Team Members

Natalie Hess, Vamshi Teja, Erin Peacock, Jayashima Reddy, Rajesh Varagani, Dileep

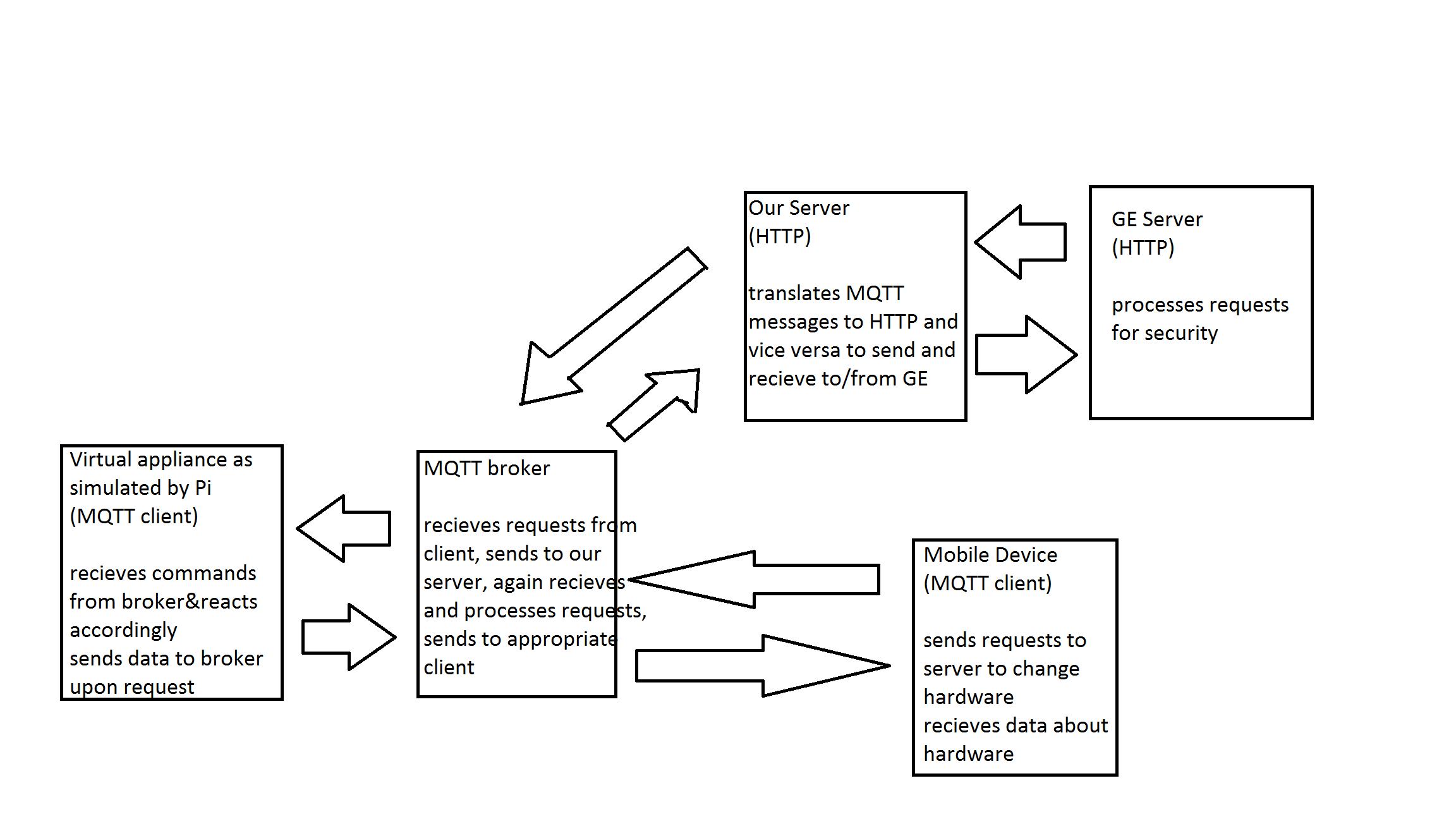
Virtual Appliance Simulator

GE Appliances

Abstract

Problem: Create a system in which a customer can easily communicate with and command their home appliances from their mobile device, but still have the GE server maintain full control to ensure security. Simulate this system using virtual appliances that can respond to inputs and send data the way a smart appliance would.

