Analysis of Performance and Energy Cost of a Cloud-enabled SWAN System

Author: Roshan Bharath Das

Supervisors:
Prof. Henri Bal
Dr. Stefania Costache
Dr. Aart van Halteren

A thesis submitted in fulfilment of the requirements for the degree of Master’s in Parallel and Distributed Computer Systems in the Faculty of Sciences Vrije Universiteit Amsterdam

October 2014
## Contents

1 Introduction 1

2 Related Work 4

3 Experimental Setup 6
   3.1 SWAN - Sensing With Android Nodes 6
   3.2 Advantages of SWAN 6
   3.3 Going beyond SWAN 8
   3.4 Cloud-enabled SWAN system 8
      3.4.1 Base Service 9
      3.4.2 Data Processing Engine 9
      3.4.3 Web Service 10
      3.4.4 Cloud Infrastructure 11
   3.5 GPS-based Application 11
   3.6 Calculating Energy 13
      3.6.1 Power Profile 14
      3.6.2 Calculating CPU clock frequency 14
      3.6.3 Effect of Voltage 16
      3.6.4 Calculating Energy Consumption 16

4 Performance and Energy Evaluation 18
   4.1 SWAN versus Direct Sensor Readings 19
   4.2 Evaluating Computation 21
      4.2.1 Type of Tasks 21
   4.3 Evaluating Data Offloading 23
      4.3.1 Network Mode 25
      4.3.2 Dataset Size 25
      4.3.3 Frequency of Data Sent 27
      4.3.4 Round Trip Time 27
   4.4 Understanding Energy 28

5 Conclusion and Future Work 30
   5.1 Conclusion 30
   5.2 Future Work 31

Acknowledgements 32
Contents

Bibliography 33
Chapter 1

Introduction

Smartphones are becoming ubiquitous. Along with the extensive use of computation and communication in the device, embedded sensors such as accelerometer, gyroscope, magnetometer and GPS are also gaining importance. A multitude of context-aware applications has been created using sensors. These applications are becoming popular in a variety of domains such as health care[1], social networks[2], safety, transportation[3] and environmental monitoring[4].

In order to support the growth of such sensor based applications, it has become necessary to build an efficient framework that can help application developers to easily build context-aware applications. To this extent the SWAN[5] framework was developed. It eliminates storage and computational redundancy by allowing third-party applications to register sensor expressions. Once the expression is registered, the sensor data collection and evaluation are done by the SWAN framework and only the results are sent back to the registered applications.

As we are storing and processing sensor data in the device, it is important to realize the amount of data we are dealing with. An accelerometer application at the fastest frequency gives 100 sensor data readings per second. If multiple such sensing applications run on a daily basis, the data generated in a device can grow up to several gigabytes. We could think of deleting this data after processing, but some of the sensor based applications could be interested in saving history data for offline analysis. For example, computing the road traffic statistics for the past year given all the paths followed by multiple vehicles in a particular city in the past year. Storing and processing such big data becomes a problem since the device has limited computing power, storage and battery. Hence, we could benefit from a cloud solution for dealing with such a huge amount of sensor data.
Since SWAN supports data processing and storage only in the device, we propose Cloud-SWAN, an extension to SWAN which also enables cloud storage and processing. The decision of where to store and process data is left to the application developer. With such a cloud solution which can be used to offload data and computation, we might be able to save one of the most scarce resource in a mobile phone, namely battery.

It would be interesting to see if Cloud-SWAN has an impact on improving battery life. Hence, we focus our research on the following questions:

- Is SWAN an energy efficient system? - We analyze the performance and energy cost of using SWAN compared to using sensors directly, at various frequencies of fetching sensor data.

- What is the impact of computation on energy? - We analyze the performance and the energy cost of performing simple and complex computations on the device based on real world scenarios. We also analyze the energy cost incurred when increasing the frequency of the computation based on the assumption that real time sensor based applications will need to show results of computation in real time.

- What is the energy cost for offloading data to the cloud? - We analyze the impact of network technologies like 3G and 4G on the energy consumed for data offloading. We evaluate the impact of sending sensor data in batches on the energy consumed. We also check the effect of sensor dataset size on the performance and energy cost.

The list of contributions to this thesis is given below.

- Cloud-SWAN - We built a cloud-enabled SWAN system that can support data processing and storing on both the device and the cloud.

- GPS sensor based bike application - We built a GPS sensor based bike application using Cloud-SWAN that calculates the distance and instantaneous speed in real time. We use this application as a real case scenario to understand the amount and type of sensor data generated for real time processing and offloading.

- Accelerometer applications - We built two accelerometer applications, one using SWAN and the other using the direct sensor API, to test the energy efficiency of the SWAN system.

- A mechanism to calculate the energy consumed by the CPU in a multi-core architecture. To this extent, we built the following applications.
Power profile application - We built a power profile application to identify the current consumed by multiple hardware components in a smartphone.

State of charge versus Voltage application - We also built an application to identify if the voltage level would remain constant when there is a change in the battery consumption.

For our analysis, we used SURFnet provided smartphones with 4G network powered by KPN.

