ECE 558 - Fall 2018
Final Project - FINAL Report

Brian Henson & Jamie Williams

Final Project: Security/Theft Detection App and Base Station

FINAL Report

Summary of whole-system features

Our system consists of both a Raspberry Pi base station and an Android app, and serves as a security and intruder detection system. There are two kinds of sensors supported: magnetic contact switches and vibration sensors. A user can connect any combination of these sensors, up to 20 total. The sensors can be added/removed through the user's app. When an intruder is detected, it will play an alarm tone through a standard headphone jack; a default alarm tone is included, or the user may record & upload a custom alarm sound through the app. It will also capture two photos when the alarm is triggered, a low-resolution preview image and a high-resolution detail image. The last thing that happens when an alarm is triggered is that a notification is sent to the user's phone, prompting them to open the app and see a photo of the intruder. The preview photo is automatically downloaded & displayed in the app; if the user likes what they see, they can request to download the high-resolution image and save it to their device (viewable through any standard photo gallery app).

Of course, the user has the ability to remotely arm and disarm the alarm system with the app. Other features include: the ability for the user to capture an image whenever they want, not simply during an alarm; when an alarm is triggered, the system automatically disarms itself to prevent spamming the user with dozens of notifications about the same event; and, the system is set up to alert the user with a different notification if the base station loses WiFi connection or power.

Lastly, the system supports multiple simultaneous app/pi pairs; the app has a login/logout function, and will only communicate to a base station that's logged in with the same credentials. The app also has an autologin feature that will remember the last credentials you used. (currently the base station is hardcoded to use one specific account)

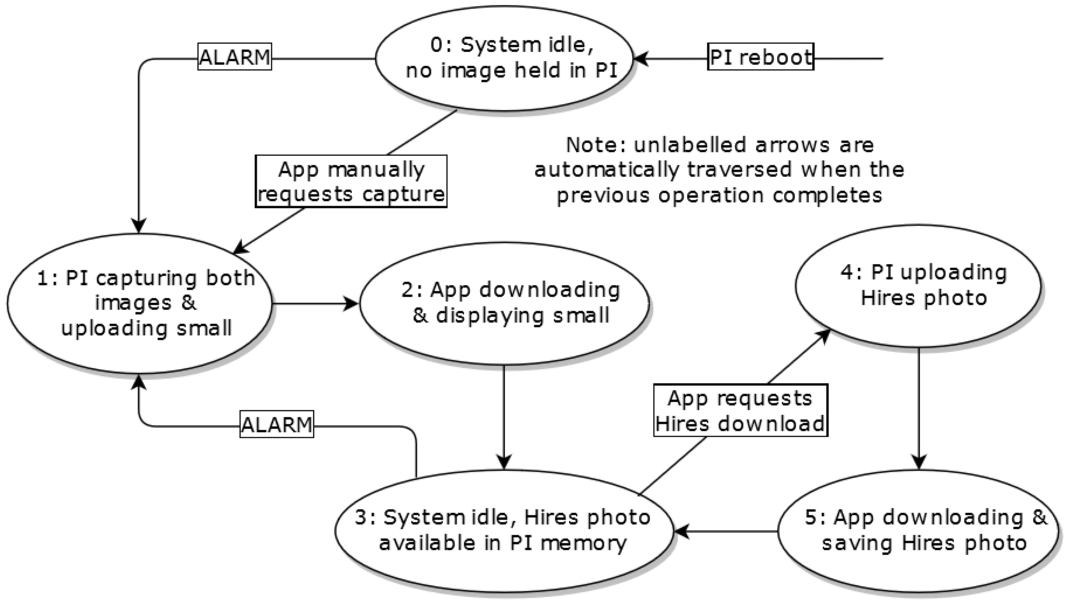
Description of architecture/how it works

Full-system architecture (communication)

First, it should be noted that the app and base station do not *directly* communicate with one another as we first envisioned. Instead, all their communication goes through the Firebase Realtime Database. Additionally, the notifications sent to the phone app are triggered by the base station, but actually sent by a set of Firebase Functions written in Node.js and running on Google's Firebase servers. The Firebase Messaging service is intended to be used for server-to-client notifications; client-to-client notifications are impossible, as near as we can tell. Because Firebase Functions can be triggered by conditions in the Realtime Database, that was the solution we chose. (NOTE: could create a simple flowchart showing app/pi/db/functions and what communicates to what…. But it's probably not worth it)

Since the photo upload pipeline is a bit complex, and shared between both the app and the base station, it will be explained here. We determined that we needed only 6 states to fully describe what each half (app/Pi) was supposed to be doing at any given point in its lifecycle, and used only a single database field (values 0-5) shared by the two to communicate in this regard. The app and the Pi are both designed to be re-entrant as well: if the app or the Pi unexpectedly shut down in the middle of a state, when they re-launch they will read the database, see what state the system is in, and resume doing whatever they're supposed to do for that state.