Solution:



* Virtual appliances are simulated using a Raspberry Pi 2 with wifi capability, switches, and LEDs
* User’s mobile device and our Raspberry Pi are both MQTT clients that subscribe to topics on the MQTT broker (the topics are each appliance: refrigerator, washer, oven)
* The MQTT broker code is written on a server created through Amazon Web Services with its own unique IP address; this code controls the logic of the hardware based on user input. It also collects data from the hardware to send to the user’s mobile device.
* We made a second server (also using AWS) to translate our MQTT messages into HTTP messages and vice versa in order to communicate with the GE server, which runs on HTTP
* Communication path: User sends a request from their mobile phone through an MQTT client app. The message goes to our MQTT broker, to our HTTP server to be translated to HTTP, is sent to GE’s server to be processed to ensure security, is sent back to our server and recoded into an MQTT message, sent back to the broker where it is processed as a command, and finally to the Raspberry Pi hardware, which reacts accordingly. The hardware also continuously sends status updates of the appliances through the broker to the user’s mobile device.

***Introduction:***

For a company to be successful, they have to be innovative in supplying customers with the newest and most sought after technology. Innovation and creativity is a huge part of any company, especially one like GE Appliances which draws in customers with its product’s new and unique features. One aspect of home appliances which many companies are currently looking into pursuing is connecting the physical appliance in a person’s home with one’s mobile phone. In today’s world, our phones rule almost every aspect of our lives. In order to make life easier for their customers, GE is proposing to remotely connect each customer’s appliances to his or her mobile device in order to accomplish trivial but annoying tasks with the push of a button. For example, if someone wants to have dinner ready when they come home from work, they can put a casserole in the oven that morning and then start preheating the oven from their phone before they head home. Or if someone is busy in a different room but wants to know the status of their clothes that they put in the washing machine some time ago, all they need to do is open their phone and check the status of the load. These seemingly simple features will help eliminate tedious and unnecessary tasks from consumers’ daily lives.