The test results show that complex tasks run at higher CPU frequency and lead to higher energy consumption. The number of task executions is another important factor when dealing with real time applications. In case of data offloading for offline analysis, the 4G mode consumes lower energy and has lower round trip time than the 3G mode. Also, sending data in batches to the cloud is preferred.

The remainder of this thesis is organized as follows. In Chapter 2 we discuss the related work on the evaluation of the Cloud-based mobile sensor framework and the types of mechanisms used to calculate energy. In Chapter 3 we talk about the experimental setup we built for our analysis such as Cloud-SWAN, GPS based sensor application and energy calculation method for Android. In Chapter 4 we discuss the performance and energy evaluation for three cases: SWAN versus sampling sensors directly, on computational task and on data offloading. We conclude with future work in Chapter 5.
Chapter 2

Related Work

We classify the related work done into two types as shown below.

Evaluation on cloud-based mobile sensor framework

Lipyeow et al.[6] built a cloud-based coordination service for large-scale, continuous mobile sensing. The focus of their research was on identifying the energy benefits, for different sensors, of offloading the sensing or computational tasks to an alternate device. The offloading experiments were conducted using Wi-Fi mode. We are dealing with real time applications with device movement involved, so we focus our research on using 3G and 4G networks for our experiments on data offloading.

Niranjan et al.[7] studies the energy consumption characteristics of offloading data based on the dataset size, using network technologies such as 3G, GSM and Wi-Fi. They also provide insight regarding the tail energy overhead in 3G and GSM. However, no comparison is made with the 4G network technology.

MAUI[8] is a system that enables fine-grained energy-aware offloading of mobile code to the infrastructure. It compares energy consumption caused by 3G and Wi-Fi for code offloading. Since we have sensor data to be offloaded from the device to the cloud, our focus is on the energy consumption caused by data offloading rather than code offloading.

Junxian et al.[9] discuss the performance and power characteristics of 4G LTE networks on general mobile applications. We focus our analysis of 4G LTE networks on sensor based real time applications.
Mechanism used for calculating energy

NEAT [10] is an energy-analysis toolkit that allows fine-grained monitoring and analysis of energy consumption in a portable way. To get more accurate results, it uses a combination of portable power metering board and a software tool that annotates the power traces from the metering board. Since the hardware is not available in the market and as we are aware about the hardware cost incurred on other traditional power monitoring tools, we focus our research on software-based power monitoring.

Android’s built-in battery information[11] gives details about the percentage of battery used, but it does not provide a fine-grained power profile. Since we are dealing with analyzing the energy consumed by a particular computational task, we need to get more details about the energy consumed by specific hardware components like a CPU. Dong and Zhong proposed Sesame[12], an automatic smartphone power modeling scheme using a built-in current sensor. Their work was focused on overall system power rather than power analysis on individual hardware components.

PowerTop [13] is a Linux tool used to diagnose issues with power consumption and management. But this tool is not available for smartphones. Trepn Profiler[14] is a hardware sensor-based power profiler, but it is only available on Qualcomm based chipsets.

PowerTutor[15] is a more fine-grained, state-of-the-art technique which uses different methods to access usage statistics from procfs[16] and BatteryStat[11] for each hardware component. AppScope[17] uses standard kernel functionalities to collect hardware usage information through an event-driven mechanism. Although both of them are fine-grained methods for calculating energy, they do not support multi-core architectures.

Eprof [18] is also a fine-grained energy profiler for smartphone applications. Eprof has the ability to analyze the asynchronous energy state of an application, modeling the energy characteristics of hardware components such as power state of 3G modem, tail state of GPS chip. The disadvantage is that Eprof requires modifications in the Android framework to trace the API calls. Also, the application code may need to be modified in case it uses Android NDK.

Our method differs from the above mentioned in that it is used to specifically calculate energy consumed by the CPU and it supports multi-core architecture. We focus our research on calculating the energy consumed by the CPU as we are interested in precisely identifying the energy consumption caused by real-time computations. We will discuss it in more detail in the next Chapter.
Chapter 3

Experimental Setup

In this chapter we will discuss SWAN and its advantages. We will then explain what SWAN was lacking and introduce Cloud-SWAN, an extension to SWAN. Further we will discuss a GPS-based real time application developed using Cloud-SWAN. Also, we will introduce a mechanism used to calculate energy consumption caused by CPU in multi-core architectures.

3.1 SWAN - Sensing With Android Nodes

SWAN is a framework used to build context aware applications. Context aware applications will register sensor expressions using the SWAN API. These sensor expressions are constructed using a domain specific language, named SWAN-Song that allows application developers to easily deal with sensors on a high level. The evaluation engine in the SWAN service will parse these expressions and invoke related services. On receiving a new sensor reading from the hardware sensor, the evaluation engine will broadcast the update to those applications that have registered receivers for the event. Figure 3.1 shows SWAN’s architecture.

3.2 Advantages of SWAN

SWAN has the following advantages.

