Dylan Porter
Project Paper
Professor Jackson
05/11/2019

Silent Voyager

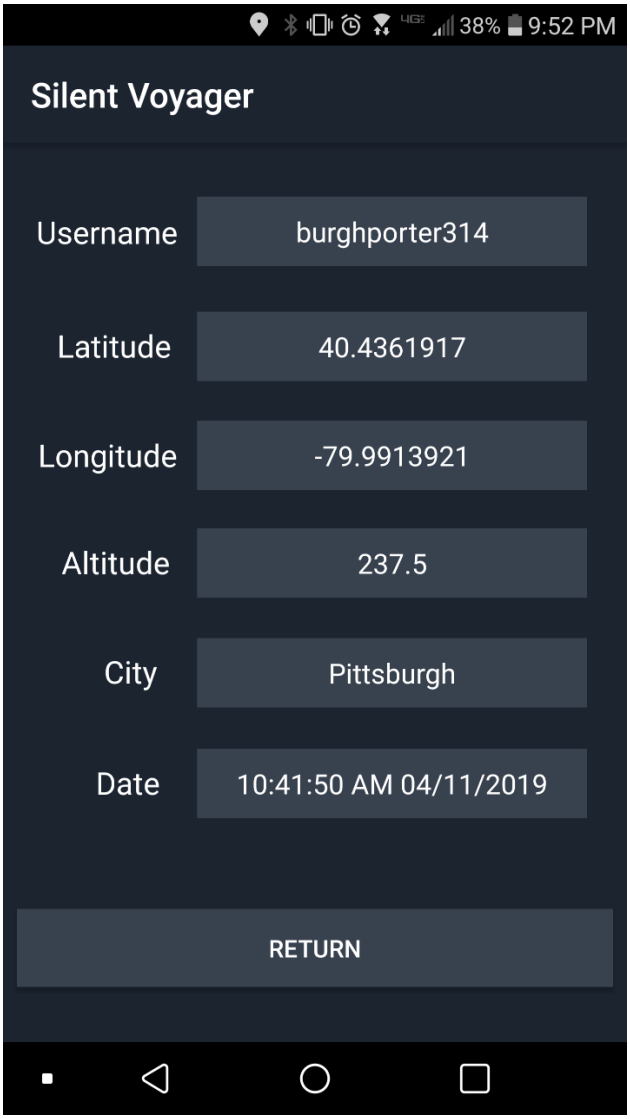
Website: http://burghporter31415.x10host.com/Silent_Voyager/
Source Code: github.com/burghporter314/SilentVoyager

BACKGROUND:

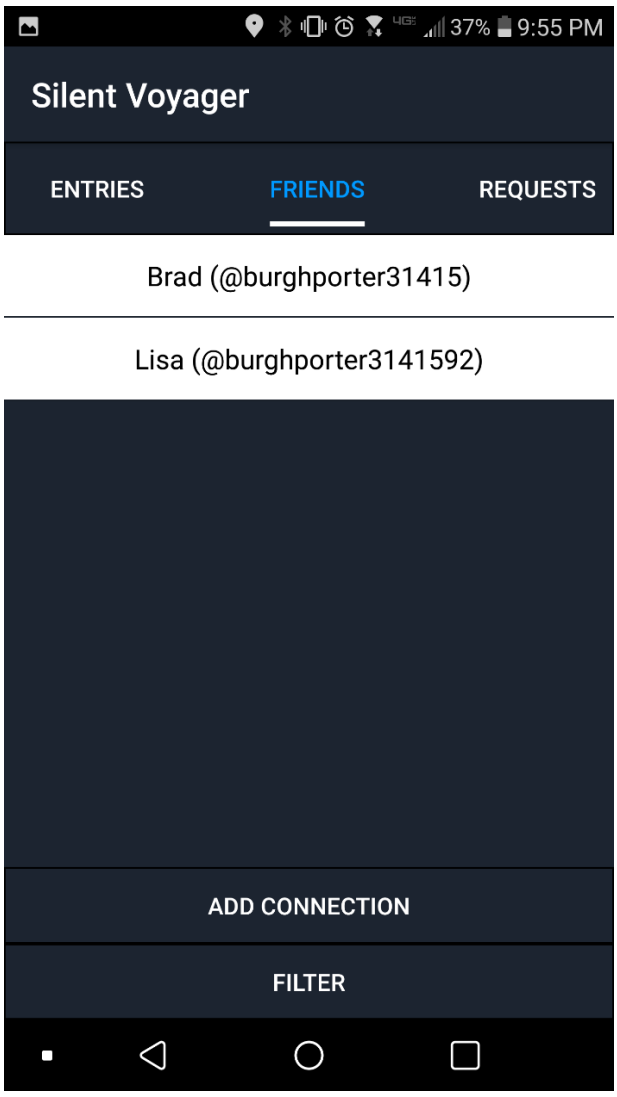
Silent Voyager, my Senior Project Application, is a utility that mimics many popular location tracker concepts. Silent Voyager consists of both Android and Web Application support. In total, the goal of the Silent Voyager application is to create a location tracker for the Android Operating System—extending analytic support to a Web-Based system that is easily accessible from both mobile and non-mobile platforms. A big part of both sides of the application is to provide easy user functionality.

With regards to the Android Application, a user can choose to either register or login to the Silent Voyager App. Naturally, if a user chooses to register, they will need to fill out the required information suggested on the App before proceeding to the main dashboard page. If the user already has their credentials, logging in will allow them to proceed to the dashboard page as well. Luckily, the software stores the username and password in a file within the device—so there is no need to enter the credentials more than once.