***Problem statement:***

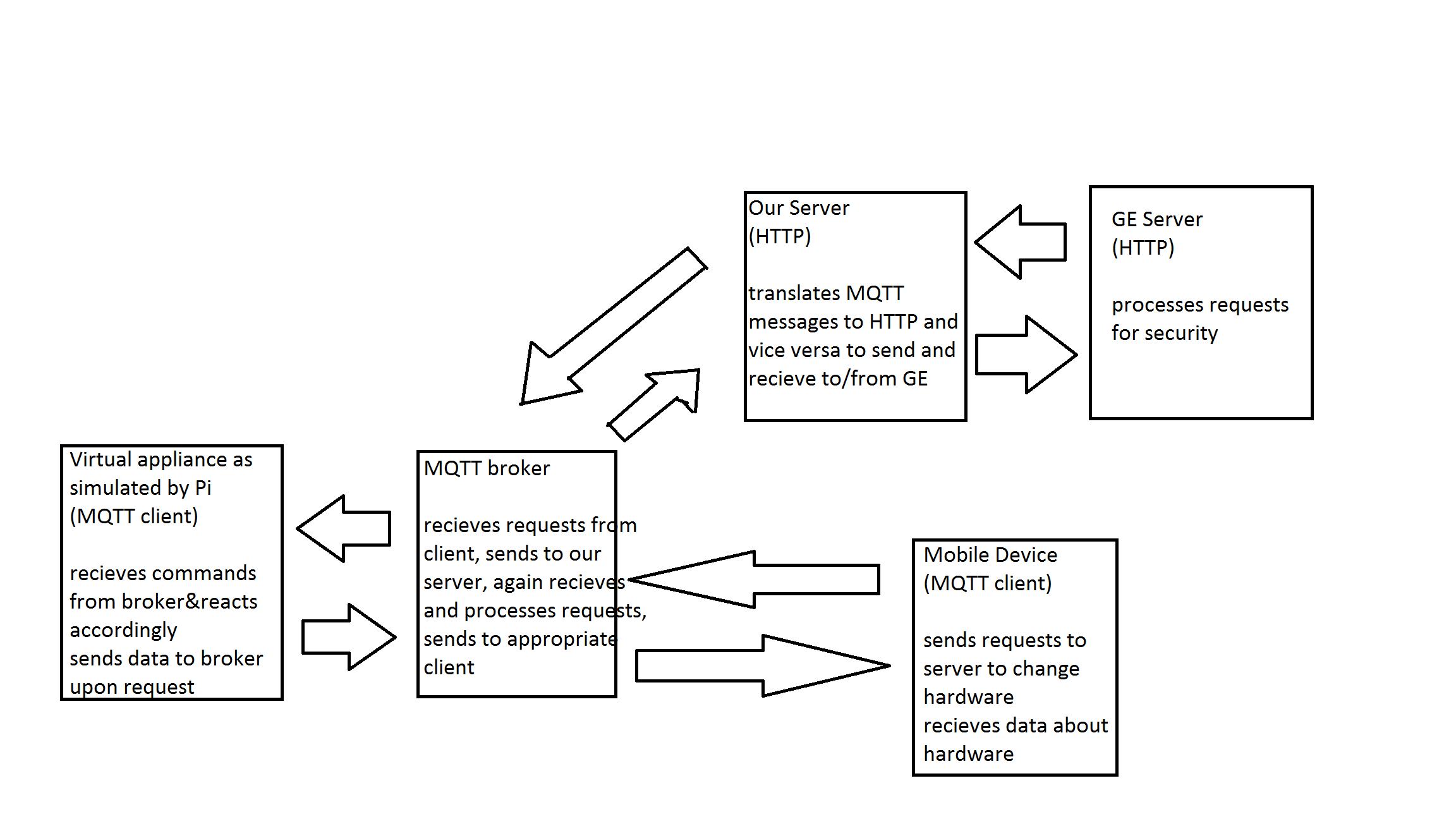
In order to accomplish this, our team was assigned to simulate how this mobile control will work. A user will send an input from an app on his or her mobile device. That input will then be sent to a server IP address, which we will create. For security reasons, GE wants to monitor all the requests to make sure the network is not hacked; our server will send all data to them, which they will process and then send back. Our server will take the appliance ID and message request, process it in an application that we create, and send a command to our “appliance”, which in this case is simulated using simple hardware. Our server will also take data from each “appliance” and send it back to the user’s mobile device as a status update. This should all happen over the cloud, with no need for a certain distance between the user and the appliance.

***Research:***

In order to come up with a solution, our team had to do a lot of research. Almost all the concepts were foreign to us at first, but after scouring articles, tutorials, and discussion boards, we gained what we thought was a good understanding of our task. First, we researched the basics of networking and Internet of Things concepts, which proved to be vast and at times very complicated. We started by googling basic questions and keywords, such as, “How do servers talk to each other?” and, “Raspberry Pi communication with web server.” From there, we used what we found out to move on to more specific topics such as MQTT vs. HTTP protocols, how to code Raspberry Pi applications in Python, how MQTT works as both a broker and a client, and how one would go about setting up a network of communication between a hardware device such as our Pi and a mobile phone. Once we understood the need for a server, we discovered Flask, a Python web framework that allows us to code applications based on user inputs. We all had to learn how to use that framework, and the UT students had to develop at least a basic understanding of the Python language as a whole by using online tutorials and practice labs. Hours of article reading and tutorial watching allowed us to take advantage of Amazon Web Services, which taught us how to create and configure servers to match our needs. We also needed to make our servers communicate with each other (and with the GE server) so we researched messaging and communication protocol as well as options to translate MQTT to HTTP and vice versa. A websocket seemed to be the best option, so we also had to teach ourselves what websockets are, how they communicate with multiple servers through multiple protocols, and how to implement one in Python. Most of the time spent on this project was spent on research so we would have the knowledge and background to properly implement our solution.