**Easy to Use** - As an application developer it becomes easy to use SWAN. SWAN-Song relieves developers from dealing with how to use a particular sensor API. It becomes even easier when an application uses multiple sensors. SWAN-Song
also supports Tri-State expressions where app users can register a combination of multiple expressions to get a result. For example, the expression below checks whether in the past hour the screen was off and the battery percentage difference was more than 25%. This expression can be used to check if the battery consumption is caused due to any process running in the background.

$$\text{screen : on\{ALL, 1h\} == false \&\& (battery : level\{MAX, 1h\} - battery : level\{MIN, 1h\}) > 25}$$

Eliminating Redundancy - Using the Android API, multiple applications will have to register multiple listeners even though they are using the same sensor. SWAN improves the performance of the system by registering only one listener when multiple applications want to use the same sensor.

Distributed Sensing - SWAN supports distributed sensing, allowing multiple devices to cooperate in order to provide necessary sensor data for an application. For example, a device that has a temperature sensor can send its weather data to a nearby device that doesn’t have a temperature sensor. SWAN uses the Near Field Communication mechanism to validate the identification and Google Cloud Messaging mechanism to send data.
3.3 Going beyond SWAN

SWAN enables on-device sensor data processing. But as we have seen earlier, an accelerometer application gives sensor data at the fastest frequency of 100 sensor data readings per second. When we have multiple applications that use sensors with such high frequency, storing and processing data from all the sensors becomes a tedious task since devices have limited compute power, memory and battery. Also, there can be applications that need sensor data from multiple devices for offline big data analysis. It is also important to have a data backup mechanism in case memory gets corrupted. For these reasons it is important to extend SWAN to support Cloud computing.

SWAN does not provide application developers with a storage option. The current architecture supports storing sensor data in the local database or in memory, but the decision on where to store data is made at the implementation level. So it would be helpful for the developers if they are given the option to set where the data should be stored at runtime.

Also, when we analyze the functionality of multiple sensors, we could categorize some of the functions of a particular sensor. For example, GPS sensors give us the latitude and longitude of a particular location based on the timestamp. Using this information we can calculate the distance, speed etc. So, to improve the application developers experience, it would be good to have a layer of abstraction where the data processing based on sensors can be handled both on the device and the cloud.

To this extent we introduce the Cloud-enabled SWAN system which tries to solve the above mentioned problems. The next section discusses the architecture of the Cloud-enabled SWAN system.

3.4 Cloud-enabled SWAN system

The Cloud-enabled SWAN system, also known as Cloud-SWAN is a framework that supports sensor data storage and processing on both the device and the cloud. Cloud-SWAN runs in a separate process. Multiple applications can send messages/requests to Cloud-SWAN using Android’s inter process communication [19] mechanism. We will describe each component of Cloud SWAN in the next section. Figure 3.2 shows the architecture of Cloud SWAN.
3.4.1 Base Service

The base service is responsible for handling messages sent by the application and sending back the response to the application. Based on the type of message sent, the base service will interact with the different components of Cloud-SWAN.

Some examples of the messages an application can request to the base service are shown in Table 3.1.

3.4.2 Data Processing Engine

The data processing engine is responsible for processing sensor data based on the input from the base service. Some of the important functionalities of this module include sending data to the cloud in batches, getting new sensor data from SWAN and storing it in the local database, doing sensor specific functionalities like calculating distance and
speed in case of GPS based expressions. The main purpose of building these functionalities is to compare energy consumption when sending data to the cloud on every update from sensors or in batches, varying the frequency and the amount of data. Also, it is interesting to know the performance cost on storing data in the cloud.

### Table 3.1: Examples of message request to Cloud-SWAN

<table>
<thead>
<tr>
<th>Type of Messages</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPS REGISTER_EXPRESSION_IN_SWAN</strong></td>
<td>A message to register a given expression in SWAN.</td>
</tr>
<tr>
<td><strong>GPS_SAVE_SENSOR_DATA_IN_CLOUD</strong></td>
<td>Message to save newly received GPS sensor data to cloud.</td>
</tr>
<tr>
<td><strong>GPS_SAVE_SENSOR_DATA_IN_LOCAL</strong></td>
<td>Message to save newly received GPS sensor data in the local database.</td>
</tr>
<tr>
<td><strong>GPS_SEND_SENSOR_DATA_FROM_LOCAL_TO_CLOUD</strong></td>
<td>Message to send latest sensor data stored in local database to cloud.</td>
</tr>
<tr>
<td><strong>GPS_RECEIVE_SENSOR_DATA_FROM_CLOUD_TO_LOCAL</strong></td>
<td>Message to receive latest sensor data from cloud to local database.</td>
</tr>
<tr>
<td><strong>GPS_CALCULATE_DISTANCE_IN_DEVICE</strong></td>
<td>Message to do data processing in the local device</td>
</tr>
<tr>
<td><strong>GPS_CALCULATE_DISTANCE_IN_CLOUD</strong></td>
<td>Message to do data processing in the cloud.</td>
</tr>
</tbody>
</table>

