Doorbell Camera

Alex Mead, Kelly Peng, Daniel Rolandi

EECS 149/249A Final Report, Fall 2014

GitHub Repository: https://github.com/darolandi/cs149
Contains the iOS App code, Tornado Server code, Apache Server code, and a copy of mbed code.

mbed Repository: http://developer.mbed.org/users/doorbellcamera/code/DoorbellCamera/
Contains the mbed code.

Youtube video part 1: https://www.youtube.com/watch?v=aLYSg_7Tb7c
Youtube video part 2: https://www.youtube.com/watch?v=xibnRn-P710

Introduction

The Doorbell Camera is our take on connecting the doorbell to the Internet. It rings like a regular doorbell, but it also does something extra depending on whether the homeowner is at home or not.

If the homeowner is at home, it will display a message on an LCD screen. If the homeowner is not at home, it will send an email to the homeowner's email address.

One possible use case here is knowing that a package carrier has arrived at your home (when you are expecting a package and you are away from home). When the carrier pressed the doorbell, you will get an email notification (the caveat is you will not know who pressed the doorbell).

Overview

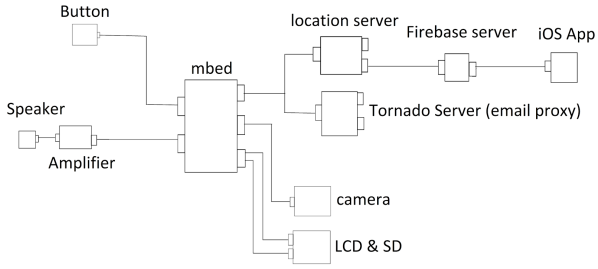


Figure 1: Overall System Map. Note: state machines for each component's operation can be seen in Append A.

The longest pipeline in our system is detecting whether the homeowner is at home. Using an iPhone app, the homeowner's location data is sent to a Firebase server, which is queried by the location server, which calculates whether the user is at home. The mbed can fetch the information from this location server.

The mbed simply sits waiting for a button press. A separate thread continuously polls the server every few seconds. On a button press, the mbed will check the 'home' variable and perform the behavior mentioned before.

If the homeowner is at home, the mbed tells the LCD screen to display a message. If the homeowner is not at home, the mbed contacts a proxy server and requests it send an email.

Component 1: mbed

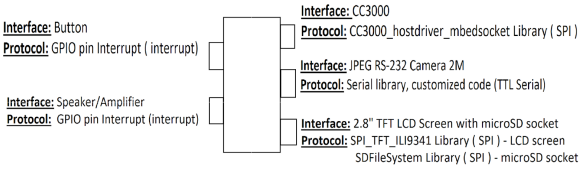


Figure 2: mbed

The mbed runs 2 threads: one called the "main thread" and the other called the "wifi thread."

1.1 Wifi and Concurrency

The wifi thread's job is to connect the wifi to the Internet (and reconnect if needed). It contacts the location server that has the "is homeowner at home?" data.

After parsing the HTTP Get Request HTML, the wifi thread then writes the data to a shared variable (called "`is_at_home`") that is read by the main thread if a button press is detected. It will always try to reconnect to wifi whenever service is interrupted for reliable system behavior.

We chose to do this concurrently so that the main thread needs to deal with only the state transition and making writes to the LCD screen and to a Websocket to talk with the Tornado email server.

This was essential because the button press triggers an ISR, and it is a bad idea because a scenario might happen where the ISR returns to a line right before sending a HTTP Get Request, causing the LCD screen or Websocket writes to be delayed until later. Separating state transition and WiFi connection/reconnection allows to minimize response time.

1.2 Button ISR

The main thread is the one that initializes variables and waits (by thrashing) until the button ISR gets called. The button ISR turns on the speaker and checks the "`is_at_home`" and issues a state transition for the main thread. The state transition is where the main thread issues a write on the LCD screen or on the WebSocket.

1.3 LCD Screen

The LCD screen was originally planned for showing the guest's photo when the guest presses the doorbell. The homeowner is at home, so the homeowner can check the guest's face (from indoors). There is an issue with showing a picture (to be discussed later in the "Challenges and Issues" section) so we ended up showing a predefined text instead. Using the SPI_TFT_ILI9341 library, mbed writes to the screen using SPI.

1.4 Sending Email

The WebSocket serves as a connection to the Tornado server, our email proxy server. The mbed writes a predefined key that will trigger the proxy server (which acts as an SMTP client) to send email. The proxy server handles the email sending.