***Solution:***

This is a brief diagram of the problem and our proposed solution:



We decided to use MQTT (using Mosquitto as an open-source broker) as a messaging protocol to send the messages back and forth between servers and devices, because we believe it will be a great way to communicate back and forth without clogging up the servers with too much data. We have included several articles in our sources that explain in detail how MQTT client/broker protocol works and the advantages of using it compared to HTTP. For example, MQTT uses less bandwidth, supports longer battery life, and is able to send and receive messages faster than HTTP. It is an ideal service for sending and receiving data from a mobile device, which is the purpose of this project. Even though it was somewhat hard to implement and required more work, we believe that it will be a better solution in the long run.

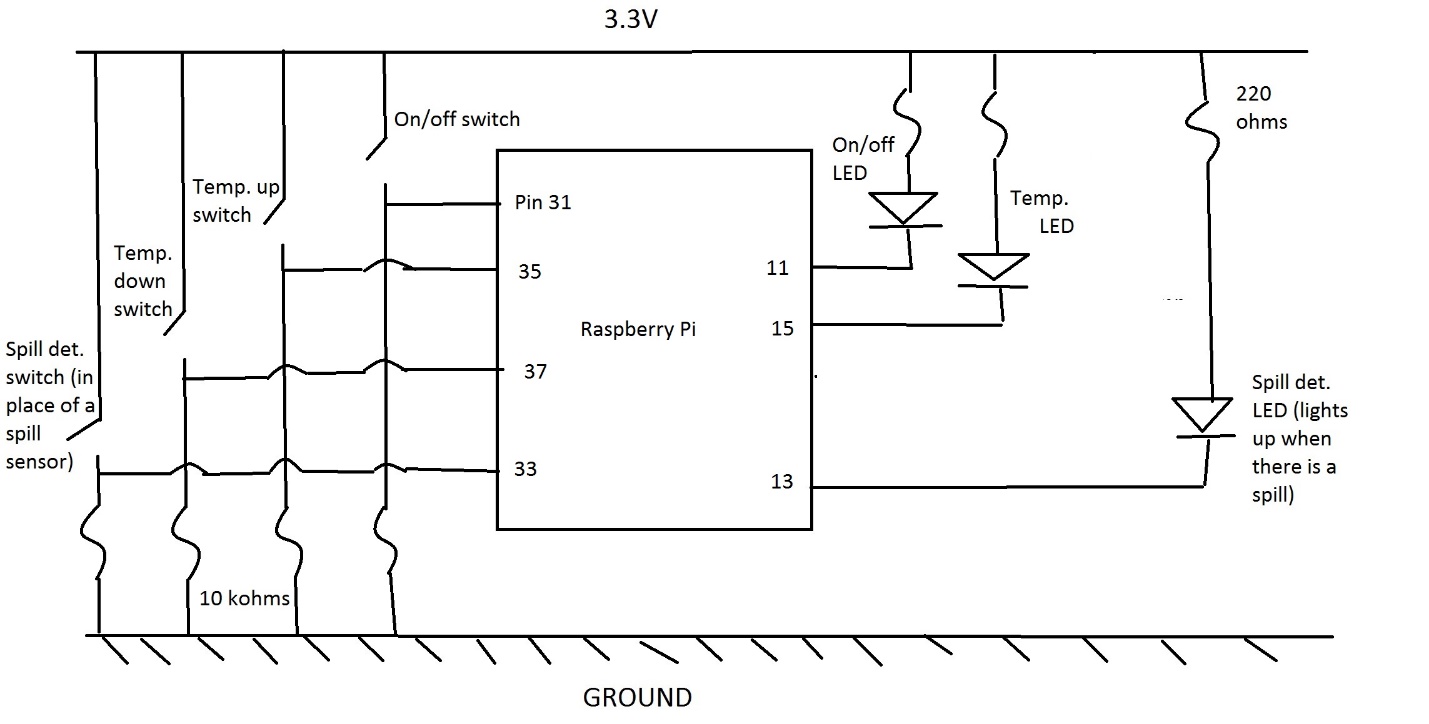
For our server, we set up an Amazon Web Services account and created two servers with their own unique IP addresses. Then, we used the Flask library in Python to code both our server and our MQTT client and broker code. The broker code is written in one server, while the other server acts as a translator for messages written in MQTT protocol and HTTP (which the GE server uses). MQTT has a client app available on both the Android and IOS app stores, so we downloaded and configured that app to send and receive messages to and from our MQTT server. Our servers are capable of communicating with any other servers, but we don’t have access to GE’s secure server until we give our final presentation.