#### 3.4.3 Web Service

We use a web service module that runs in the background and exchanges data between the device and the cloud using the HTTP protocol. The data is sent and received in JSON format. The example below shows a POST request /response with sensor data reading received at the cloud from the device with IP address 145.108.76.47. The JSON data contains two fields, the request_state field contains the type of operation to be done in the cloud, and the data field contains the data to be stored in the cloud database. In the example below, the device sends 6 location based data in a batch to the
cloud. In the first row of data field, \textit{ce3184f8139636e0} represents the device id, 52.3349 represents latitude, 4.86747 represents longitude, 1407508946 shows the timestamp and 0 is a validation parameter to confirm whether the data is stored in the cloud.

\begin{verbatim}
145.108.76.47 - -[08/Aug/2014:41:36]"POST/HTTP/1.1"200-
{"data": "[ce3184f8139636e0 52.3349 4.86747 1407508946 0,
  ce3184f8139636e0 52.3349 4.86747 1407508937 0,
  ce3184f8139636e0 52.335 4.86785 1407508926 0,
  ce3184f8139636e0 52.335 4.86785 1407508916 0,
  ce3184f8139636e0 52.335 4.86785 1407508907 0,
  ce3184f8139636e0 52.335 4.86798 1407508896 0],"
, "request_state": "REQUEST_SEND_DATA_TO_CLOUD_DATABASE"} (3.1)
\end{verbatim}

\section*{3.4.4 Cloud Infrastructure}

We use the DAS-4\cite{20} cluster as our cloud infrastructure. We run a http server that handles requests from devices and based on the type of request, it will either process the data or save the data in the cloud.

We chose Apache Cassandra\cite{21} database solution as our storage option in the cloud. Apache Cassandra is a NoSQL\cite{22} solution typically categorized into a column store like HBase and BigTable. As we are dealing with sensor data based on time series and as we can store variable size data in a row in runtime we chose a column store solution. Cassandra allows for setting a replication factor which indicates how often each individual data should be replicated. In terms of horizontal scalability, Cassandra allows for seamless addition of nodes to the clusters and the cluster will utilize the new resource automatically.

\section*{3.5 GPS-based Application}

We built a GPS based real time biking application that uses Cloud-SWAN. The application gives real time information about the distance travelled and the instantaneous speed based on the latitude and longitude information from the GPS sensor and timestamp. A screenshot of the application is shown in figure 3.3. Figure 3.4 shows the route...
travelled in Google maps. This application was used by a colleague for the project on improving the lifestyle of citizens in the municipality of Alkmaar.

On pressing the Start Tracker button, the application tries to bind to Cloud-SWAN and sends standard messages defined by Cloud-SWAN using the messenger. The base service receives these messages and registers an expression for GPS in SWAN. With every new value received from the sensor, SWAN will notify the base service which will get this value and store it in the heap or local database based on the request from the application. The application will also send a message to the base service for enabling data processing, like calculating distance or speed in real time. To improve battery efficiency we enable GPS requests every 10 seconds and instead of sending data to the cloud with every update from the sensor, the base service sends sensor data in batches every minute. To calculate precise real time distance, we use the Haversine\cite{23} formula.

In the evaluation section we will discuss the impact of energy consumption based on sending data in batches and also on the frequency of data sent to the cloud. We will also discuss the performance and energy cost of using Haversine formula in comparison with an approximate calculation known as Equirectangular approximation.

Figure 3.5 shows State of charge versus Time graph when the application was used. As can be seen, the rate of change of state of charge is constant. We checked other
applications with recurring behavior and we found out that the rate of decrease of state of charge is always constant. We use this inference to calculate the energy consumed by the application. In the next section we will discuss this in more detail.

Figure 3.5: Battery Versus Time for GPS based Application

3.6 Calculating Energy

The energy consumption caused by the CPU for an application can be calculated by checking the battery percentage difference when the application starts and stops. For example, let us assume that the initial charge percentage of the battery is 90% and let us say the battery percentage after the application runs for 10 minutes became 60%. This essentially means that there is a 30% reduction in the battery in 10 minutes. Complete discharge would happen in \((10/30)*100 = 33.33\) minutes. Then total current consumption will be 2600 (Current capacity, as can be seen in every battery) \(\times 60/33.33\) mAh. When multiplied with a constant voltage, this will give the energy consumed. We can use this method to calculate the energy consumed. As we saw earlier, the rate of change of state of discharge is constant for applications with recurring behavior. But since battery percentage is as integer, a change from one value to the other may take some time. Due to the delay in the change, a precise knowledge of energy consumed by an application is not possible. Also, battery consumption can occur due to other
factors like the LED display. So, we can get only the battery consumption caused by a combination of components and not individual ones.

When calculating energy consumed by some functionality of the application using the above method, it is important to calculate the energy consumed when the application is in standby mode, i.e. no functionality is running. In order to get a more precise value, the energy consumed in the standby mode must be removed from the overall energy caused by the application (when running with the functionality on). We use this method in our experiments to calculate energy consumed in data offloading using multiple operators. While trying to calculate the energy, we disabled all other applications and any other possibilities that can interfere.

We explain our next method to precisely calculate energy utilized by CPU, used for our experiments in data computation. This method could be extended to calculate energy consumed by other hardware components as well.