The original, naive idea was that mbed has a library that can send an email. Unfortunately, mbed's SMTPClient library does not support the level of encryption required (TLS/SSL) by SMTP servers. We ended up using a proxy server. More on this in the "Issues" section.

The second button press turns off the speaker and clears the LCD screen. The speaker rings using an ISR, creating a 440 Hz square wave just like what we did in one of the labs, but due to use needs, an amplifier is used to increase the speaker volume.

Component 2: Location Server

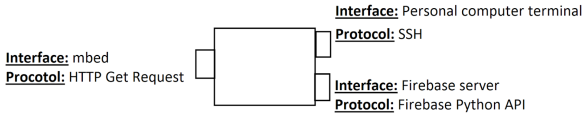


Figure 3: Apache Server

We have an Apache server with a Python script that runs continually, querying the Firebase Database for the homeowner's most recent location and calculating home and not home.

Using Python was a convenient choice since Firebase has a Python API for it, and since really many things are easier and more concise to write in Python than in C++.

The Python script stores a predefined "home" constant that determines whether the homeowner is at home. The location data (a latitude-longitude pair) is compared with the constant (this is done through a simple equation from geography); if the distance is smaller than a certain threshold the homeowner is considered "at home." Otherwise, the homeowner is "not at home." The Python script then writes 1 or 0 (respectively) to a text file called "home.txt" so the mbed can easily read it using a HTTP Get Request.

Component 3: Tornado Server (email proxy)

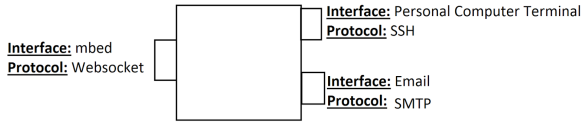


Figure 4: Tornado Server

We also have a Tornado Server written in Python that can accept communication using WebSocket. This server acts as a proxy to send an email on mbed's behalf.

Conveniently, Python has library support so that it can act as an SMTP client. The SMTP server we are using is Gmail's SMTP server (owned by Google). The content of the email and the from/to/subject headers are predefined in the Python script. We use TLS to encrypt the email, as required by Gmail's SMTP server.

The script simply waits around, waiting for a WebSocket connection and reads its message. The predefined key is "email" and so if the message is exactly that, we execute all the code to send the email.

In the GitHub repository you will find an index.html file that serves as our debugging spot.

Component 4: Firebase Database

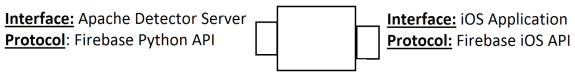


Figure 5: Firebase Database

Firebase is a database service that acts as a meeting point between the Apache server and the location-tracking iOS App. It has great API support to interface with Python (used by the Apache server) and iOS (used for the location-tracking app).

The database stores only the most recent latitude-longitude pair that represents the homeowner's location, since we never need the homeowner's trace or history of locations.

Component 5: iOS App

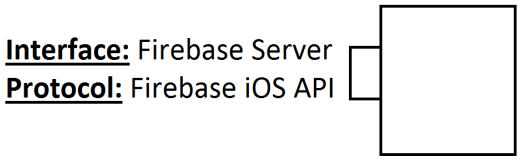


Figure 6: iOS App

Written in the recently-released Swift, this app reads GPS data (latitude-longitude pair) and uses the Firebase API to upload that to the Firebase data. Sending that data to Firebase requires a specific key, and that same key is known by the Apache Server so the Apache Server knows what to look for.

One thing that we failed to elaborate in the demo video or in the presentation was why used an emulator instead of a real iPhone. The reason is by using the emulator, we do not need to run across campus to test its home/not-at-home functionality. The emulator can emulate different locations in the world, so we can expedite the testing process.

One caveat needs to be addressed here. We model the homeowner being at home as simply his/her distance to a predefined point (even then, it is just his/her phone, which might be misplaced). This is a coarse-grained solution to locality, but might work well for our case since if a person is close enough to home (the threshold is fine-tuned to a few meters) then the person might as well be considered "at home." That is sufficient. The only problem here is probably the GPS's internal accuracy and error margins, which itself might be a few meters in magnitude. That is the level of fidelity that our model provides.

Component 6: Camera

Although not featured in the final system, we did get the camera working. The camera communicates with mbed using a buffered TTL Serial connection. The camera has a buffer within itself, so to give commands to it the mbed has to write bytes into that buffer. (Much alike how we send byte commands to WiiMote and iRobot in our labs.) Once captured, the image data is stored in the camera's buffer and mbed can fetch it incrementally. The image is encoded as JPEG.

Challenges and Issues

a. Sending Email

Originally we wanted mbed to be the SMTPClient (without a proxy server) but it is not compatible with the SMTP servers out there.