Once logged in, the user should be able to see the components of the dashboard page. The page includes entry, user, and map fragments. Within the entry fragment, the user will be able to see all coordinate upload times. If a specific entry is clicked, another activity will pop-up showing information like username, latitude, longitude, city, and date. Here is an example:

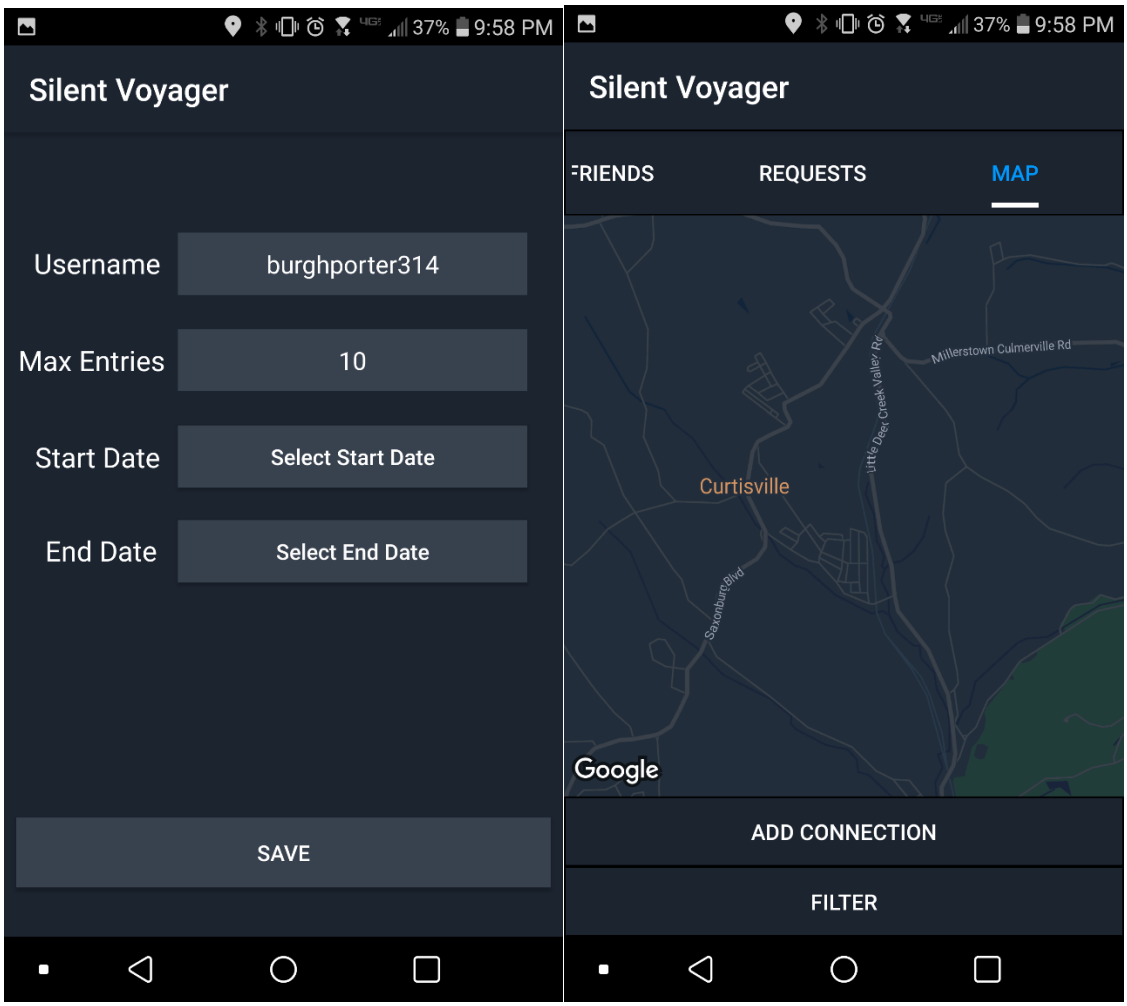


When the user fragment is selected, the Application will display all of the connected friends or users associated with the account. Ultimately, a user can monitor the location of all of their friends within the app—so the functionality is important to have. From that fragment, a user can also request other accounts to be their friends. Here is an illustration:



Finally, the map fragment will display the current location of the user—and up to one-thousand previous locations detected. To accomplish such a task, the map will display a polygon of lines indicating the direction of movement. Ultimately, the user will have a map of total movement for that specific account. The information is mapped directly from coordinate uploads that were different enough in location and time. There is also a filter option within the main dashboard page. When clicked, a new activity will be brought up with the username, max entries, start date, and end date fields. If desired, the user can adjust those parameters to visualize different results on the dashboard. This can prove extremely useful to data analysis through adjusted variables. Once the user presses the

save button, the dashboard will update the entry, user, and map fragments accordingly. This is what the map and filter activities look like:



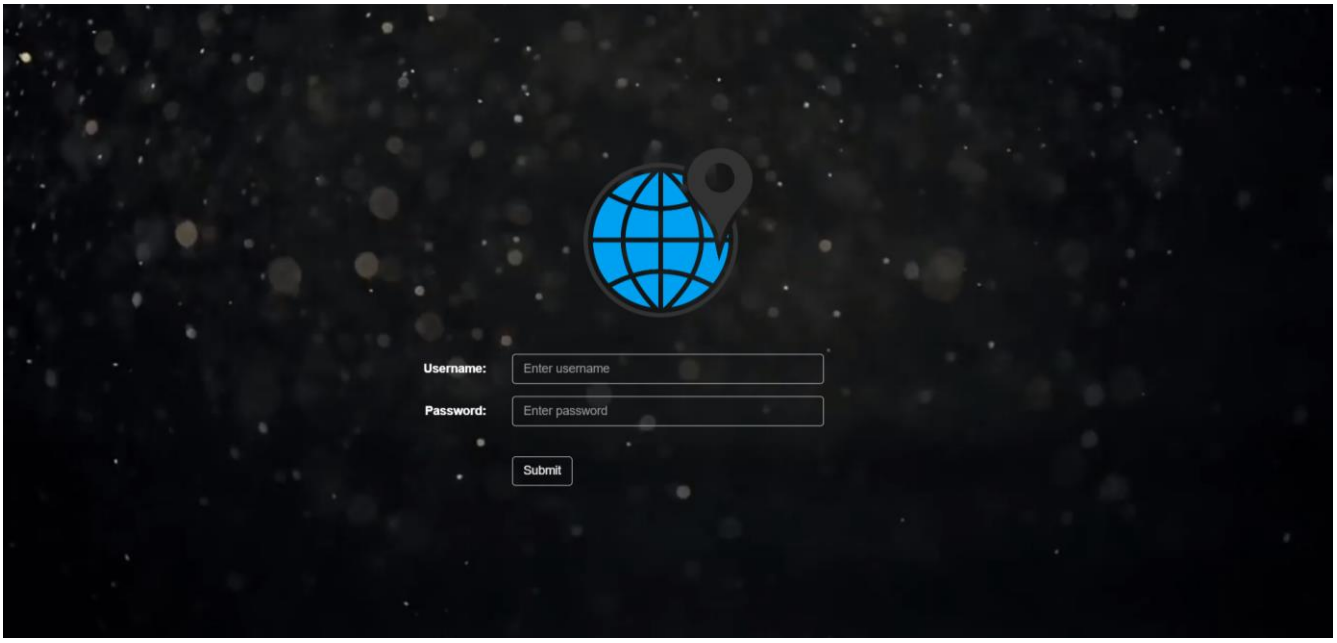
It should also be noted the Silent Voyager Application uses Android Services to upload the coordinates. For the Application to consistently upload coordinates with high reliability—it needed to be programmed to work outside of the App visual. Due to the nature of location detection and the reliability needed, Silent Voyager implements both a Foreground and Background Service to upload coordinates outside of the application. It is a never-ending process that will start up on the opening of the application—and the powering on of the device (through the use of broadcast variables). If there is no internet available, the Application will simply store the coordinates in a text file on the device and upload them when the internet turns back on.

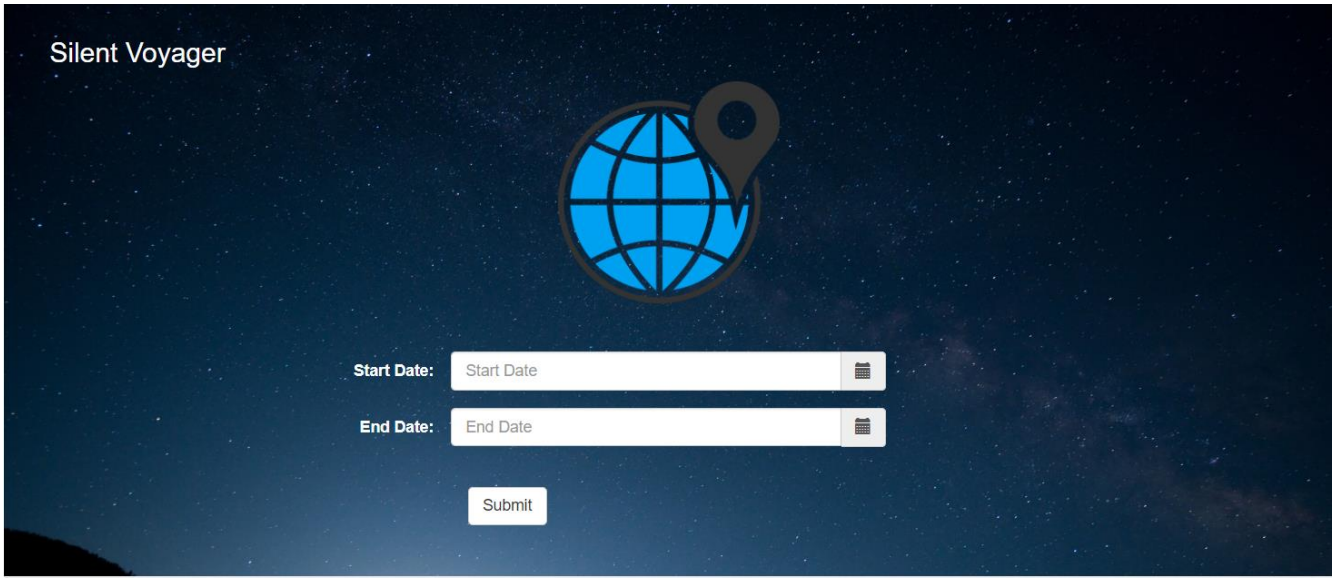
With regards to the Silent Voyager Website, most of the same functionality provided within the Android Application is available. Currently, the main page only has a login functionality—as there is no immediate need to provide a register page outside of the Silent Voyager App. Once logged in, the user will see a variety of content provided on the dashboard page. On the website, there are three main sections—the header, the table, and the map.

For the header section of the site, the user can, like the Android Application, enter the username, max entries, start date, and end date for the query result. Available in the header are easily-used date pickers where the user can select the year, month, day, hour, and second of the date. Mimicking the Android Application, the header of the website provides a drop-down for both the username and max entries fields. Once all of the fields are completed, the user can click the submit button to get the updated query results.

For the table section of the page, the Silent Voyager website provides a great deal of functionality. For instance, there are multiple columns including username, latitude, longitude, city, and time that the user can filter or sort the results by. In addition, there is an entry selector and a search box provided by default for the table—allowing the user to easily filter the results and check occurrences with the other components of the page.