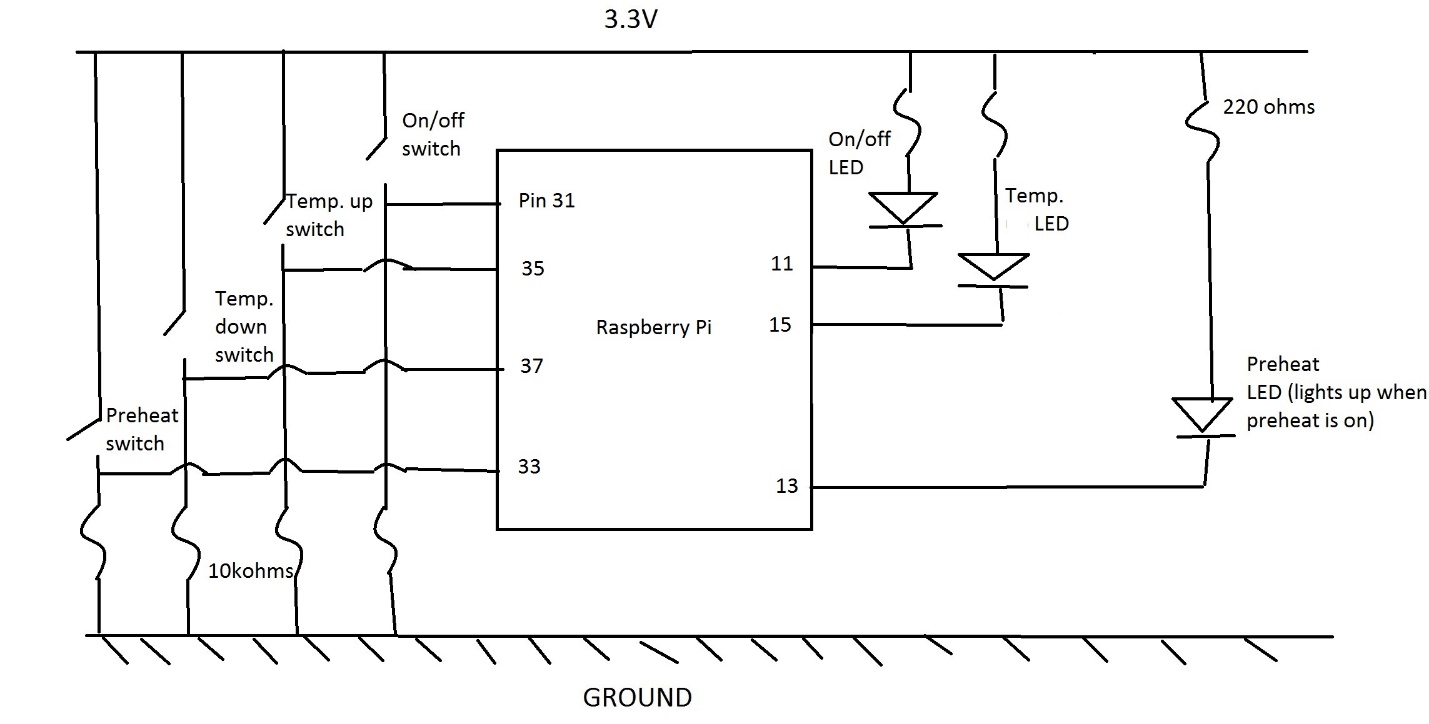
On to our next challenge: how to translate MQTT messages into HTTP messages and vice versa. This has proven to be a much more complicated issue than we originally thought. We researched ways to bridge HTTP and MQTT and eventually settled on using a websocket, a protocol that has multiple communication channels and can support any protocol over a single connection. Unfortunately, websockets are almost exclusively written in Javascript, which none of the team members have any experience in. Luckily, we were able to find Pywebsocket, an implementation of a websocket coded fully in Python. We installed Pywebsocket, then connected it to our MQTT broker code and HTTP server. Unfortunately, we were not able to fully implement our websocket in time for the presentation, but with a little more work it is possible to do.