Antonio introduced us to Temboo, a cloud service that can serve as a proxy for us, but it accepts HTTP Post requests with "Content-Type: application/json" and our HTTPClient supports only "Content-Type: x-www-form-urlencoded", which is a completely different thing. Five HTTPClient libraries were investigated in total and none of them satisfied our needs.

So we turn to creating a proxy service ourselves, using Tornado. To connect, we use Websocket instead of HTTP Post.

SMS was originally discussed, but never investigated due to time constraints.

b. Displaying Picture on the LCD Screen

Originally we planned to snap a picture of the guest using the camera and display it on the screen.

The screen requires a 24-bit BMP image on its microSD card to display. If we wanted to show a preset image instead, we know how to do that and we can. Unfortunately, the picture we snap using the camera is of JPEG format and we would require a conversion.

We tried several libraries for conversion (the algorithm is too hard to implement ourselves, said the GSI) but none worked (or even compiled). The last one we tried, which was promising judging from the discussion threads, required us to put the JPEG image inside the microSD card and do the conversion using this external memory. We tried several filesystem libraries (SDFileSystem and SDHDFileSystem) and even with several different microSD cards, but they always fail to recognize the microSD cards. So we scrapped the picture idea and display text instead.

Final Remarks

This project helped us reinforce the ideas of several topics from class.

Reliable real-time behavior - We did our best to keep the system as reliable as possible. We ensured there's no possibility of deadlock, and the wifi always tries to reconnects if there is a connection failure.

Concurrency - We had to work with two threads in mbed to achieve a purpose that is naturally parallel. We did not employ mutexes because (1) only one thread is writing, so no possibility of overwriting exists, (2) the only one reading is an ISR, which is not allowed to have mutexes, and (3) in reality, the race condition should matter only when the homeowner is transitioning between home to not-at-home or vice versa, which means he/she is close enough to the doorbell anyway; recall the speaker always rings for these types of situations.

Design methodologies for embedded systems design - Each of our subsystems have their ways of unit testing. For example, the "`index.html`" in the Tornado server allows for testing sending the email. Then we have various test functions in the mbed's "`main.cpp`" for different components.

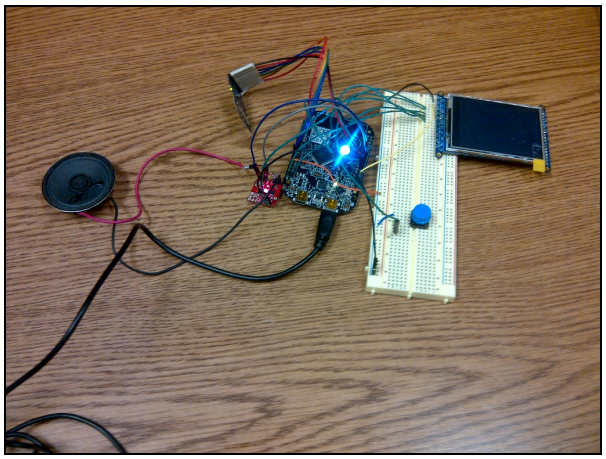


Figure 7: Final System

Credits:
mbed Official Library
Project Demo; by Antonio Iannopollo
CC3000 (wifi driver); by Martin Kojtal
HTTPClient; original by Donatien Garnier, latest fork by David Smart
Firebase (iOS-side); by Firebase.com
SPI_TFT_ILI9341; by Peter Drescher
Tornado (Python-side); by tornadoweb.org
Websocket (Python-side); by python.org
WebSocket; by Samuel Mokrani
Serial Interrupt Cookbook; by mbed
Camera: Link Sprite (Arduino)

*Big thank you to Antonio Iannopollo for advice and much patience

Appendix A: State Machines for System Component Functionality

For the exact protocol of communication between system components, see each component's corresponding section and figure in the main text.

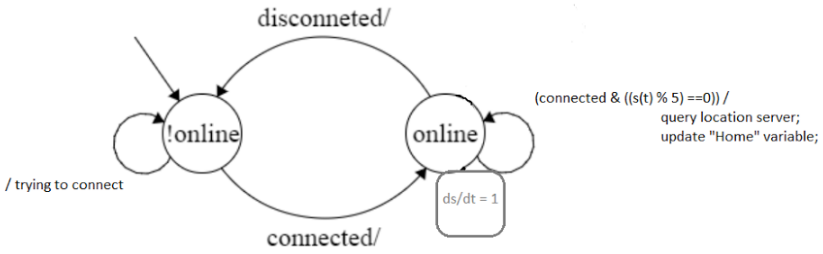
Component 1: mbed

mbed:

Variable: Home - Boolean

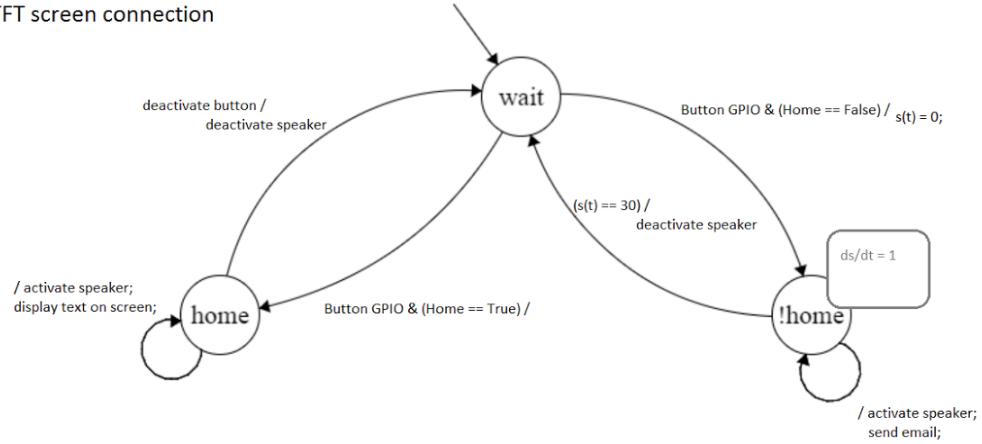
WiFi Thread:

In: Location Server query (HTTP Get Request)
Out:



Main Thread:

In: GPIO Pin (Button)
Out: GPIO Pin (Amplifer -> Speaker)
WebSocket (email)
TFT screen connection



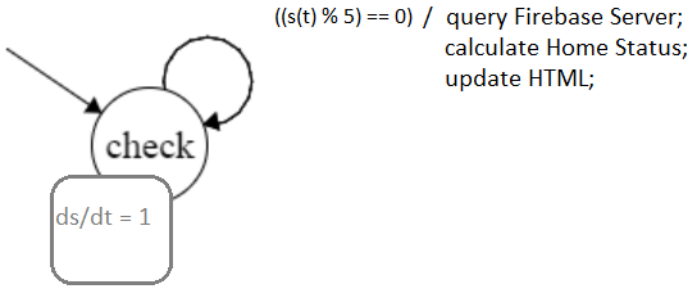
The two state machines which model the funcitonality of the mbed configured in an asynchronous fashion whose semantics consist of both machine executing independantly. The one shared variable between them is 'Home' a boolean variable. In cases of race condition, Main Thread will get read access over WiFi Thread's write access. Due to the physical nature of this problem, race conditions which actually change the value of Home simply mean that a user is near the threshold defined to be home or not home. Thus, they will most likely see if someone is at the door or hear the bell. Meaning, this race condition is of lesser importance that it would be in other shared variable

Component 2: Location Server

Location Server:

In: Firebase Python API

Out: /home.txt HTML

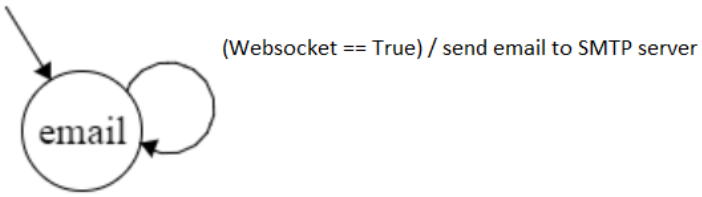


Component 3: Tornado Server (email proxy)

Tornado Server (email proxy):

In: Websocket from mbed

Out: email to SMTP server

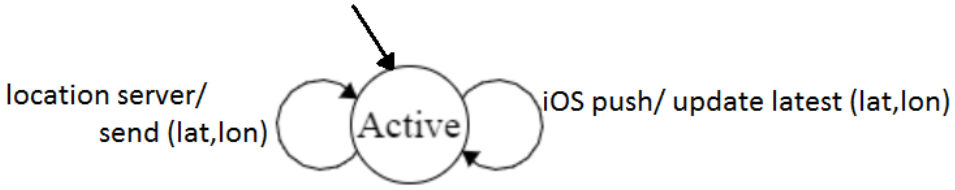


Component 4: Firebase Database

Firebase Server:

In: iOS App push, location server pull

Out: location server (lat,lon)



Component 5: iOS Application

iOS Application:

In: GPS Chip (lat,lon)

Out: Firebase API push