Finally, for the map section of the page, the Website provides a visual that is very similar to the Android Application. For instance, the map showcases all of the detected points with a marker symbol—ultimately connecting all of the markers with a polygon line so that the overall journey of the user can be traced. If the user clicks on one of the markers, it will display the coordinates and the position relevant to the very first position. Here are some images of the website:


Username:
Enter username
Password:
Enter password
Submit


Silent Voyager
Start Date:
Start Date
End Date:
End Date
Submit

Location Info

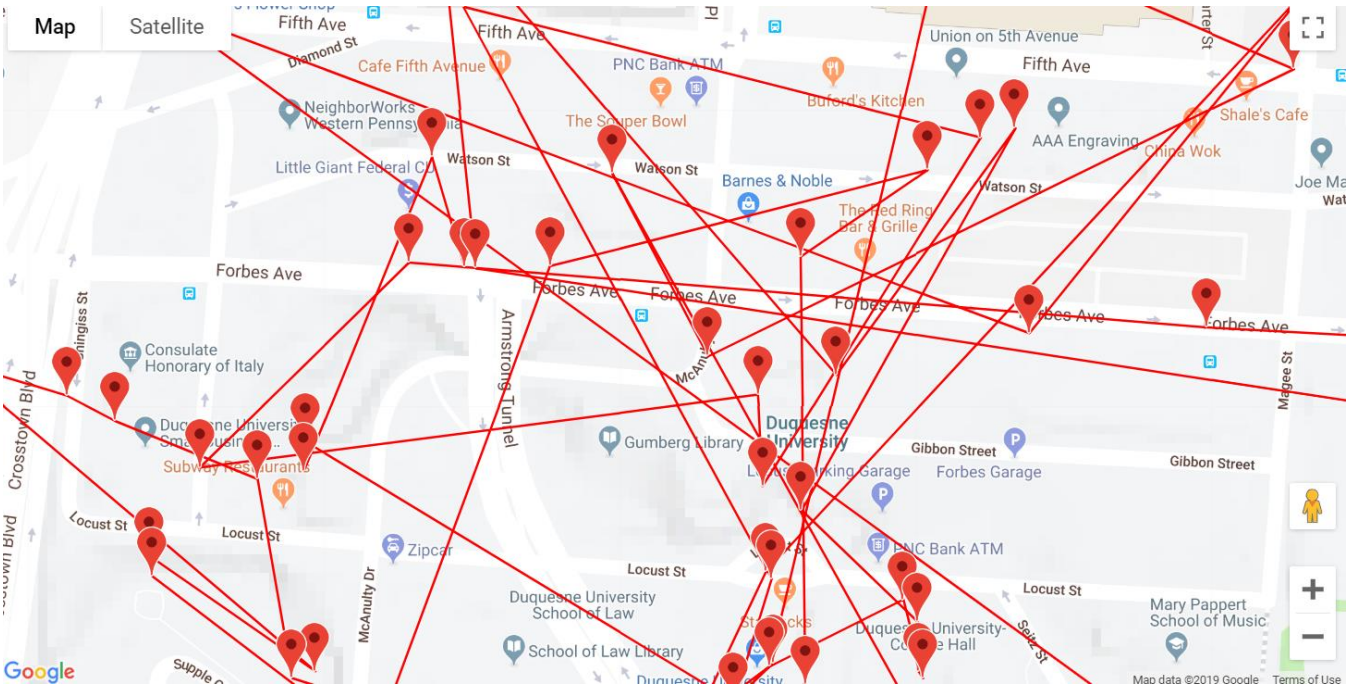
Below is the information about the specific user's location starting at the most recent occurrence.

Showing Results for: 04/10/2019 10:07 AM to 04/12/2019 2:07 AM

Show 10 entries Search:

Username	Latitude	Longitude	Altitude	City	Time	Date
burghporter314	40.4372586	-79.990793	0.0	Pittsburgh	2:26 PM	04/11/2019
burghporter314	40.4384859	-79.9897837	0.0	Pittsburgh	2:24 PM	04/11/2019
burghporter314	40.43895	-79.9921844	0.0	Pittsburgh	2:22 PM	04/11/2019
burghporter314	40.43693102	-79.99075445	214.0		2:18 PM	04/11/2019
burghporter314	40.4360095	-79.991123	238.0		11:13 AM	04/11/2019
burghporter314	40.43583678	-79.99172277	240.0	Pittsburgh	11:13 AM	04/11/2019
burghporter314	40.4369408	-79.9936266	233.5	Pittsburgh	11:11 AM	04/11/2019
burghporter314	40.43701309	-79.99364262	235.0	Pittsburgh	11:11 AM	04/11/2019
burghporter314	40.436604	-79.9928733	0.0		11:10 AM	04/11/2019
burghporter314	40.43769832	-79.99456235	212.0	Pittsburgh	11:09 AM	04/11/2019

Showing 91 to 100 of 269 entries Previous 1 … 9 10 11 … 27 Next



Overall, the Silent Voyager Android and Web Application work together to provide the user visual ability across multiple devices. With the features provided, a user can filter the results of an individual, device or user. They can also provide a start date and end date to see a visual representation of travel within that time period. Overall, the application can help determine safety, help aid in finding lost devices, and help determine the location of connected devices.

Progress First Increment

Throughout this first increment, I have made progress mostly on the Silent Voyager Android Application. So far, the main focus has been on data retrieval from the server, analysis, and visualization. As mentioned in the proposal, I intended to work extensively on the dashboard page of the Application. Originally, I had provided a proposal for city, coordinates, and map fragments. This has not proven useful—as the city is derived from the coordinates anyway. In lieu of the original design, I added an entry and map fragment to the Application.