Our virtual appliances are simulated using breadboards, LEDs, switches, and a Raspberry Pi. We also wanted to include temperature and spill sensors, but we did not have access to them, so we simulated that environmental data with manually pushing a button. Our Raspberry Pi runs using Raspbian and uses the MQTT client/broker code (coded in Python) to affect the hardware. When a user sends a message through the MQTT app connected to our server’s IP address, the input should be sent through our AWS server, which translates the MQTT message into an HTTP message, to the GE server and returned to our server to be processed into a command. This command would then be sent through our server to our Raspberry Pi to affect the hardware. One of the user commands is “Get Status”, which reads data from the hardware and displays it on the screen. (For example, getting the status from the refrigerator would tell the user whether the door is open or closed, what the temperature is, and whether any spills have been detected). We ran into problems with this plan, however, because IITH’s wifi block on MQTT prevented us from testing our MQTT code in time. When we were finally able to run our code, it did not fully work. We were able to demonstrate to GE that our server is capable of receiving inputs from a mobile device and affecting the Raspberry Pi, but some additional work would have to be done to implement the MQTT broker/client code.

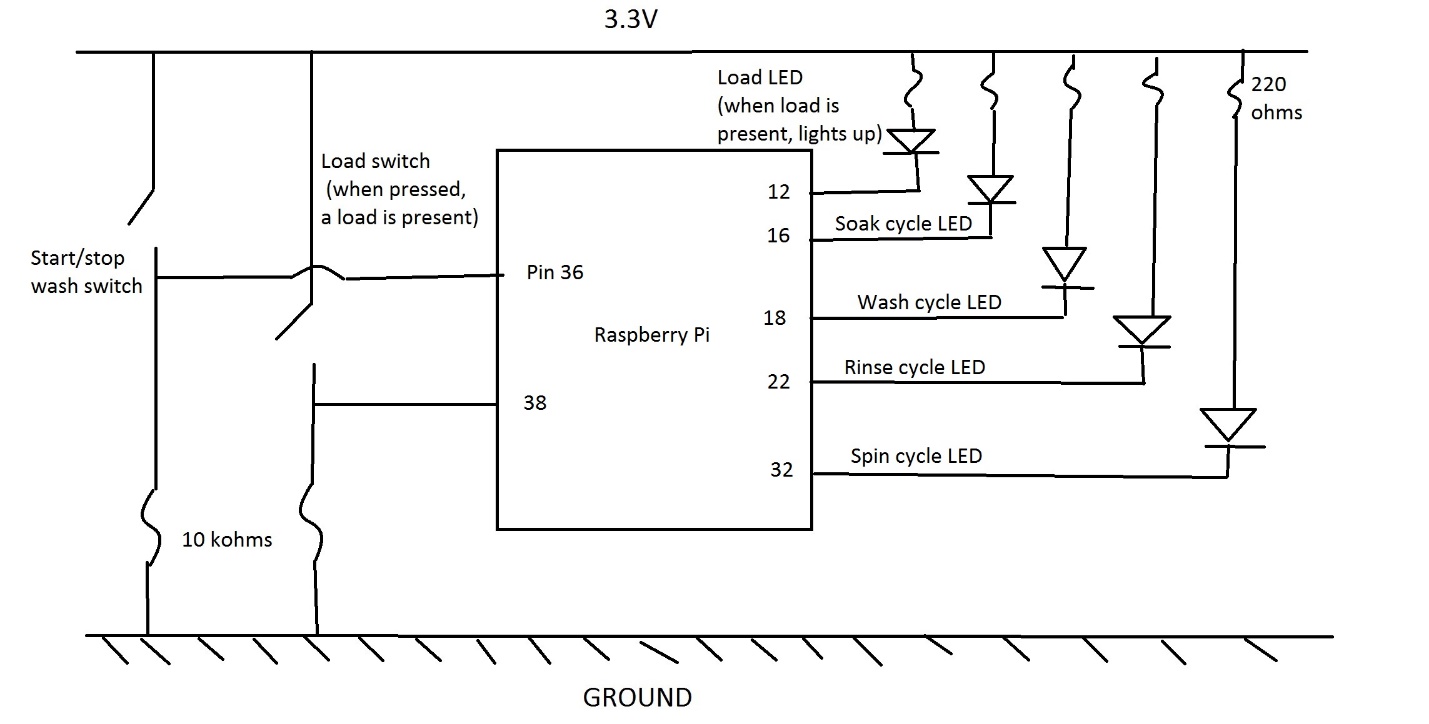
These are the circuit diagrams for our virtual appliances:

Refrigerator:



Oven:

Washer:



***Problems (and how we solved them):***

We definitely hit some snags on the way to our solution. We decided right away to use a Raspberry Pi to simulate our appliances, but had a hard time getting our hands on one with wifi capability. When we finally did get one by calling a connection in the city, we did not have an Ethernet cable, HDMI cable, SD card, or keyboard needed to set up the system. We finally got it set up, but it definitely slowed us down.

Our biggest obstacle throughout the whole project was an initial misunderstanding of the problem. We were under the impression that our own private server was not a necessity and we would just input our appliance code into GE’s server. By the time we were able to meet with our company contact and we found out that this was not the case, we had suffered a major delay. We quickly got to work researching and watching hours of tutorials on AWS and found out that the process is pretty complicated and confusing. We made a lot of progress in a very short time after that meeting, and now we definitely understand the importance of initial understanding and communication.

Another problem we had was learning how to use the Flask library in Python. The two UT students in our team didn’t know any Python whatsoever at the start of this project, and the IIT students only had a basic understanding. Once we found out that Python would be the smartest and most useful language for IOT implementation, we learned as much as we could from tutorials and practice online. We also asked Manish for guidance, and he helped us install and import Flask so we could write our Python programs.

We also didn’t decide on or know the ins and outs of MQTT until a fair way into the project, so we spent a lot of time researching and coding user interfaces for mobile apps that we never ended up using. For example, we have accounts and unfinished projects on IBM Bluemix, MIT App Inventor, and Microsoft Virtual Studio. Although this work was a good learning experience, it didn’t contribute anything solid to our final project solution, as we decided to use an MQTT app already on the market.

Once we did have our MQTT code, we realized that we cannot directly send MQTT messages to the GE server (which runs in http) due to the different protocols. This was an unexpected hiccup for us, but we decided on using a second web server to translate the MQTT message to an HTTP message before sending it to GE, and vice versa when receiving messages. We ran into some trouble trying to translate these messages, as our laptops were having trouble installing any MQTT-HTTP bridges or web sockets. We finally got Pywebsocket to download, and after that we configured it to our broker. The code for our websocket is not fully finished, but the basic idea and framework is there.

