the setpoint, and OFF if the temperature is above the SETPOINT. However, to avoid hysteresis, we added/subtracted 0.5 degrees from the setpoint to prevent repeated ON/OFF commands.

Some of the critical challenges we had to overcome related to accurately calibrating the temperature sensor, and properly wiring the water heater (insulation, etc.) to the relay to reduce electrical hazards.

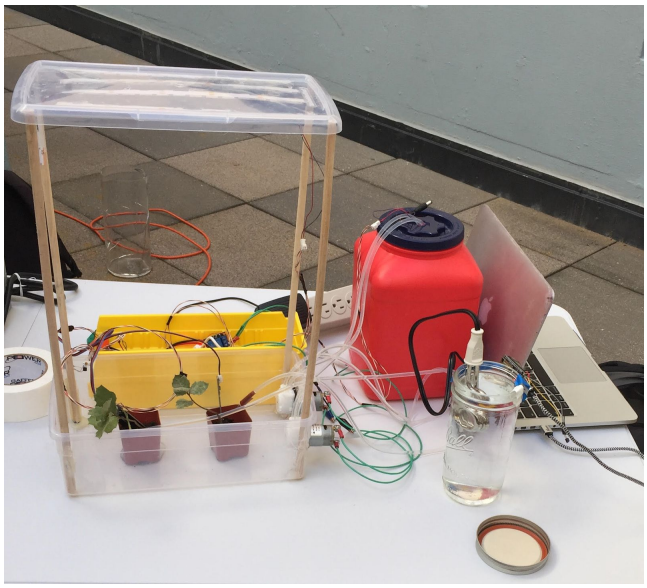
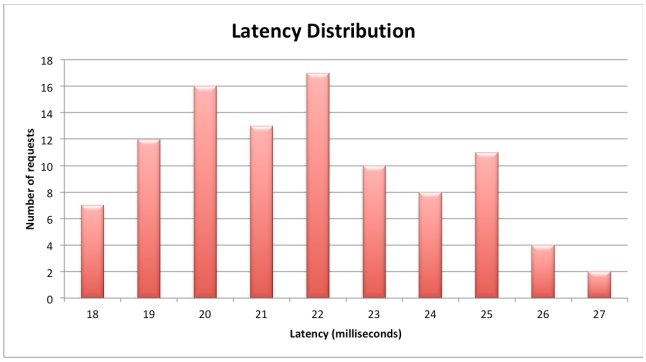


Figure 4: **GRASP during operation**

## 2 EVALUATION

The following figure shows the distribution of 100 requests to the broker running on our server.




= Figure 5: **Latency Distribution**

The data was generated by setting up a MQTT client such that it would publish on a given topic and then measure the time until it got a response back from the device. This script was run 100 times. The data seems to exhibit a normal distribution with mean of 21 milliseconds and a variance of a few milliseconds. Although the latency didn't cause any issues with our implementation as the concept scales, there may be need for a different communication protocol that could handle an increase in payload size.

## 3 DISCUSSION

### 3.1 Continuation Work on Project

If the project were to continue, one major feature that would need to be added is a timeout feature. Currently, our system will send setpoints, but if for some reason that setpoint is never reached, the target device will keep operating at the incorrect setpoint. A timeout feature would allow the server to see if the target device has failed to reach the setpoint in a certain amount of time, and then send a signal to kill the process so the user can inspect it.

Additionally, it would be useful to include some time synchronization between the server and each target device, to check the time of received data or set points, or to enable setpoints that are dependant on the time of the day.

Security is another area which can be improved upon, mainly by using TLS with MQTT for security between the devices and the server. It is also possible to have the website have a login function with the capability of having each user only able to view the data and control the sepoints for their own devices.

Sensing could be made more accurate. The 12-5 volt converter was a switching converter and therefore there was the possibility of voltage ripple. In addition to this, it is possible that the voltage may have dipped when under load. Thus, analog measurements may have been off. A separate ADC with a 3.3V supply could be sourced in order to take advantage of the low dropout regulator on the NodeMCU board.

Due to the timescale of the project and the fact that the electronic design was still in flux, it was not feasible to develop a PCB for the smart garden. Given more time, it would be useful to design and manufacture a PCB and an enclosure in order to prevent possible wiring issues and protect the electronics from the elements.

### 3.2 High Level Implications

If further developed, our project has several implications that could create transformative technological value:

- Enable a single interface for the user to control the suite of devices that will make up the smart home of the future with products from unaligned providers
- Facilitate sophisticated analytics & visualization of data by interested parties (eg. power-utility companies, etc.)
- If expanded further to the macro/societal scale, our system could be used by companies to integrate devices across application areas for a more comprehensive user experience:
    - Google using GRASP to integrate its Waymo (SmartCar) product with its GoogleHome offering,
    - Oil companies using GRASP to supervise & monitor extraction facilities, refineries, and pipeline network
    - Manufacturing companies using GRASP to supervise the entire supply chain (from resource acquisition, smart/fully-automated production, to logistics/ distribution & warehousing.

## REFERENCES

[1] https://github.com/PaulStoffregen/FreqCount
[2] https://github.com/tzapu/WiFiManager
[3] https://github.com/knolleary/pubsubclient
[4] https://arduinojson.org/
[5] https://github.com/esp8266/Arduino/tree/master/libraries/Ticker
[6] https://github.com/Julian/jsonschema
[7] https://pypi.org/project/paho-mqtt/

(*) All project files + code may be obtained at the following URL:
**https://github.com/adityagi1/eecs149-proj**