So far, the fragments are working successfully—including their on-click listeners that provide the user with additional activities for help with data visualization. For instance, the entry fragment now displays the results in a time format of occurrence. Once clicked, it brings up all of the relevant location data associated with that time entry. I have also updated the XML file associated with that fragment—and I programmed a custom Text View component to better match the overall flow of the application. The custom Text View helps compliment the overall UI design of the Silent Voyager application.

```
/*Start a new Activity upon a click--sending over the appropriate data in a Bundle*/
listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {

    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {

        String[] rowElements = arr2[(int)id].split(",");

        Intent intent = new Intent(getContext(), DashboardInfo.class);

        intent.putExtra("username", rowElements[0]);
        intent.putExtra("latitude", rowElements[1]);
        intent.putExtra("longitude", rowElements[2]);
        intent.putExtra("altitude", rowElements[3]);
        intent.putExtra("city", rowElements[4]);
        intent.putExtra("datestamp", rowElements[5]);
        intent.putExtra("PATH", bundle.getString("PATH"));

        startActivity(intent);

    }

});
```

Illustration 1: Example code from the entry fragment that displays user data

With regards to the map fragment, I have also made substantial progress as well. Like the Silent Voyager Website, mostly all visualizations have been added to the Android widget including polygon and marker components. I also dealt with the default permissions required from the map access including Manifest COARSE_LOCATION and Manifest FINE_LOCATION. Instead of requesting these permissions on the loading of the map—I simply request them when the Silent Voyager Application loads. See illustration 2 for how the map is populated.

```
/*POLYLINES*/
PolygonOptions rectOptions = new PolygonOptions();

for(int i = 0; i < (Math.min(50, this.arr.length)); i++) {
    String[] components = this.arr[i].split(",");
    rectOptions.add(new LatLng(Double.parseDouble(components[1]),
                               Double.parseDouble(components[2])));

    Marker marker = map.addMarker(new MarkerOptions()
    .position(new LatLng(Double.parseDouble(components[1]), Double.parseDouble(components[2])))
    .title(i == 0 ? "Last Position" : i == this.arr.length - 1 ? "Start Position" : "Position: "
            + (this.arr.length - i)));

    if(i == 0) {marker.showInfoWindow();}


}

Polygon polygon = map.addPolygon(rectOptions);
polygon.setStrokeColor(Color.parseColor("#0099ff"));

map.animateCamera(CameraUpdateFactory.newLatLngZoom(coords, (float)18.0));
map.setMapType(GoogleMap.MAP_TYPE_HYBRID);
```

Illustration 2: Code to populate Android Map

I also added two additional Android Activities including dashboard_info and filter. The dashboard_info Activity is created on the click of an entry element (see Illustration 1 on how the activity is created). Within the activity, there are columns for username, latitude, longitude, altitude, city, and date. The activity was programmed with the overall app theme in mind as well. The date and city are automatically parsed for better visualization to the end user. For the filter activity, there are columns for username, max entries, start date, and end date—all of which can be modified by the user. The start date and end date columns utilize a user-friendly date picker that is easy to use and change. Illustration 3 shows how the activity displays and sends back the date results selected by the user.

```
    @Override
    public void onActivityResult(int requestCode, int resultCode, Intent data) {

        /*Handle the pending intent based off of the code that was unique to Start Date and End Date Widget*/
        Bundle bundle = data.getExtras(); //Would have d1, d2, d3 parameters

        switch(resultCode) {

            case 100:

                this.startDate.setDate(bundle.getInt("d1"), bundle.getInt("d2") + 1, bundle.getInt("d3"));
                timeFragmentStart.show(getSupportFragmentManager(), "timePicker");
                break;

            case 200:

                this.startDate.setTime(bundle.getInt("d1"), bundle.getInt("d2"));
                this.dataIntent.putExtra("startDate", this.startDate.toString());

 this.pickerStartDate.setText(FragmentUtils.returnDateStamp(this.startDate.toString().split("_"), false));
                break;

            case 300:

                this.endDate.setDate(bundle.getInt("d1"), bundle.getInt("d2") + 1, bundle.getInt("d3"));
                timeFragmentEnd.show(getSupportFragmentManager(), "timePicker");
                break;

            case 400:

                this.endDate.setTime(bundle.getInt("d1"), bundle.getInt("d2"));
                this.dataIntent.putExtra("endDate", this.endDate.toString());
                this.pickerEndDate.setText(FragmentUtils.returnDateStamp(this.endDate.toString().split("_"),
false));
                break;

            default:
                break;
        }
    }
```