Photo pipeline state diagram:



0: System idle, no image held in PI
PI reboot
ALARM
App manually requests capture
Note: unlabelled arrows are automatically traversed when the previous operation completes
1: PI capturing both images & uploading small
2: App downloading & displaying small
4: PI uploading Hires photo
App requests Hires download
ALARM
3: System idle, Hires photo available in PI memory
5: App downloading & saving Hires photo

How it works: SecurityApp (Brian)

The SecurityApp(SecApp) is really quite complex, and has many features that are not initially apparent. I'm quite proud of it, frankly. There are 6 major files/pieces to discuss: MyMessagingService, LoginActivity, PagerActivity, ControlFragment, SensorConfigFragment, and CameraFragment. They will be described in that order.

Messaging

The MyMessagingService file, along with some lines in the Manifest, set up the device to receive notifications from the server. Services are declared in the Manifest, and they are initiated when the app is first installed and on every reboot thereafter; they are not instantiated from within the app proper. So how does it handle notifications? There is a distinction between what happens when the app is open in the foreground versus all other times. While in the foreground, if a Firebase Message is received, it executes the onMessageReceived function and does nothing more. In my code, I unpack that Message and create and display a Notification. While in the background (or if the phone is locked), when a Firebase Message is received, the service automatically translates the fields in that message into parameters for a new Notification to display. In both cases, an Intent is set up so that, when a Notification is clicked, the Login activity is launched with an intent to indicate which kind of notification was actually clicked, and either stay on the Control page or jump to the Camera page.

Another important concept is the FirebaseInstanceId, which I simply refer to as the AppToken. This is essentially the delivery endpoint for the Firebase Messaging Service; it's auto-generated, globally unique, and doesn't need any sort of login or authentication to get it. This token must be known by the server in order to send Messages to a given phone; that registration is done in the Login activity.

LoginActivity

The first page that a user sees when they launch the app is the Login screen. When it hits onStart, if Autologin is enabled, it will try to re-use an existing Authentication session or re-use a username/password from SharedPreferences. Otherwise, it waits for the user to hit Submit, then launches a chain of asynchronous callbacks. First, it *waits* for the user/pass to authenticate; then, if it succeeds, it *waits* while it checks to see if the Database node for that user exists; if that fails, it creates all the needed fields with default values, then *waits* again for however long it takes to ask the background MessagingService what apptoken it uses; and once THAT has finished, it sends this apptoken to the Database and only then does it finally proceed to the PagerActivity, the main hub of the app. It's worth noting that in some/all of these asynchronous callbacks, UI accesses are forbidden and cause crashes; to animate my progress bar I had to set up many short Runnables that run on the LoginActivity main thread. It's also worth noting that this parses the launching Intent to see what page it should display when it launches the PagerActivity.

PagerActivity

The PagerActivity is quite simple. It contains a ViewPager hosting the 3 fragments described below. It only does 4 simple but important functions. First, it hosts the DatabaseListener for pi_connected. This was because I thought it would be silly and inefficient to have 3 fragments independently listening to the same field; wiser to have the shared Activity listen. To get the behavior I wanted, I had to add a new function to the ViewPager Adapter that holds a list of the instantiated fragments so I could access them and their setPiConnected() functions. The second thing this activity does is set up a PageChanged listener; this is used to dismiss Notifications when the user opens the page they pertain to, and also when the user leaves the Camera page it aborts any audio recording they may have been taking. The third thing accomplished by the PagerActivity is that is hosts the callback function that runs when the user accepts any permissions dialogs the system throws up. Since I know what the user was trying to do before we had to ask for any given permissions, I use my reference to the hosted fragments to retry executing that function. Finally, the last feature in PagerActivity is the ReturnToLogin() function. This had to be implemented at this level so I could finish() the PagerActivity and preserve my illusion of a one-activity app. The rest of what the function does should be apparent in the comments.

ControlFragment

The ControlFragment is actually the simplest of my fragments. This is the “main screen” of the app, but there isn’t actually much here. Everything done here is self-explanatory, nothing fancy at all. Just reading from and writing to the Database and SharedPreferences. At one point, there was an EditText box on this screen to control the “timeout delay” of the system. I was annoyed by how it was always automatically selected when opening the activity, and therefore bringing up a soft keyboard, so I discovered that I could explicitly declare a non-editable TextView as “focusable”, and it would automatically get focus instead of the EditText. However, we ended up removing that feature, so my workaround was not used.

