Location Services

CS443 – Mobile Applications
Instructor: Bo Sheng

Location Services

- Applications access the location services supported by the device through the classes in the android.location package.

- The central component of the location framework is the LocationManager system service.
  - As with other system services, do not instantiate a LocationManager directly.
  - Request an instance from the system by calling getSystemService(Context.LOCATION_SERVICE). The method returns a handle to a new LocationManager instance.

Location Providers

- GPS
  - The most accurate
  - Only works outdoors
  - Quickly consume battery power
  - The response could be slow

- Network location provider
  - Uses cell tower and Wi-Fi signals
  - Works indoors and outdoors
  - Faster response
  - Uses less battery power

- An app can use both of them, or just one

Challenges in Determining User Location

- Sources of error in the user location include:
  - Multitude of location sources: GPS, Cell-ID, and Wi-Fi can each provide a clue to users location. Determining which to use involves trade-offs in accuracy, speed, and battery-efficiency.
  - User movement: You must account for movement by re-estimating user location every so often.
  - Varying accuracy: Location estimates coming from each location source are not consistent in their accuracy. A location obtained 10 seconds ago from one source might be more accurate than the newest location from another or same source.

- These problems can make it difficult to obtain a reliable user location reading.

Requesting Location Updates

- Getting user location in Android works by means of callback.

- You receive location updates from the LocationManager by calling requestLocationUpdates().
  - The first parameter of requestLocationUpdates() is the type of location provider to use.
  - The second is the minimum time interval between notifications
  - The third is the minimum change in distance between notifications
  - Setting both to zero requests location notifications as frequently as possible.
  - The last parameter is your LocationListener, which receives callbacks for location updates.

- Your LocationListener must implement several callback methods that the Location Manager calls when the user location changes or when the status of the service changes.
An Example of LocationListener

• To receive location updates from NETWORK_PROVIDER or GPS_PROVIDER, you must request user permission by declaring either the ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION permission.

• Without these permissions, your application will fail at runtime when requesting location updates.

• If using both NETWORK_PROVIDER and GPS_PROVIDER, then only need to request ACCESS_FINE_LOCATION permission.

– Permission for ACCESS_COARSE_LOCATION includes permission only for NETWORK_PROVIDER.

Defining a Model for Best Performance

• Location-based applications are now commonplace, but due to many factors, getting user location is complicated.

• To obtain a good user location while preserving battery power, you must define a consistent model that specifies how your application obtains the user location.

• This model includes when you start and stop listening for updates and when to use cached location data.

Flow for obtaining user location

• The typical flow for obtaining the user location is as follows:
  – Start application.
  – Sometime later, start listening for updates from desired location providers.
  – Maintain a “current best estimate” of location by filtering out new, but less accurate fixes.
  – Stop listening for location updates.
  – Take advantage of the last best location estimate.

Location Class

- float getAccuracy(): Get the estimated accuracy of this location, in meters.
- double getAltitude(): Get the altitude, in meters above the WGS 84 ellipsoid.
- float getAltitudeInDegrees(): Get the altitude, in degrees.
- long getElapsedRealtimeNanos(): Return the time of this fix, in elapsed nanoseconds since system boot.
- Bundle getExtras(): Return additional provider-specific information about the location fix as a Bundle.
- double getLatitude(): Get the latitude, in degrees.
- double getLongitude(): Get the longitude, in degrees.
- String getProvider(): Return the name of the provider that generated this fix.
- float getAccuracy(): Get the speed if it is available, in meters/second on the ground.
- long getElapsedRealtimeNanos(): Return the time of this fix, in milliseconds since January 1, 1970.

When to start listening for updates

• You might want to start listening for location updates as soon as your application starts, or only after users activate a certain feature.

• Be aware that long windows of listening for location fixes can consume a lot of battery power, but short periods might not allow for sufficient accuracy.

• You can begin listening for updates by calling requestLocationUpdates():
Using Last Known Location

- The time it takes for your location listener to receive the first location fix is often too long for users to wait.
- Until a more accurate location is provided to your location listener, you should utilize a cached location by calling `getLastKnownLocation()`.

When to stop listening for updates

- The logic of deciding when new fixes are no longer necessary might range from very simple to very complex depending on your application.
- A short gap between when the location is acquired and when the location is used, improves the accuracy of the estimate.
- Always beware that listening for a long time consumes a lot of battery power, so as soon as you have the information you need, you should stop listening for updates by calling `removeUpdates()`:

```
// Remove the listener you previously added
locationListener.removeObserver(locationListener);
```

Maintain a Current Best Estimate

- The most recent fix is not always the best.
- You should include logic for choosing location fixes based on several criteria.
- A few steps to validate the accuracy of a location fix:
  - Check if the location retrieved is significantly newer than the previous estimate.
  - Check if the accuracy claimed by the location is better or worse than the previous estimate.
  - Check which provider the new location is from and determine if you trust it more.

An example of choosing location fixes (I)

```
// Check whether the new location fix is a real one or just noise.
if (location != null) {
    // Your logic here to choose a location fix.
}
```

An example of choosing location fixes (II)

```
// Check whether the new location fix is a real one or just noise.
if (location != null) {
    // Your logic here to choose a location fix.
}
```

Adjusting the Model

- When you test your application, you might find that your model for providing good location / performance needs some adjustment.
- Reduce the size of the window
  - A smaller window in which you listen for location updates means less interaction with GPS and network location services, thus, preserving battery life. But it also allows for fewer locations from which to choose a best estimate.
- Set the location providers to return updates less frequently
  - Reducing the rate at which new updates appear during the window can also improve battery efficiency, but at the cost of accuracy. You can reduce the rate of updates by increasing the parameters in `requestLocationUpdates()`.
- Restrict a set of providers
  - You might choose to see only the Network Location Provider or only GPS, instead of both.
Case 1: Tagging contents with a location

- Create an application where user-created content is tagged with a location. A model of how this interaction might happen is visualized below.
- For best location accuracy, you might choose to start listening for location updates when users begin creating the content or even when the application starts, then stop listening for updates when content is ready to be posted.
- Need to consider how long a typical task of creating the content takes and judge if this duration allows for efficient collection of a location estimate.

Case 2: Help users decide where to go

- Create an application that attempts to provide users with a set of options about where to go.
  - For example, you’re looking to provide a list of nearby restaurants, stores, and entertainment and the order of recommendations changes depending on the user location.
  - To accommodate such a flow, you might choose to:
    - Rearrange recommendations when a new best estimate is obtained
    - Stop listening for updates if the order of recommendations has stabilized

Provide Mock Location Data

- You can test your location-based features by mocking location data in the Android emulator.
- There are two different ways to send your application mock location data:
  - Android device monitor (DDMS)
    - The “geo” command in the emulator console.
  - Providing mock location data is injected as GPS location data, so you must request location updates from GPS_PROVIDER in order for mock location data to work.

Using DDMS

- With the DDMS tool, you can simulate location data a few different ways:
  - Manually send individual longitude/latitude coordinates to the device.
  - Use a GPX file describing a route for playback to the device.
    - GPX, or GPS eXchange Format is an XML schema designed as a common GPS data format for software applications.
  - Use a KML file describing individual place marks for sequenced playback to the device.
    - Keyhole Markup Language (KML) is an XML notation for expressing geographic annotation and visualization within Internet-based, two-dimensional maps and three-dimensional Earth browsers.

Using "geo" in Emulator Console

- Launch your application in the Android emulator and open a terminal/console in your SDK’s /tools directory.
- Connect to the emulator console:
  - `telnet localhost <console-port>`
  - Send the location data:
    - `geo fix` to send a fixed geo-location. This command accepts a longitude and latitude in decimal degrees, and an optional altitude in meters. For example:
      ```bash
      geo fix -121.45356 46.51119 4392
      ```
    - `geo nmea` to send an NMEA 0183 sentence. This command accepts a single NMEA sentence of type "$GPGGA" (fix data) or "$GPRMC" (transit data). For example:
      ```bash
      geo nmea $GPGGA,016260.000,4651.131900,N,12145.356000,W,000.0,000.0,000.0,000.0,000.0*68
      ```

Location with Google APIs

- Google API Client
  - Provide access to the Google APIs in the Google Play services (e.g., Google Sing-In, Games, or Drive)
  - Entry point to all Google Play services
  - Manage network connection between user device and Google service
Location with Google APIs

- **Google API Client**
  - Create a `GoogleApiClient` object:
    ```java
    public class MyActivity extends FragmentActivity implements OnConnectionFailedListener {
        GoogleApiClient mGoogleApiClient
        = new GoogleApiClient().addApi(LocationServices.API)
            .addConnectionCallBacks(this)
            .addOnConnectionFailedListener(this)
            .build();
        mGoogleApiClient.connect();
    }
    ```

- **FusedLocationProviderApi**
  - Automatically choose location provider
  - Provide the latest and most accurate location
  - Configure the update interval
    ```java
    LocationRequest mLocationRequest = LocationRequest.create();
    mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
    mLocationRequest.setInterval(10000);
    mLocationRequest.setFastestInterval(5000);
    ```
  - Link with the `GoogleApiClient`
    ```java
    LocationServices.FusedLocationApi.requestLocationUpdates(mGoogleApiClient, mLocationRequest, this);
    ```

- **LocationAddress**
  - Convert GPS coordinates to addresses
  - Geocoder class
    ```java
    Activity
    location, Geocoder
    send(Address, Bundle)
    onReceiveResult(int, Bundle)
    ```

- **Geofence**
  - Define a set of regions, and get notified when you enter or exit them
    - (latitude, longitude, radius)
    - 3 transitions
    ```java
    private GeofencingRequest getGeofencingRequest() {
        GeofencingRequest.Builder builder = new GeofencingRequest.Builder();
        builder.addAllGeofences(mGeofenceList);
        return builder.build();
    }
    ```
Location with Google APIs

- **Geofence**
  - Handle the Geofence events
  - IntentService is a common choice
  - Launch background work
  - Post a notification
  - Prepare a PendingIntent

```java
public class MainActivity extends FragmentActivity {
  ...
  private PendingIntent getGeofencePendingIntent() {
    // Reuse the PendingIntent if we already have it.
    if (mGeofencePendingIntent != null) {
      return mGeofencePendingIntent;
    }
    Intent intent = new Intent(this, GeofenceTransitionsIntentService.class);
    // We use FLAG_UPDATE_CURRENT so that we get the same pending intent back when:
    // adding a PendingIntent, and removing it.
    return PendingIntent.getService(this, 0, intent, PendingIntent.FLAG_UPDATE_CURRENT);
  }
}
```

Location with Google APIs

- **Geofence**
  - Add geofence (via Google API service)
  - PendingResult and ResultCallback

```java
public abstract void onResult(R result)
Called when the Result is ready.
```

```java
public class MainActivity extends FragmentActivity {
  ...
  LocationServices.GeofencingApi.addGeofences(mGoogleApiClient,
      getGeofencingRequest(),
      getGeofencePendingIntent()()
      .setResultCallback(this);
}
```