Illustration 3: Pending intent handler for the filter class

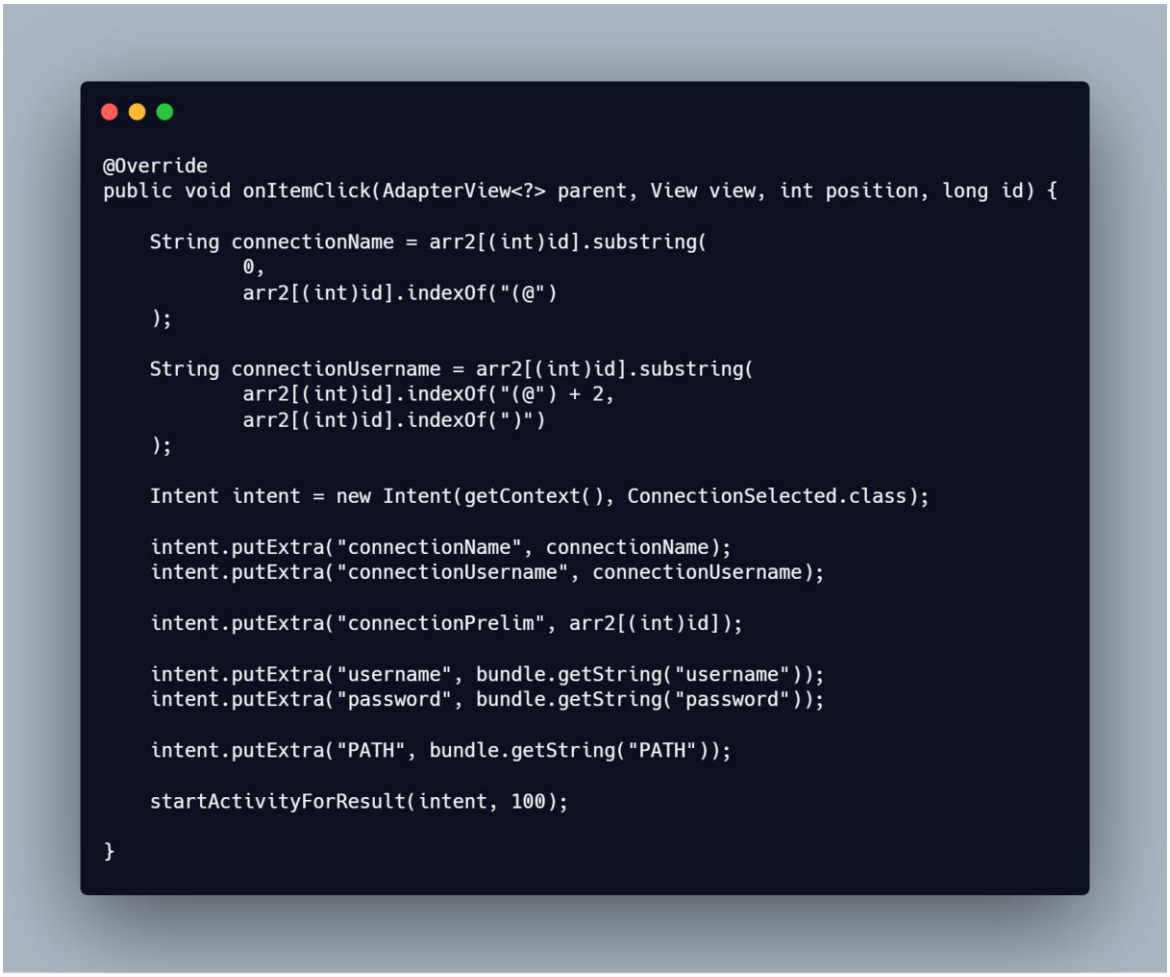
Finally, I have made progress on data passing between the activities. As before, the filter Activity needs to pass the column values back to the dashboard activity. In addition, the entry fragment needs to pass the needed information to the dashboard_info Activity. I've also implemented a variety of callback methods and onclick listeners for the Activities added during this increment.

Progress Second Increment:

Throughout the second increment of the project, I have made significant progress on the android and web-based applications. To start, I made considerable updates to the filter activity in the Android application. As mentioned before, I needed to find a way to update the corresponding fragment activities when the user pressed ‘save’ on the filter activity. After considerable adjustment to the dashboard activity, all components now update and populate the truncated data accordingly.

In addition to the work completed on the filter activity, I largely focused on multi-user support during this iteration. In total, I needed to add support for multi-user connection. As expected, this turned out to be a very difficult task—as there needed to be considerable changes both on the application and on the server-side scripts. Even with a few weeks to complete the task, I was unable to finish the work required for this functionality to be supported.

In the application, I added multiple new activities for the multi-user support components. For instance, I added a ‘connections’ fragment that displays the current connections associated with the current account (each individual item pertaining to one list view). When the user clicks on an individual item, the application will display the ConnectionSelected activity. In this activity, the user can view the corresponding name and username of a particular user that they are connected with. If they choose, they can remove the connection using this activity.



```
@Override
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {

    String connectionName = arr2[(int)id].substring(
            0,
            arr2[(int)id].indexOf("(@")
    );

    String connectionUsername = arr2[(int)id].substring(
            arr2[(int)id].indexOf("(@") + 2,
            arr2[(int)id].indexOf(")")
    );

    Intent intent = new Intent(getContext(), ConnectionSelected.class);

    intent.putExtra("connectionName", connectionName);
    intent.putExtra("connectionUsername", connectionUsername);

    intent.putExtra("connectionPrelim", arr2[(int)id]);

    intent.putExtra("username", bundle.getString("username"));
    intent.putExtra("password", bundle.getString("password"));

    intent.putExtra("PATH", bundle.getString("PATH"));

    startActivityForResult(intent, 100);

}
```

Illustration 4: Summoning selected user activity.

On the main dashboard activity, there is now an “add connection” button. Once pressed, the user will be taken to the ConnectionAdd activity. In this activity, the user can search for a specific user and view a returned result from the server. If the user is found, the user can click on the result and send a connection request. For the connection to be accepted, the requested user will have to login and press “accept” when a dialogue for the request appears. Once this happens, the connection will be added and both users can view each other’s location data. As mentioned before, this connection can be removed any time by simply going to the connections fragment, clicking on the user, and pressing the ‘remove connection’ button.

```
listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {

    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {

        final int pos = (int) id;


        /*FROM: https://stackoverflow.com/questions/2478517/how-to-display-a-yes-no-dialog-box-on-android*/
        DialogInterface.OnClickListener dialogClickListener = new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {

                switch (which){

                    case DialogInterface.BUTTON_POSITIVE:

                        resultFormPost.addPair("requested", listItems.get((int)pos));

                        Thread thread = new Thread(new Runnable(){
                            @Override
                            public void run() {
                                resultFormPost.submitPost(resultRequestPageRequest, MethodType.POST);
                            }
                        });

                        try {
                            thread.start();
                            thread.join();

                            Toast.makeText(ConnectionAdd.this, "Request send to: " + listItems.get(pos),
                                    Toast.LENGTH_LONG).show();

                            listItems.remove(pos);
                            adapter.notifyDataSetChanged(); /*Update the Data*/

                        } catch (Exception e) {
                            //TODO
                        }

                        break;

                    case DialogInterface.BUTTON_NEGATIVE:
                        //No button clicked
                        break;
                }
            }
        };
```

Illustration 4: Example of sending a connection request.