SensorConfigFragment

The SensorConfig fragment uses a RecyclerView with a custom item layout. Everything here is well-documented by comments, variables and function names. It builds a SensorListObject from the displayed spinners, turns it into a JSON via the Jackson serializer library, and stores it in the Database as a simple string. The Pi reads this and uses the Jackson serializer to unpack it into an identical object, to know what sensors are on what pins. No data is sent to the Database until the Apply button is clicked.

CameraFragment

This is by far the most complex part of the app. This contains everything to do with sound recording, as well as implementing the whole image pipeline described above. It's big, but function names and comments do a good job of explaining everything. I even left in most of my code that would have supported VOIP, though it's commented out because the Pi was unable to support the library. We instead replaced with with the ability to record audio files and upload them to the Pi for playback during an alarm. Permissions must be requested before saving an image to external storage (accessible by other standard photo viewer apps) and before beginning to record a new audio file. Both the audio upload and photo download have progress bars associated with them.

How it works: Raspberry Pi (Jamie + Brian)

The Raspberry Pi Base Station for the app, running Android Things, communicates with the phone app solely via the Firebase Database, which is where it sends and receives control signals. While it also uploads and downloads from Firebase Cloud Storage, it does so only in response to the control signals. The base station does not receive any information from the Firebase Functions.

The base station app contains two activities: a login activity, which displays an email and password prompt (this is currently hardcoded with an email and password to speed up the setup process), and a main activity, which runs all of the actual security and alarm code. The login activity is responsible for checking that the user is a valid user of the app, and logging them into Firebase Authentication. The main activity can then use that authentication, and user ID, when checking data from the database and cloud storage.

At its core, the base station checks configuration changes from the firebase (armed status, new custom alarm sound file, sensor configuration, etc), listens for sensor changes, and upon alarm trigger, writes a triggered value to the firebase, along with uploading an image taken by the camera.

The sensors on the Pi are reconfigurable by the user. When the config field changes in the database, it is saved to a string, the Jackson serializer is used to rebuild a SensorListObject from that string, and then the code iterates over it, applying settings appropriate for that sensor type to that pin. Just to be extra safe, there is a flag "mIsConfiguringGpio" that is set at the beginning of this function, and unset at the end, that prevents any stray alarms from happening as a result of the pins' state changing or anything strange like that.

Attached to each of the GPIO pins is a callback, which is triggered upon a detected sensor value transition. The code then checks 3 local variables to determine whether to treat this as an actual alarm, or just meaningless noise. If the base station is armed, and is not still setting up or configuring, then it triggers an alarm. To do this, it sets the “pi_triggered” field to be true in the database (causing a Firebase Function to send a notification), and then plays the custom alarm sound, which plays on loop for some duration (currently set to 5 seconds). It also follows the “mousetrap” policy of self-disarming after being triggered once; we named it this because it needs to be re-armed after each time it is tripped. Whether this would be desirable in a finished product is debatable, but in the system as it is now it is working as intended.

The sound pipeline works as intended, and is 100% functional (though our testing only used earbuds, so the sound quality and volume was abysmal). On bootup, the Pi checks locations in this sequence: first, it checks if there is a sound file in local storage, which is only the case if it has in the past successfully downloaded a recording from the cloud. If that is missing, it then checks the cloud to see if a file was uploaded while the Pi was off. If that still isn’t there, the last resort is to use the default alarm sound included in the assets folder. After bootup, the app is able to upload the freshly recorded sound file, and after setting a flag to indicate that it has done so, the Pi downloads it from the cloud and saves it to local storage to use it henceforth.

Aside from the fact that the camera doesn’t work, the image pipeline works perfectly (lol). If we were able to capture images, the pipeline of saving and uploading files works as intended. If it worked, it would work by first capturing the small image, and once the camera is finished doing its thing, it launches in parallel two jobs, that of capturing the high-resolution image and uploading the low-resolution version to the cloud. When the low-resolution one is finished uploading, it sets the state to 2. Once the high-resolution image is done capturing, it is saved to local storage. When the user requests the high-resolution image, it is uploaded to the cloud and then the state is set to 4 when it completes.

How it works: Firebase Functions/Node.js (Jamie)

Because we wanted the phone to be notified in real-time about base station changes, but also to react well to connectivity loss, we wanted some functionality of the alarm system to run on a web server. Since we relied heavily on the Firebase database, Firebase functions were the obvious choice.

Firebase Functions are a series of functions, written in Node.js (a JavaScript based language), that are deployed on the Firebase server, and can act upon Firebase data. They work with the Firebase Cloud Messaging system to be able to send notification to the phone app. The functions that we used are set to trigger on database value writes.

We used two different function: one ran when the pi triggered an alarm. If the trigger occured when the pi was armed, it sends a notification to the phone. The other function activated when the pi lost connection to the firebase database, and also sent a notification. Both notification also cared a status data bit, to be read by the phone app, and also signaled to be accompanied by a sound

Issues while making it

One minor issue encountered while building the app was with getting the "context" of the activity in functions within the scope of the fragments. Sometimes these functions would cause the app to crash if the app was minimized when a database value changed, or when PagerActivity tried to modify the "connected" state of a fragment that had been unloaded.

Another strange issue I encountered was that some listeners (but not all) wouldn't allow any access of UI elements. Specifically this was happening in the VOIP callbacks (which I eventually gave up on) and in the Login callbacks. Database value listeners work just fine, and upload completion/failure listeners also work fine.

Inexplicably, the night before our presentation, either our Pi hardware or our Android Things image failed in a critical way. We were no longer able to take pictures with the camera, and the code executed the same way, and hung in the same location, both with and without the camera physically connected to the Pi. Therefore, the Pi was somehow rendered unable to detect the camera, ruining the most impressive and most presentable part of our system. We were able to get images from the camera once during testing, but reverting our code to the way it was at that time simply does not produce the same results.

Notifications are inconsistent… when the field is set to true via the console, the app receives an alarm notification. When the field is set to true via the Pi, it sometimes does not receive an alarm (even though I see the value change). This was discovered far too late to effectively debug or fix.

Final status:

We suffered what might be a hardware failure that prevented us from getting the camera working by the due date, but the image pipeline works overall… it is only capturing images in the first place that is broken. Alarm notifications on the app are inconsistent; it's unclear whether this is an issue with the Firebase Functions or something we overlooked in the Pi alarm function. Sadly, we weren't able to even get a start on a range finder/motion sensor. Aside from these 3 points, everything in the overall system is working perfectly, and we have delivered our targets.

Who did what

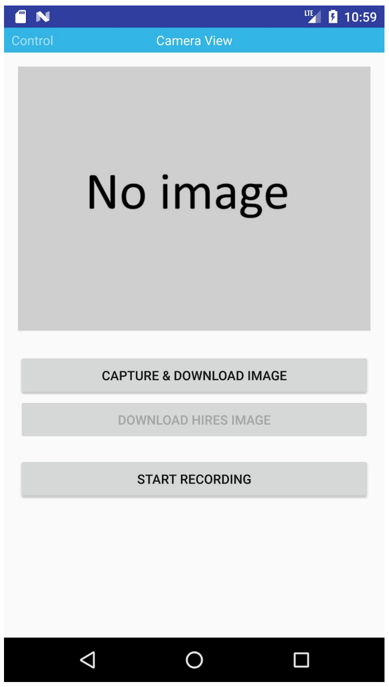
Brian: App, PI (sensor config, refined alarm executable, overall reformat/cleanup)

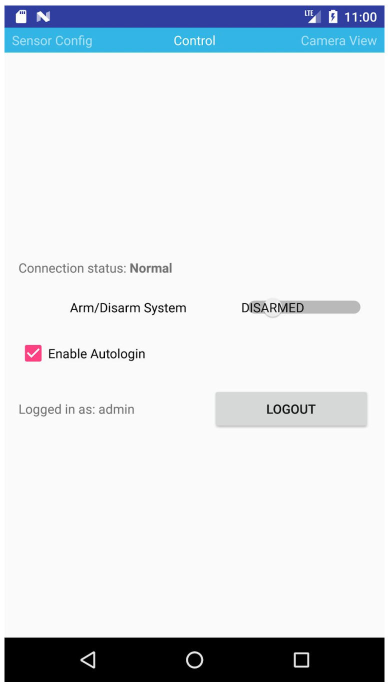
Jamie: PI (audio player, figured out how to use the camera in the first place, first skeleton of everything else), Node.js notification generators

What we would change if we were to do it over again

Clearly our system was too ambitious. We were trying to learn too many new things and add too many features, and weren't willing to give up on any of the stretch goals even as we were approaching the deadline, so we failed to meet some of our core goals. Frankly I feel very cheated by how the camera spontaneously failed on us, too late for us to find a replacement camera or board. I'm utterly mystified by the inconsistency of the notifications; I suspect that unlike the camera, it is something we are legitimately doing wrong, something we could theoretically fix, but don't have time to figure it out. Lastly, beyond simply starting sooner, I feel like we should have placed more emphasis on literal face-to-face sitdown testing, rather than working on our two halves independently.

App screenshots



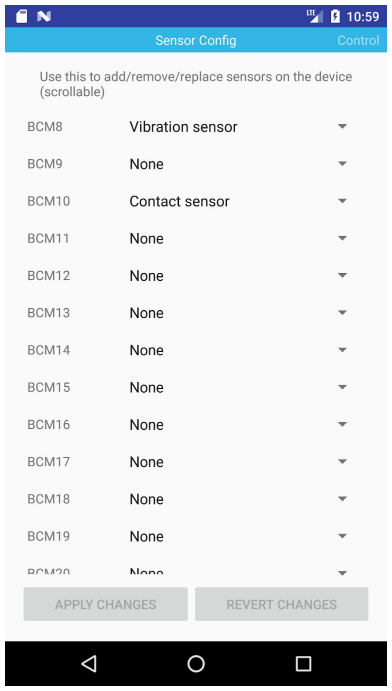


Connection status: **Normal**

Arm/Disarm System DISARMED

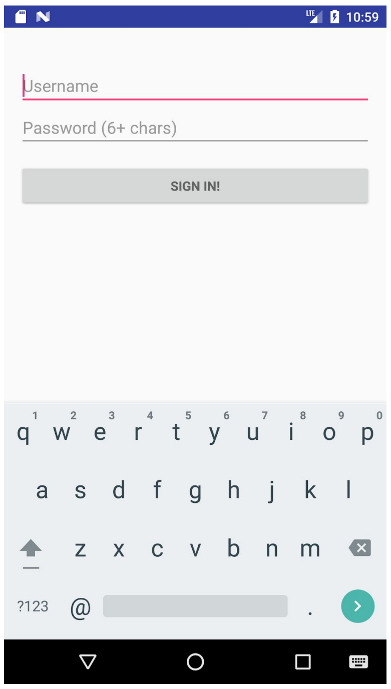
Enable Autologin

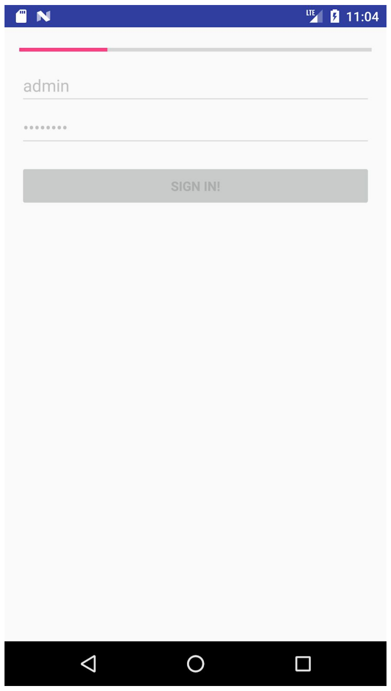
Logged in as: admin

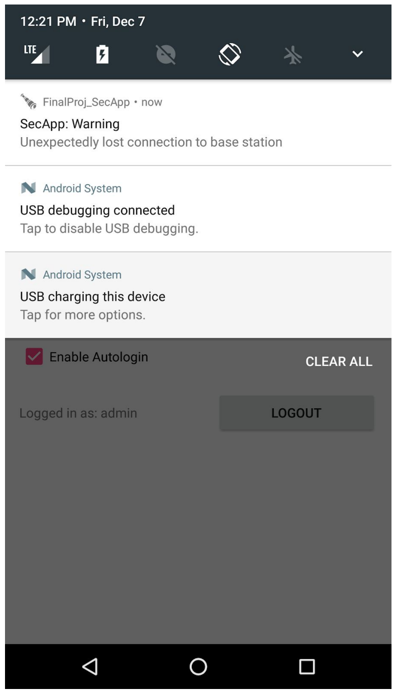


Use this to add/remove/replace sensors on the device (scrollable)

BCM8	Vibration sensor
BCM9	None
BCM10	Contact sensor
BCM11	None
BCM12	None
BCM13	None
BCM14	None
BCM15	None
BCM16	None
BCM17	None
BCM18	None
BCM19	None
BCM20	None







FinalProj_SecApp • now
SecApp: Warning
Unexpectedly lost connection to base station

Android System
USB debugging connected
Tap to disable USB debugging.

Android System
USB charging this device
Tap for more options.