### 3.6.1 Power Profile

Power profiling is a mechanism where in the device manufacturer provides current consumption values for different hardware components when they manufacture the device. These values will vary among different devices. To get the current consumption details of each component we use the power_profile class in Android. We use the reflection mechanism to get the contents from this internal class. Using this class we built an application that can provide a power profile for a device. Figure 3.6 shows the current consumption values of various components in a Galaxy S4. For example, we can see from the figure that Wi-Fi on active mode consumes a current of 309 mA. Since we are interested in the energy consumption caused by the CPU for the computational tasks, we will focus more on it in the section 3.6.2.

### 3.6.2 Calculating CPU clock frequency

The Android Dalvik Debug Monitor Server provides a Systrace tool which uses the scaling_cur_freq (/sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq) file to get the current frequency in which a CPU core is running. Figure 3.7 shows clock frequency with respect to time, for a running application on a quad core device. The red dot shows that CPU Core 0 running at 1890000 frequency in timestamp 14760.01ms.

From the discussion above, we can calculate the current flow in a core based on clock frequency at a particular point in time. The total current consumed by the CPU will be a combination of current consumed by each core. It is also important to calculate the
Chapter 3. Experimental Setup

Figure 3.6: Power Profile for a Galaxy S4

Figure 3.7: Systrace Output
current usage when the CPU core is in idle state. In Galaxy S4, the current usage when
the core is in idle state is 4mA.

3.6.3 Effect of Voltage

We plot voltage versus state of charge graph for Galaxy S4 device as shown in figure
3.8. Although voltage is not constant, the variation is minimal. This change would be
essential when calculating energy more precisely.

![Figure 3.8: Voltage versus State of Charge](image)

3.6.4 Calculating Energy Consumption

Energy is the ability to do work. In this context we refer to energy as electrical energy
and it can be calculated as shown below.

\[
E = V \times I \times t
\]

\(E = \text{Energy, } V = \text{Voltage, } I = \text{Current, } t = \text{time}\)

Voltage as Constant :

\[
E = V \times (I_1 \times t_1 + I_2 \times t_2 + \ldots + I_n \times t_n)
\]

Where \(V = \text{Constant Voltage, } I_n \text{ is the current usage in tn time period and } E = \text{Energy}
\)

consumed in time \(t(t = t_1 + t_2 + \ldots + t_n)\)

Voltage as a Variable :

\[
E = V_1 \times I_1 \times t_1 + V_2 \times I_2 \times t_2 + \ldots + V_n \times I_n \times t_n
\]
Where $V_n$ is the Voltage and $I_n$ is the current in the time period $t_n$. We already know $I_n \cdot t_n$ from above discussion and we can calculate the voltage $V_n$ from the battery sensor based on time. Here, we consider voltage as a variable, so the energy consumption calculated will become more precise.

In the next Chapter we will see the performance and energy evaluation based on this setup.
Chapter 4

Performance and Energy Evaluation

In this chapter we conduct our evaluation based on our research questions. We divide it into three main parts: the overhead of SWAN, the costs of computation, and the costs of data offloading.

In table 4.1 we describe the components used for the evaluation on performance and energy cost of a sensor based application.

<table>
<thead>
<tr>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Galaxy S4</td>
</tr>
<tr>
<td>CPU</td>
<td>1.9 GHz, Quad-core Snapdragon 600</td>
</tr>
<tr>
<td>Memory</td>
<td>2 GB LPDDR3 RAM</td>
</tr>
<tr>
<td>WiFi</td>
<td>802.11a/b/g/n/ac</td>
</tr>
<tr>
<td>3G Network</td>
<td>HSDPA 850/900/1900/2100</td>
</tr>
<tr>
<td>4G Network</td>
<td>LTE 800/850/900/1800/2100/2600</td>
</tr>
<tr>
<td>Sensors</td>
<td>Accelerometer, Gyroscope, Proximity, Compass, Barometer, Temper-</td>
</tr>
<tr>
<td></td>
<td>ture, Humidity, Gesture, GPS</td>
</tr>
<tr>
<td>Network Provider</td>
<td>KPN</td>
</tr>
<tr>
<td>Cloud</td>
<td>DAS4 with httpserver and Apache Cassandra database</td>
</tr>
</tbody>
</table>

Table 4.1: Components used for testing
4.1 SWAN versus Direct Sensor Readings

Since we are using SWAN for our application, it is important to understand the performance and energy cost of using SWAN. To this extent we use two accelerometer applications; an application using SWAN and another directly registering to the sensors. We chose to perform our tests using the accelerometer sensor since it has a higher frequency of sensor reading. The results will be more reliable compared to using a GPS sensor since the frequency of sensor data reading received from a GPS sensor would depend more on location (outdoor or indoor).