For the server-side part of the application, I had to add additional scripts to handle the multi-user functionality. In total, I added request_connection.php, request_connection_results.php, add_connection.php, and user_match.php. The user match script is the most intuitive. It takes in a

‘search_param’ POST variable and queries the SQL database using the LIKE operator with the following indicator: ‘%”.$_POST[‘search_param’].”%’. This line of code will return any results that contain the element that the user is searching for. For instance, if the user searches ‘dog’, possible results may include adog, mydogburgh, and dog since all of those sequences contain the word dog.

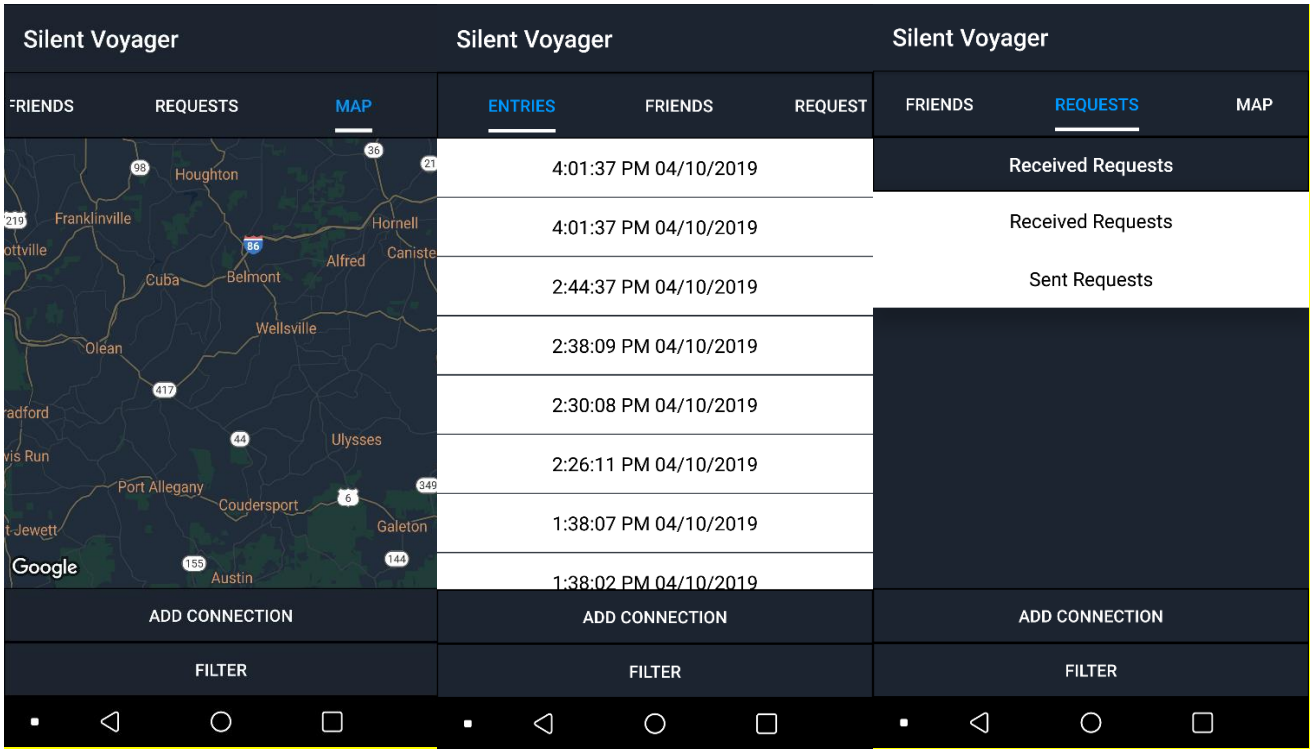
```php
$sql = "
SELECT A1.Username, Password FROM REDACTED AS A1
INNER JOIN REDACTED AS A2
ON A1.Username = A2.Username
WHERE A1.Username='".$_POST['username']."'
AND Password='".$_POST['password']."'
";

$result = $conn->query($sql);
$counter = 0;

if($result->num_rows > 0) {

    $servername="REDACTED";
    $username="REDACTED";
    $password="REDACTED";
    $database="REDACTED";

    $conn = new mysqli($servername,$username,$password,$database);

    $sql = "
    INSERT INTO SilentVoyagerRequestedConnections (requester, requested)
    VALUES ('".$_POST['username']."', '".$_POST['requested']."');";

    $result = $conn->query($sql);

} else {
    echo "INVALID";
}
```

Illustration 5: Example code for created a requested connection in PHP

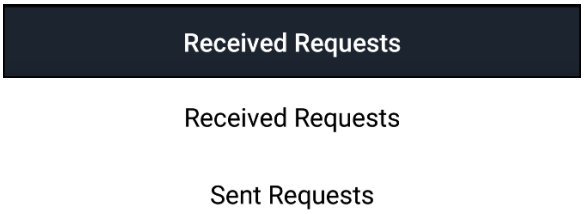
For the request_connection_results.php script, results corresponding to the user are returned. For instance, the script will return all of the individuals that a specific username has requested. This is starkly different from the request_connection.php script since it does not alter the state of the database. For the request_connection script, however, a new connection will be made corresponding to the username making the connection and the person being requested. The relationship for all purposes will be reciprocal. Finally, the add_connection.php script finalizes the relationship. Once the user accepts a request from another user, the relationship will be formalized in the database—again granting access to all of the users' data from both ends. It should be noted that the PHP scripts are vulnerable to SQL injection. Currently, I am working on escaping the variables to fix this problem.