We discovered that IITH (for reasons unknown to us) has a wifi ban on MQTT. This was a huge problem for us, because we realized that we could not flash or run our MQTT code while using the IITH wifi. We emailed some of the IITH officials about overriding this restriction for us, but they never responded. We were able to use hotspots some of the time, but our code remained untested for much longer than we would have liked. By the time we were able to run our code, we didn’t have time to do the necessary debugging to make it work. We still have plenty of MQTT code written and in the GitHub, but it needs some more work and debugging before it can be flashed and fully working on our Pi.

***What we learned:***

This project has been an incredible (and challenging) learning opportunity for all of us. 8 weeks ago, most of us barely knew what a server was, much less how to code and communicate with one. Just understanding the problem statement was a big struggle initially, but it was definitely exciting to learn how networking and IOT really works. We learned how servers communicate with both each other and the outside world, how to send messages and commands to those servers using specific messaging protocols, and how to navigate Amazon Web Services and create/code our own server and applications.

Aside from the technical details, we also learned a lot about working in teams and with a company in general. Our initial misunderstanding about the problem was a huge setback, and now we realize how important it is to make sure we fully understand every aspect of what we’re supposed to do before we start. While it was difficult to schedule meetings around our traveling and academic commitments and our contact’s work schedule, we realize that we should have suggested a skype session or other alternative earlier into the project so we could have overcome our misunderstanding earlier.

We also realize that communication between team members is extremely important to coming up with a good solution. If someone doesn’t understand, they can’t fully contribute what is necessary to the group. Once we discovered that projects such as this involve much more individual learning and research than we thought, we realized we needed each other to fill the gaps in our understanding. Many of us had a preconceived notion that most of the concepts in the project would be like an extension of the classes we are taking/have already taken, but we were very far off. We had to learn how to be our own teachers for concepts that were originally way over our heads (and even learn a completely new programming language), and we are very proud of how much we’ve learned in such a short time.

***Future work on this project:***

This project, although useful, is merely a prototype for GE’s overall vision. The purpose of this solution is just to show that cloud communication between appliances and consumers is possible, not to perfect it. In order to actually implement and market this idea, GE will have to configure a network with much more data storage and security than our server. They will also need to produce actual smart appliances that will communicate with the servers the way that our simulated ones do. There is also the option for many more features that could be useful to customers, such as taking a picture of the inside of the refrigerator when it closes so one can see what they need to buy. This idea is far from over, but we hope we contributed something that can help further GE’s innovative mission.

***Sources:***

MQTT:

"MQTT Essentials." *Hivemq.org*. HiveMQ, n.d. Web. (multiple articles)

Holm, Kathleen. "Using MQTT Protocol Advantages Over HTTP in Mobile Application Development." *IBM DeveloperWorks*. IBM, n.d. Web.

"Mosquitto." *Mosquitto RSS*. Eclipse, n.d. Web.

Lewis, James. "MQTT Tutorial for Raspberry Pi, Arduino, and ESP8266." *Baldengineer.com*. N.p., n.d. Web.

Server creation and communication:

"AWS Tutorials." *Getting Started with AWS*. Amazon Web Services, n.d. Web. (multiple articles/videos)

Morgan, Jacob. "A Simple Explanation Of 'The Internet Of Things'" *Forbes*. Forbes Magazine, n.d. Web.

"Cloud Computing." *IBM Bluemix*. IBM, n.d. Web.

"Flask." *Full Stack Python*. Real Python, n.d. Web.

Harrington, Jonathan. "Displaying MQTT Messages in a Browser in Real Time." *Jonathan Harrington*. N.p., 02 Nov. 2013. Web.

"MQTT-HTTP Bridge." *Google Docs*. N.p., n.d. Web.

"Experiment with Websockets, MQTT (Mosquitto) Uses PyWebsocket." *GitHub*. Gist, n.d. Web. 2012.

"What Is WebSocket?" *Kaazing*. N.p., 2016. Web.

Applications:

"MIT App Inventor." *MIT App Inventor*. N.p., n.d. Web.

"Application Development." *Visual Studio*. Microsoft, n.d. Web.

Multiple pages on StackOverflow.com and Quora.com were also used when we googled specific questions