The first evaluation was done on two standard frequencies provided by Android, namely SENSOR_DELAY_FASTEST and SENSOR_DELAY_NORMAL. As the name itself suggests, SENSOR_DELAY_FASTEST gives the sensor data reading at a higher rate compared to SENSOR_DELAY_NORMAL. In figure 4.1, WithSwanNormalFirst shows the time taken to send a register expression request from the application and to get back the first result from SWAN. The ‘First’ in WithSwanNormalFirst indicates that SWAN is running for the first time after installation. WithSwanNormalSecond shows the time taken to register the expression when SWAN is already running in the background. WithOutSwanNormal shows the time taken to register an accelerometer listener using the Android API and get the result. As we can see from the graph, the application using SWAN takes more time using WithSwanNormalFirst. The application WithSwanNormalSecond has improved performance compared to Application WithSwanNormalFirst. The reason for this difference is that the first time there is an overhead in SWAN to start the service in a different process. From the second time onwards, since SWAN already runs in the background, applications just needs to bind to the already running service. We also note that the application WithSwanNormalSecond is three times slower than WithoutSwanNormal. The delay is due to the time taken for inter process communication when using SWAN. The results obtained are similar when using SENSOR_DELAY_FASTEST frequency. We note that WithoutSwanFast takes almost half the time compared to WithoutSwanNormal, but the time taken by WithSwanFastSecond and WithSwanNormalSecond is similar due to the overhead on SWAN.

We plot an energy versus time graph to compare the accelerometer application using SWAN and directly registering listeners using Android API shown in the figure 4.2. The total duration of calculation was 1 minute and the tests were done 5 times. Since the variation in the result was minimal we calculated the average energy of all the tests. As can be seen from the result, applications using SWAN consume more energy as compared to applications directly registering listeners using the Android API. The reason for this overhead in SWAN is due to the evaluation engine that continuously evaluate expressions.
Chapter 4. Performance and Energy Evaluation

Figure 4.1: Processing Time for the Accelerometer Application

Figure 4.2: Energy Consumption for the Accelerometer Application
We presume that when multiple applications start using SWAN, it will become more energy efficient because it eliminates computational redundancy (very energy expensive) which would occur when multiple applications perform the same local evaluations without using SWAN. Also, it is interesting to note that the energy consumed increases when the frequency of sensor reading increases. So if accuracy is not important, applications should tune to using a lower frequency when using accelerometer sensor.

4.2 Evaluating Computation

In this section we evaluate the energy consumed by a CPU when running multiple tasks. As we are dealing with real case scenarios, we take possible computation on a GPS based application into consideration. One of the main and recurring computations in a GPS based application is to calculate distance and speed. We classify calculating distance given latitude and longitude into two types: simple distance calculation and complex distance calculation. We then analyze the throughput and energy cost of running this computation. Based on the results, we increase the complexity of the task by calculating the shortest path, given the number of paths (n) and number of points (m). The complexity of the algorithm is O(n*m). In the next section we describe the type of tasks in more detail.

4.2.1 Type of Tasks

In this section we describe three types of tasks.

Simple Distance Calculation - Equirectangular approximation

This is an approximate calculation method for calculating distance given latitude and longitude of two points. We show the formula for Equirectangular approximation. It uses the Pythagoras theorem on an equirectangular projection.

\[
x = (\text{lon}2 - \text{lon}1) \times \text{Math.cos}((\text{lat}1 + \text{lat}2)/2)
\]
\[
y = (\text{lat}2 - \text{lat}1)
\]
\[
d = \text{Math.sqrt}(x \times x + y \times y) \times R
\]

Where,

- \text{lat}1 = \text{Latitude of point 1}
- \text{lat}2 = \text{Latitude of point 2}
- \text{lon}1 = \text{Longitude of point 1}
lon2 = Longitude of point 2
R = Radius of earth (6371 km)

If accuracy is not very important, we can use this method to calculate the distance.

**Complex Distance Calculation - Haversine formula**

To calculate distance more accurately, we use the Haversine method. This formula is complex compared to the Equirectangular approximation since it uses more trigonometric functions and more square root functions. We show below the formula for Haversine method.

\[
\begin{align*}
lat_{diff} &= (lat_2 - lat_1).toRadians() \\
long_{diff} &= (lon_2 - lon_1).toRadians() \\

a &= Math.sin(lat_{diff}/2) \times Math.sin(lat_{diff}/2) + \\
&\quad Math.cos(lat_1) \times Math.cos(lat_2) \times Math.sin(lat_{diff}/2) \times \\
&\quad Math.sin(long_{diff}/2) \\
c &= 2 \times Math.atan2(Math.sqrt(a), Math.sqrt(1 - a)) \\
d &= R \times c
\end{align*}
\]

Where,
- \(lat_{diff}\) = latitude difference of two points
- \(long_{diff}\) = longitude difference of two points
- \(R\) = Radius of earth

**Shortest Path Calculation**

When we compared the throughput of the above mentioned tasks, we identified that the difference is small. So, we took another real case scenario, namely Shortest Path calculation. This task is more complex compared to calculating distance given latitude and longitude. Figure 4.3 shows the throughput comparison of different tasks running for 1 minute.

We calculate the shortest distance given the number of paths(n) and number of points(m) in a path by calculating the total distance between the points in each path using Haversine formula and then calculating the shortest distance comparing multiple paths. The algorithm runs with a complexity of \(O(n^*m) + O(n)\). In figure 4.3 we can see that the throughput difference is much higher when we compare the haversine method with shortest distance task given 10 paths and 10 points in each path.
Figure 4.3: Throughput measure of different tasks

Next, we plot the energy cost comparison between multiple tasks as shown in figure 4.4. To calculate the energy cost, we keep the number of occurrences constant. From the graph, we can see that there is not much difference in the energy consumption when calculating distance using the Haversine formula or Equirectangular method. But, when we compare energy consumption between calculating distance and calculating shortest distance, we find that there is a big difference in the energy consumption since calculating shortest distance consumes CPU at a higher clock frequency. So, it is worth to offload the task computation if the given task is complex.

We also plot a graph to check the energy consumption with respect to the number of occurrences of a particular task. To calculate this, we increase the number of paths to 1000 and the number of points in a path to 1000. The total time duration for this test is 1 minute. Figure 4.5 shows the plot of energy consumption with respect to the number of occurrences. We note that with the increase in the number of occurrences there is an increment in energy consumption as well.

4.3 Evaluating Data Offloading

As we have seen in the previous section, the task complexity has an impact on the energy consumed. For example, in the use case we present, consider that the bike application tries to compute the shortest path from the current location to Alkmaar. Assume that
Figure 4.4: Energy Consumption of different tasks

Figure 4.5: Energy Consumption versus Number of Occurrences
there are 10000 paths to reach Alkmaar and that every path has around 10000 coordinates. We are then dealing with a complex task. To make this task more challenging, let us assume that while biking the user needs updated information about the shortest path every 10 seconds, taking traffic congestion into account. When the complexity of a task increases, the energy consumed also increases. It would be interesting to compute complex tasks in the cloud based on the assumption that we can consume unlimited amount of energy in the cloud. However, for offloading tasks in the cloud, we need to first send the sensor data. Transmitting data will consume energy since we will be using network connection modes like 3G, 4G and Wi-Fi. In this section we will discuss about the impact of network mode, dataset size and the frequency of data offloaded on energy consumed.

4.3.1 Network Mode

We will discuss the impact of modes of network connection on the energy consumed. We use the same operator to test both 3G and 4G connections for a duration of 30 minutes. We compare the energy consumed when we send one sensor dataset every second. One sensor dataset comprises of a header, request type and one GPS sensor reading(latitude, longitude) along with a timestamp. In figure 4.6 we can see that the energy consumed by 3G is more than double the energy consumed by 4G. Wi-Fi was consuming the least energy. As we are using the GPS based bike application we can assume that the Wi-Fi access points are not available everywhere. Also we are interested in getting real time information after processing in the cloud. Hence, we focus our evaluation on 3G and 4G network connections.

4.3.2 Dataset Size

When we decide to offload data, it would be interesting to understand if there is any considerable savings in energy when sending sensor data in batches than sending data every time we receive from the hardware. We plot the energy consumed by 3G and 4G connections on multiple dataset sizes as shown in figure 4.7. We keep the frequency of data sent to the cloud constant, i.e. sending the dataset every 30 seconds for 30 minutes.

From the graph we note that the energy consumed when sending data to the cloud is comparable until 1000 dataset size. So, by sending data in batches we can save a considerable amount of energy. This information is useful if we decide to do offline analysis in the cloud. Also, we can see that using a 4G network we can save a considerable amount of energy compared to 3G.
Figure 4.6: Energy Consumption versus Modes of Network Operation

Figure 4.7: Energy Consumption versus Dataset Size
4.3.3 Frequency of Data Sent

In a real case scenario, it would be interesting to see the energy consumption caused when we change the frequency of data sent to the cloud. Figure 4.8 shows the energy consumption plotted against different frequencies for a constant dataset size of 1000.

As can be seen from figure 4.8, in the x-axis we decrease the frequency of data sent to the cloud and we note that the energy consumed decreases in most cases. We can also see that by using a 4G network we gain energy efficiency. In case of 3G, when we send data every 5 seconds, the energy consumed is more compared to sending data every second. We presume that this is due to the tail energy caused by 3G. Once in high power state, it takes around 5 seconds to go to a low power state for 3G. Interestingly there is no tail energy observed for 4G network. So, in case an application plans to send data with lower frequency to the cloud using 3G services, it is important to send data at a time interval more than 10 seconds to save energy.

4.3.4 Round Trip Time

We now analyze the round trip time taken for multiple datasets in case of 3G, 4G and Wi-Fi network mode as shown in figure 4.9. The round trip time we calculate is the time taken to send the sensor data to the cloud and get back a response from the cloud. When we receive a request in the cloud, we start a new thread to handle this request and a response is sent back to the device immediately. When comparing 3G and 4G
network, we notice that 4G has lower round trip time. Also, the round trip time is comparable when the dataset size is up to 1000. So, the performance is not affected much when sending data in batches to an optimal limit.

![Round Trip Time versus Dataset Size](image)

**Figure 4.9:** Round Trip Time versus Dataset Size

### 4.4 Understanding Energy

Given the energy consumed by an application we can directly correlate it with the time it would take to discharge the battery completely. Let us take an example to explain this more clearly.

Assume that an application consumes 1000J of energy. Let us consider that the voltage is constant = 4.2V. Then current hour of the application is \(\frac{1000 \times 1000}{4.2 \times 3600} = 66.13 \text{ mAh}\). We know that the current capacity of Galaxy S4 = 2600 mAh. Then total time the application would last till complete discharge of battery = \(\frac{2600}{66.13} = 39.31\) hours.

We will get a feeling of how much time will the battery sustain based on this information. This could be used to take the decision of doing the computation on cloud.

We will discuss the energy consumption on the GPS-based application. Let us consider the following cases.

1) Computation in the device: Assuming that the application wants to calculate the distance in real time, if we use the haversine formula every millisecond to get an accurate
calculation, the energy consumed by the CPU will be \( \frac{23.8 \times 60 \times 1000}{100000} \text{ J} = 14.28 \text{ J} \). Therefore, the total time the application would last is \( \frac{2600}{\left(\frac{14.28 \times 1000}{4.2 \times 3600}\right)} = 2753 \) hours, which would be around 115 days. We know that battery life for a smartphone is around one day. This is due to the fact that screen, network, storage and other components consume more energy. Thus, we can suggest that energy consumption caused by the CPU is much lower compared to other components.

2) Data Offloading to the cloud: Assuming that the GPS-based application sends one sensor dataset to the cloud every second using the 4G service, the energy consumed will be \( 1855.62 \text{ J} \). This means that the battery will sustain till 21 hours. When using the 3G service for the battery will last around 9 hours. If the application sends data in batches, for example 1000 dataset every 30 seconds to the cloud, then the battery life is around 71 hours. When we increase the frequency of sending 1000 dataset to every 10 seconds using the 4G service, the battery life decreases to 45 hours.

From the above, we can suggest that the computation like calculating distance using sensors can be done in the device and in case offloading needs to be done for offline analysis, sending GPS data in large batches with less frequency is preferred.
Chapter 5

Conclusion and Future Work

5.1 Conclusion

We implemented Cloud-SWAN, which extends SWAN to support a cloud infrastructure. The sensor data can be stored and processed both in the device and in the cloud using Cloud-SWAN. We built a GPS based bike application that uses Cloud-SWAN, to understand the amount of data processed and stored in a real case scenario. Further, we developed a mechanism to calculate energy consumed by the CPU in an Android device. This method supports multi-core architecture and can be extended to calculating energy consumed by other hardware components. We identified that the voltage changes based on the change in the state of charge. We argue that this can influence the calculation of energy consumption.

We analyzed the performance and energy cost of an accelerometer application using SWAN as compared to using the Android sensor API directly. We showed that an application using SWAN had improved performance when SWAN already running in the background. Overall, the application using the Android sensor API directly was performing better and was energy efficient compared to the application using SWAN. We then analyzed the performance and the energy cost of computational tasks based on real case scenario. We identified that complex tasks run at higher CPU frequency which lead to higher energy consumption. The number of occurrences of a task was directly proportional to the energy consumption. Further we analyzed the performance and the energy cost of the sensor data offloading to the cloud. We analyzed the impact of 3G and 4G on energy consumption and identified that 4G consumes lower energy and has lower round trip time than 3G mode. We also identified that sending data in batches to the cloud is preferred.
5.2 Future Work

In this work we analyzed the energy consumption for computational tasks and sensor data offloading in Cloud-SWAN using a GPS based bike application. It would be helpful to analyze more applications that use other sensors like accelerometer, gyroscope, magnetometer etc. We have analyzed the energy consumed in data offloading using only one network operator. In the future we need to test this with more network operators. In this work we also see that a lot of sensor data is saved in a local database. A calculation of the performance and energy cost on I/O operations needs to be performed. Developers can use this in order to decide where to store the sensor data by comparing the energy consumed when storing in the memory, local database or in the cloud. We also need to test the energy and performance cost when running multiple concurrent applications using SWAN because when running one application it was consuming more energy as compared to directly using sensor API. Another aspect to investigate is the application level performance when comparing local calculations versus calculations in the cloud. Even though we can choose to send data in batches to the cloud to save energy, it can lead to a significant delay that may not be suitable for real time applications. Also, our mechanism for energy calculation could be extended to support other hardware components.
Acknowledgements

We thank SURFnet for providing the 4G phones. This research was done in the context of ACBA(Amsterdam Center of Business Analytics), in collaboration with the municipality of Alkmaar. Also, the work will be used by COMMIT (a public-private research community) in the SENSEI project.

I would like to express my special appreciation and thanks to my advisor Prof. Henri Bal, you have been a tremendous mentor for me. Your advice on the research have been priceless. I would also like to thank my supervisors Dr. Stefania Costache and Dr. Aart van Halteren for reviewing my thesis and giving me valuable suggestions.

A special thanks to my family. Words cannot express how grateful I am for all of the sacrifices that you have made on my behalf. Your prayer for me was what sustained me thus far. I would also like to thank all of my friends who supported me in writing, and incented me to strive towards my goal.
Bibliography


services.html.