Progress Third Increment:

For the third increment, I worked almost entirely on the Android Application. To start, I made significant changes to the user interface design. Before this update, I was not satisfied with how the application looked. Overall, I needed to develop a consistent theme with a professional look. By this logic, I decided to go with a consistent dark-blue, white theme that would adequately reflect the application's components and displayed information. This proved to be visually appealing, as there was a more consistent theme now provided across the application. Some of the various components are now easier to view. These are some of the images of the newly-updated user interface:



Additionally, I also worked on message passing between the fragments of the application. Across the various fragments and activities, I needed to find a way to update the user interface of the application when a change was made. For this iteration, the requests fragment was a pivotal component for consistent functionality across the device. Under the fragment, there is a list popup window that the user can click to switch between received and sent requests:



On the received requests option for the popup window, the user can either accept or reject a request sent to them. For the sent request option of the popup window, a user can choose to cancel a sent request to another user. Either way, the entire application will be updated accordingly if the user performs any action within this fragment.

Finally, I had to work on integrating the multi-user support with an extremely difficult updating mechanism. Unfortunately, I did not use live data for this application, so I had to find another way to

efficiently deliver new information to an end device running the Silent Voyager application. To start, I knew that I could use some of the information already on the device to truncate the data sent from the server. Overall, I was looking for a way to limit the breadth of data sent over the network. This is because the application requests new data every ten seconds and a lot of repetitive data sent over the network could be costly.

To start on the update mechanism, I created a check update PHP script that expected a variety of variables that were used to determine which data to send back. The simplest solution came with the first part of the request. On the device, the user has the most recent date stamp already loaded. When the device makes a request for an update, I decided to send the most recent date stamp to the PHP script. Once received, the script will query the entries database for all of the entries that have a date stamp larger than the one sent. This is preferable to sending the entire batch of date stamps back to the device every ten seconds. Naturally, the results will then be appended to a global JSON string.

Additionally, the device also needs to worry about newly received connections, received requests, and sent requests. Unfortunately, I did not have a clear way to send back the necessary data on this, so I had to add a unique ID column to the appropriate databases. In the request, the device sends three numbers num_0, num_1, and num_2. These numbers represent the most recent connection, received request, and sent request, respectively. When the update script receives the three numbers, it will query the appropriate databases and return the results that have a higher corresponding ID than num_0, num_1, and num_2 for each database. These results will also be appended to a JSON string, including the relevant data and updated numbers.

The check update script was a good solution up to this point, but there was an additional problem that I had to deal with. Even though I could ensure that the user was receiving the most updated data and new connections and requests, the user was not receiving data on deleted requests. This is because a deleted connection or request does not affect the maximum ID associated with the databases. To fix this problem, I have an action script that is called when a user removes a connection,

removes a request, or rejects a request. When any of the scripts are called to perform those actions, a new entry in the actions database is created.

Quite simply, there are two possibilities for the action script. For the action, there is either a DELETE_CONNECTION or REMOVE_REQUEST string. For the removal of a request, there is no need to make a distinction between rejecting a request or removing a sent request, as the result is the same. The total fields of the database are {ID, ACTION, USER1, USER2}. In that format, there is an action from user1 against user2 that is being performed. It is now guaranteed that these actions will refer to removable data that cannot be queried by the original max ID method before. It is important to keep in mind also that for one action, two entries must be made. If this was not the case, only one end user (in relation to two end users) would see a result from a query of the action database.

For the final part of the check update script, I needed to query the action database appropriately so that the script could return results pertaining to any removed connections or requests. To do this, I retrieve all the entries corresponding to the user making the check update request. Within the returned results, I also receive the max ID in the action database referring to that user. Once completed, the check update script appends the results to the global JSON string including removed connections, requests, and the max ID.

Finally, the check update script is completed and will send a JSON string back to the requesting user. The JSON string has the following format: {“names_0” : FIELDS, “names_1” : FIELDS, “names_2” : FIELDS, “last_id_0” : FIELD, “last_id_1” : FIELD, “last_id_2” : FIELD, “last_id_3” : FIELD, “ACTION” : FIELDS}. Once the device receives the JSON string, it updates the UI components appropriately. Additionally, it also makes a POST request to a delete action script with the max ID field associated with the latest action received. The script will then delete all entries related to that user. Now that everything is completed, the UI is updated appropriately, and minimal data was sent. It should be noted that the application does not query the check update script when the user exits

the application. This helps reduce overhead, as it is unnecessary to update any components if they are not visible.

Conclusion / Future Work:

With the completion of the update mechanism, the Silent Voyager Android Application is now completed. Upon testing the application on multiple devices, it seems to work perfectly. I now have multi-user support which I can view through the analytical tools provided by the application. With that said, the Silent Voyager website does not provide an updating mechanism or multi-user capabilities. Since the Android Application took a great deal of time to complete, I did not have the opportunity to translate those features to the website.

In total, the Silent Voyager project was an immense success. I achieved a newly-founded understanding of Android and Web development. I dived into real-world problems that were challenging but rewarding to solve. I also provided consistency and reliability through edge testing on the Android application. I believe that there is immense potential for this application as well. In tandem with its complex updating mechanism and sleek user interface, there are many additional features that I could foresee adding to the application.

For instance, the current version of Silent Voyager does not have any settings at all. Once the user logs in on their device, it will upload coordinate data indefinitely. Unfortunately, there is no way to stop it from the app screen. There should also be a permission base that users can specify for their various connections. For instance, a given user might not want to share their information with another individual connected to their account. As it currently stands, there is no way to limit access. Connections are binding and provide the user with access to all information related to their coordinates.

I could also see adding more utility to the Silent Voyager application. For instance, the application could have a better analytics platform to it. As it currently stands, only basic information is provided. Also, the entry fragment should have more purpose as well. There is little purpose for providing the user with the hard-coded latitude and longitude numbers. Perhaps the app could use an

algorithm to provide analytics about commonly visited places by time period. This would provide more analytical power at the hands of the user and would also make the application more useful.

Silent Voyager Screenshots: