Performance issues in writing Android Apps

Octav Chipara
## The process of developing Android apps

| Problem Definition | Focus: Define the problem  
|• Define the problem  
• What is the input/output?  
• What is the criteria for success? |
| Develop Basic Data Collection | Focus: On accuracy not on energy  
|• On accuracy not on energy  
• How do you know you got it right? |
| Develop an Algorithm | Focus: Solve the sensing problem  
|• Solve the sensing problem  
• How do you measure the algorithm’s performance  
• What factors affect your algorithm design/choice? |
| Integrate on Platform | Focus: Deal with the system challenges |
The process of developing Android apps

- Problem Formulation
- Data Collection
- Algorithm Design
- Integrate on Platform

1. Refine problem formulation
2. Refine experimental design
3. Initial experimental design
4. Initial design
Develop a Step-Counter Application
Problem definition

• Develop an application that counts the number of steps a user takes using a mobile phone using acceleration sensors

• Problem formulation:
  • input: acceleration samples
  • output: step counts
  • goal: determine $f : \text{acceleration} \Rightarrow \text{step counts}$
  • confounding factors?

• Requirements:
  • it must work reliably for a large population of users
  • it must work reliably for different phone placements
Data collection plan

- Key concern: get an sample that covers many real-world scenarios

- Collect acceleration readings from the phone
  - from multiple users => necessary to get a diverse set of users
  - from different environments => necessary to ensure reliability
  - try to model user and environmental differences (confounding factors)
    - user variability: height, stride length
    - phone placement: enforced during data collection
    - environment: floor type, shoe type, etc.
Building the data collection application

- Focus: on data quality rather than application optimization

- Minimal data collection requirements:
  - collect the data from the accelerometer
  - save on disk
  - straightforward?! 

- How will you validate that the data collected is reasonable?
Motion sensors on Android

- **Sensor motion API is straight forward:**

```java
... initialize ...
SensorManager _sensorManager =
(SensorManager)_context.getSystemService(android.content.Context.SENSOR_SERVICE);
sensorManager.registerListener(this,
    _sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER),
    SensorManager.SENSOR_DELAY_FASTEST);

... get readings ...
public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() != Sensor.TYPE_ACCELEROMETER) return;
}
```

- **For best performance you should keep the CPU awake!**
Power Management API on Android

- Android uses power-locks to manage power
  - use carefully => it significantly affects the power consumption of your phone

```java
PowerManager _powerManger = (PowerManager) context.getSystemService(Context.POWER_SERVICE);
_wakeLock = _powerManger.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
        "accelerometer");

@Override
public void onStart() {
    _wakeLock.acquire();
}

@Override
public void onStop() {
    _wakeLock.release();
}
```

Why did I put the release of the wake lock onStop() rather than onPause()?
Sensor loops

- **Sensor loops are performance critical routines you cannot do too much**
  - onSensorChange() is called by Android’s sensor thread
  - long processing times onSensorChange() will result in DROPPED values!

```java
public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() != Sensor.TYPE_ACCELEROMETER) return;

    ...
    open file
    save event to file
    close file
    ...
}
```
Sensor loops

- File IO is generally slow and costs lots of energy!

```java
public void onCreate() {
    open file
}

public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() != Sensor.TYPE_ACCELEROMETER) return;

    ... save event to file ...

    }

public void onStop() {
    close file
}
```
Sensor loops

```java
public void onCreate() {
    open file
}

public void saveToFile (SensorEvent event) {
    save event to file
}

public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() != Sensor.TYPE_ACCELEROMETER) return;

    ... saveToFile(event);
    ... 
}

public void onStop() {
    close file
}
```

Still executed within the same thread!
A multi-thread approach

• **Multi-threaded approach:**
  • you have to create a new thread to hand the saving to disk
  • => decouples the data collection from the writing on disk (that takes long time)

• **This is called a producer-consumer problem**
  • challenge exchange data between threads in a thread-safe manner
class Producer implements Runnable {
    private final BlockingQueue queue;
    Producer(BlockingQueue q) { queue = q; }
    public void run() {
        try {
            while(true) { queue.put(produce()); }
        } catch (InterruptedException ex) { ... handle ...}
    }
    Object produce() { ... }
}

class Consumer implements Runnable {
    private final BlockingQueue queue;
    Consumer(BlockingQueue q) { queue = q; }
    public void run() {
        try {
            while(true) { consume(queue.take()); }
        } catch (InterruptedException ex) { ... handle ...}
    }
    void consume(Object x) { ... }
}
class Setup {
    void main() {
        BlockingQueue q = new SomeQueueImplementation();
        Producer p = new Producer(q);
        Consumer c1 = new Consumer(q);
        Consumer c2 = new Consumer(q);
        new Thread(p).start();
        new Thread(c1).start();
        new Thread(c2).start();
    }
}
Back to our problem

**producer:**

```java
BlockingQueue _queue;

public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() != Sensor.TYPE_ACCELEROMETER) return;

    MyEvent mEvent = new MyEvent();
    copy event fields into mEvent

    _queue.put(mEvent);
}
```

**consumer:**

```java
public void run(SensorEvent event) {
    try {
        while(true) {
            ...
            event = _queue.take();
            save to disk (event)
            ...
        }
    } catch (InterruptedException ex) { ... handle ...}
}
```

this will cause the garbage collector to fire very often

=> loose data
Preallocate memory

**producer:**
BlockingQueue _queue;
BlockingQueue _memoryQueue = add N mEvents

```java
public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() != Sensor.TYPE_ACCELEROMETER) return;

    MyEvent mEvent = _memoryQueue.takes()
    copy event fields into mEvent

    _queue.put(mEvent);
}
```

**consumer:**
```java
public void run(SensorEvent event) {
    while(true) {
        ...
        event = _queue.take();
        save to disk (event);
        _memoryQueue.put(event);
        ...
    }
}
```
Check the data collection prototype

• Metrics to ensure that it works
  • no gaps in data, possible through visual data inspection
  • plot the distribution of the time differences between consecutive samples (should be relatively narrow)
A few other tips on writing performance code