October 2016

Using a Real-Time Object Detection Application to Illustrate Effectiveness of Offloading and Prefetching in Cloudlet Architecture

XuTong Zhu
The University of Western Ontario

Supervisor
Hanan Lutfiyya
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science

Abstract

In this thesis, we designed and implemented two versions of a real-time object detection application: A stand alone version and a cloud version. Through applying the application to a cloudlet environment, we are able to perform experiments and use the results to illustrate the potential improvement that a cloudlet architecture can bring to mobile applications that require access to large amounts of cloud data or intensive computation. Potential improvements include data access speed, reduced CPU and memory usages as well as reduced battery consumption on mobile devices.

Keywords: cloud and cloudlet computing, data pre-fetching, partition and offloading

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction and Motivation

Since the first release of the iPhone in 2007, the use of mobile devices (e.g., smartphones, tablets, smart watches) has increased dramatically. The recent Cisco Global Mobile Data Traffic Forecast shows that, in 2014, global mobile devices and connections grew almost half a billion (from 6.9 billion in 2013 to 7.4 billion in 2014) [1]. Cisco also noted that these mobile devices represent 88 percent of the mobile data traffic in 2014.

Despite the growth and the convenience they bring, mobile devices have limited computing power, e.g., battery power, CPU and memory. This limits the applications that can be run only on mobile devices. While mobile devices are evolving to become more powerful and able to support more complex applications, there are still applications that would have poor performance if the computing resources were limited to those found on the device. For example, weather forecasting requires access to many data sources, e.g. radiosonde observation and satellite data. Accessing this data requires that the application has many open network connections to receive the data. As shown by Lynch et al. [2], the application also requires the calculation of numerical prediction models. The access and computation requires substantial computing resources that are often not available on mobile devices. However, the leveraging of cloud computing resources allows for this application to be realized. The weather service hosted on the cloud handles communication with the data sources and numerical prediction modelling. The weather service can be accessed through an API by software on mobile devices. The mobile device software specifies a location and receives results from the cloud service (predicted weather conditions for different locations). This saves battery power and requires little compute power and storage of the mobile device. Another example is seen with social network applications (e.g., Twitter, Reddit, Facebook and Linkedin), where a huge amount of multimedia data (e.g., texts, images, audios and videos) are stored on cloud servers. Applications on mobile devices can retrieve this data through an API. In this case, cloud

servers can also perform computational services (e.g., data mining) based on the data generated through users' behavior (e.g., referencing people the user may know based on existing friends' links, suggesting pages, posts that the user may be interested in).

The use of cloud computing resources to provide services used by mobile applications has resulted in significantly increasing mobile traffic and this is expected to continue in the future. In the Ericsson Mobility Report 2015 [3], the expected smart phones subscriptions by 2020 as compared to 2014 will increase from 2.6 billion to 6.1 billion worldwide, mobile broadband from 2.9 billion to 7.7 billion and mobile PCs, tablets and routers will increase from 250 million to 400 million. Monthly mobile data traffic is expected to increase from 3.3 ExaBytes/month as of 2014 to 30.5 ExaBytes/month in 2020 [3]. We also note that during popular global events the data flow is even more significant than usual. For example, during the 2014 World Cup, there were 26.7 Terabytes of traffic from texting, talking and posting on social networks. During the championship final, generated mobile traffic was five times higher than the average peak hour traffic.

1.1 A solution to the increasing traffic incurred by mobiles - Cloudlet

To deal with this growth, one concept that has been proposed is to use middleware in cloud network structures e.g. a cloudlet. A cloudlet has richer computing resources compared to mobile devices and can be located at places where a major population lives or places that are visited at a high frequency e.g., major residential area, large restaurants, coffee stores, malls, schools. As explained in Satyanarayanan et al. [4], the role of cloudlets is to provide computing power in between cloud servers at the network core and mobile devices at the network edge. Cloudlets access cloud servers at the network core while being easily reachable by mobile devices through either a LAN or WAN connection at a closer distance to the device than cloud servers. Figure 1.1 shows the current relationship between mobile devices and a cloud while Figure 1.2 shows possible role of a cloudlet that is assumed to be at the network edge.

1.2 Problem statement

A considerable amount of research explores the potential effectiveness of cloudlets. This research can be categorized into three areas: 1) Application partitioning and offloading which determine how applications should offload computationally complex functions to

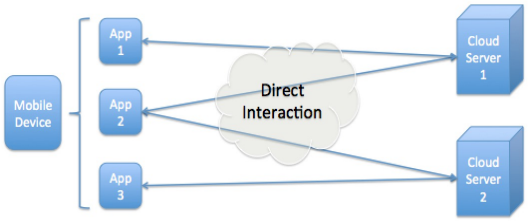

Figure 1.1: Common Cloud

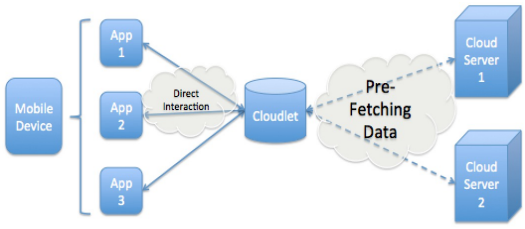

Figure 1.2: Cloudlet

cloudlets [5] [6] [7]. Typically, the goal is to improve overall performance of the application or minimize energy usage on the mobile devices through the offloading; 2) Cloudlet Structures, which focus on facilitating coordination among mobile devices, cloudlets and cloud servers [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [4]; 3) Data Pre-fetching, which studies how a cloudlet can effectively pre-fetch data from cloud servers to better serve mobile devices' needs, [19] [20] [21] [22] [23] [24] such as reduced network latency and energy consumption.

Despite the considerable research on cloudlets, little research has focused on how to improve the responsiveness of cloudlets to requests from mobile devices by pre-fetching data based on context.

In this thesis, we designed a live building detection application which involves complex computer vision computations on potentially a large base of location (context) based data that is to be processed and stored. We coded the application into two prototypes: one only uses the mobile device's local resources, and the other offloads to cloudlets. We used these prototypes to evaluate the effectiveness of prefetching of data based on context.

1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 introduces related work in these areas: 1) mobile application partitioning and offloading; 2) the different ways of using cloudlet; and

3) proposed approaches to be used by cloudlets to prefetch data from a cloud. Chapter 3 will introduce our application and cloudlet architecture design. Chapter 4 will introduce our experimental setup. Chapter 5 will present our experimental results with analysis and discussions. Finally Chapter 6 present conclusions and future work.

Chapter 2

Literature Review

This chapter reviews related research in mobile application partitioning/offloading, mobile-cloud network architecture and cloudlet pre-fetching techniques.

2.1 Mobile Application Partitioning & Offloading

In order to allow mobile applications to leverage cloud resources, decisions need to be made on how to partition and offload applications. This section describes the representative work in the area of partitioning and offloading.

2.1.1 Identify remotely executable portions in application

For applications that aim to leverage cloud/cloudlet resources, there needs to be a way to identify the remotely executable portion of the applications.

CloneCloud [6] [7] proposes a full VM (virtual machine) migration approach for the mobile device to offload computational threads to cloud servers. It uses a static analyser to identify partitions of the application automatically under a set of constraints. For example, methods that access unique features of its local device, share native states or libraries should stay local. The static analyser runs automatically instead of developers manually identifying code that can be remotely executed. We will refer to this code as remotable.

MAUI [5] uses .NET CLR, which is a cross platform run-time environment by Microsoft, that compiles and runs code from different coding sources. It enables cross platform, cross language compilation by compiling source code into CIL (common intermediate language) [25]. MAUI uses a CLR feature that allows application developers to

manually tag methods in the application source code, which indicates that the method can be remotely executed.

Cuckoo [26] proposes that developers use the Android Activity/Service Classes to separate an application's interactive part and computational part. On an Android platform, Activity and Service are two different interfaces for different functions to run. The Activity interface is the front desk interface which normally interacts with users whereas the Service interface runs in the background and is responsible for handling computationally heavy tasks. Cuckoo utilizes the two interfaces and considers methods in the Activity interface as local and methods in the Service interface as remoteable. For the service interfaces, it auto generates the remote version of the service interfaces which runs on the cloud server VM. However, the automation only generates interfaces for remoteable methods. The actual implementation of these methods on a cloud server varies and still needs developers to manually code them.

2.1.2 Profiling and Monitoring

CloneCloud [6] [7] profiles the cost of running each remotable portion of an application under different platforms based on energy consumption and CPU cycles using test runs. The test runs generate a profile tree which stores these costs and later, these are used as guidance to make partition decisions.

MAUI [5] takes a different approach. It continuously monitors program and network characteristics during runtime including the amount of state that needs to be transferred for each method offload, runtime duration and number of CPU cycles required. The monitored information is used to provide a more accurate estimate of the cost of each method's call to determine whether it should be offloaded. The profiler records each method's performance from the point it starts its execution until the end of the execution. The performance record is then analysed and used as a suggestion for the method's future invocation. For example, if the method performs poorly after being offloaded, then next time it may not be offloaded. Instead it would run locally.

2.1.3 Partitioning

In static partitioning[[6] [7] [26] [8] [9]], how the application should be partitioned is decided before application runtime. For example, in CloneCloud [6] [7], cost models are fed to an optimization solver which calculates an optimized partition plan by combining different remotable portions and their costs to meet an overall objective function e.g., lowest energy consumption, fastest execution time.

In dynamic partitioning on the otherhand, partitioning is performed during runtime. In Cuervo et al. [5] when the execution hits an offloadable method, a decision making formula will take input from the dynamic profiling (input such as energy consumption, residual energy, runtime duration, state transferring cost and connectivity) and makes a decision on whether the method should be run remotely. However, this way of profiling and partitioning incurs additional overhead because of the need to constantly monitor application behaviour.

2.2 Use of Cloudlet

There is a considerable body of research on cloud structures [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [4].

In Rossetto et al. [18], cloudlets are proposed to be used mainly as a workflow manager to process and manage a mobile device's request (submission, monitoring, and download of application results) to execute in the cloud. There are three modules: Controller, Engine and Collector uniquely created on a cloudlet for each submitted application from a mobile device. The controller is responsible for receiving requests, sorting these requests, commanding and controlling their order through the application identifier. The engine sends a resource request of the application to the resource selector which is responsible for gathering information related to the grid environment and for selecting resources based on resource characteristics, authorization, usage policies and requirements that application tasks require.

Satyanarayanan et al. [4] proposes that cloudlets be used in place of cloud servers to handle requests from mobile devices. They propose a Transient Cloudlet Customization which features pre-use customization and post-use cleanup. It uses VM encapsulation to separate the application software environment from the cloudlet infrastructure's permanent host software environment. They propose two different approaches for delivering the mobile state to the cloudlet: 1) Full VM migration, in which an already executing VM is suspended, its processor, disk, and memory state are transferred, and finally VM execution is resumed on the cloudlet from the exact point of suspension; 2) Dynamic VM synthesis, where the mobile device delivers a small VM overlay to the cloudlet infrastructure that already possesses the base VM operating system on which this overlay is able to execute.

Kovachev et al. [10], proposed a multimedia cloud architecture that specialized in processing and sharing multimedia applications and data for mobile devices. It also uses VM technology where virtual machines are grouped in realms which present different com-

puting resources: processing realm for parallel processing over many machines, streaming realm for handling streaming requests and general realm for running other servers such as XMPP or Web server.

Guan et al. [8] [9] generalize cloudlet usage into two models: 1) A remote access service model where mobile devices are used as a remote access point to interact with a cloudlet; 2) A substitution service model where services are offloaded to cloudlets for execution. They also propose a system infrastructure between mobile devices and cloudlets. In the infrastructure, the mobile devices maintain five modules: The Wireless Network Module which is responsible for low-level communication with cloudlets and other mobiles; the Device Proxy Module which is responsible for cloudlet discovery, initializing and terminating communications; the Execution Module sends a small script to a cloudlet so that it can download application code from the cloud; the Personal Information Module which stores user information and authentication and the Cache Module which stores interface codes and service status. The cloudlet's role in the infrastructure is to first process mobile device requests, and if necessary, suitable cloud services requests are invoked through the cloudlet to forward mobile requests to actual cloud servers. The cloudlet also applies VM technology and, in the infrastructure, promotes the idea of context-based ontology which uses vocabularies or short terms to describe each mobile, cloudlet and cloud server's characteristics so that a context reasoner can be implemented on cloudlets, which acquire context information from other entities and determine the optimal service to be accessed by the authenticated user.

Bahl et al. [11] proposes a RESTful programming model which promotes stateless offload, that is, cloudlets and cloud server provide common function services through http/https protocol to mobile devices which developers can take advantage of when programming applications. Such services include: (1)A Platform service that provides computing, storage, database, memcache services; (2)An Application service that provides commonly used applications functions: video streaming, location services, push notification, speech/image recognition and context-rich support service that gathers user information and provides recommendations (e.g., suggested videos, purchases), and context extraction for mobile devices.

Fesehaye et al. [12] provides a Cloudlet Routing design with two options: 1) Distributed routing approach where cloudlets are responsible for routing updates and storing routing tables that contain information on how to reach other cloudlets and mobile devices; and 2) Centralized Routing approach where a central server is responsible for computing routes by having each cloudlet submit its routing table to the central server.

Kemp et al. [13] argues that mobile devices should have more freedom in terms of

choosing which cloudlet or cloud server it requests its service from. It proposes to bundle client and service code together. The service code does not run on mobile devices, but can be installed onto a cloud by mobile devices. It also proposes the development of a toolkit called Interdroid which helps developers to develop applications with good communication with the cloud. For example, it provides an application on a cloud that helps with pulling information off the web and pushing notifications to mobile devices.

Rajachandrasekar et al. [14] argues that mobile devices are not only limited to being service recipients, but can also be computing resource providers. They argue that even though mobile CPU power and storage is limited, the aggregate of resources from a large number of mobile devices is significant. Mobile devices are used as resources in two ways: 1) Mobile Grids on-site where mobile devices are controlled by a central server, where each mobile device reports its capabilities and availability and the central server schedules the tasks; 2) Mobile Ad-Hoc Grid where there is no central server but rather devices form in an ad-hoc fashion, a virtual backbone consisting of more powerful mobile nodes responsible for coordinating distributed tasks.

Taylor et al. [15] propose to build applications which uses mobile devices with sensors as terminals to capture user interactions, and cloudlets as workspaces to hold temporary computing sessions for mobile devices in order to support rendering, processing sensor input, object tracking as well as caching information about its surrounding area.

Satyanarayanant et al. [16] propose multiple VM encapsulations on cloudlets where each encapsulates a different cognitive engine e.g., different cognitive services, object recognition, movement detection. A Control VM is responsible for all interactions with mobile devices including combining and outputing results from different cognitive VM engines to a single User Guidance VM to integrate together.

Ha et al. [27] also applies VM encapsulations on cloudlets. They propose the movement of VM encapsulations of mobile devices between cloudlets to maintain low end-to-end latency between mobile devices and cloudlets while mobile devices physically move around in a covered area.

2.3 Different Ways Of Cloudlet Prefetching Data

Duro et al. [19] makes use of memcached (An in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering [28]) to build a multi-layered cloudlet-based architecture for storing pre-fetched data. It is designed such that cloudlets in the front layers interact with mobile devices as a first level cache and other cloudlets or cloud servers serve as back-end backup storages.

Khawaga et al. [20] argues that mobile devices could form an ad-hoc network which is a multi-hop heterogeneous self-configured temporary network with dynamic topologies and without centralized administration. Within which, mobile devices can perform cooperative caching, that supports sharing and coordination of cached data among multiple mobile nodes. Each mobile device in this network also uses admission control to determine whether received data items should be cached. The admission control is based on factors such as the needs of neighbouring nodes: 1) General approach: All incoming data are cached; 2) Functional: A cost function is used to decide whether caching the data or not (distance, access frequency). Additionally, when local cached space is full on a mobile node, there are two options for cache replacement: 1) Local replacement in which cached data is evicted based only on local access behaviour; and 2) Coordinated behaviour in which the collective access behaviour of all mobile nodes in the ad hoc network is used.

Pingoy et al. [21] uses cloudlets to boost mobile video streaming performance for efficient social video sharing between users through pre-fetching videos for users and managing pre-fetched videos based on the frequencies that they have been accessed.

Sarwar et al. [22] applies Case Based Reasoning together with Artificial Neural Network algorithm in order to increase the hit rate of pre-fetching data.

Koukoumidis et al.'s Pocket Cloudlets [23] suggest that a cloud service cache architecture could reside on the non-volatile memory of mobile devices by combining multiple mobile devices together into a network. This is based on the assumption that the non-volatile memory of mobile devices will grow rapidly in future decades. The pre-fetching uses both individual user and community access models to maximize the hit rate of data pre-fetching. It also proposes pre-caching static data using wifi while pre-caching only dynamic data through radio channel when necessary in order to compromise the limited battery resources of most mobile devices.

Chapter 3

Cloud & Cloudlet & Mobile Data Pre-fetch Architecture

3.1 Example Application

This section describes an application that may benefit from the use of cloudlets prefetching data based on location.

A computer vision application collects images of objects. These samples and labels (e.g., the building name) are provided to machine learning algorithms to generate classifiers. The classifiers are used to detect objects from future images/video clips [29]. As the detectable objects pool grows, more data is needed to be stored, which includes sample images as well as object classifiers.

In a cloud computing environment, a mobile application typically relies on the cloud to store all of the data (sample images, object classifiers) and then either allowing the cloud to run the detection algorithms through streaming the visual view of the detection subject (Google Goggle [30]) or downloading small portions of the classifier data pool to detect specific objects locally.

3.2 Architecture of a Stand Alone Application

There are four major functional units as follows: File manager, location service, detection unit and user interface (UI).

The file manager is responsible for finding and downloading files from a cloud server which stores all application related data. This includes an index of all buildings that have been sampled, classifier files of building candidates based on the location of the user, and

additional feed files such as images, videos and description links about buildings that are within the camera view.

The location service is responsible for periodically updating the location of the user. This is used to filter out buildings that are out of the mobile camera's maximum distance.

For each camera frame captured, the detection unit executes the detection algorithms using the classifier of candidate buildings. These buildings are determined by location.

The User Interface (UI) is responsible for reading and rendering detection results. In additional it manages the presentation of extra feeds (e.g., image galleries, video links, floor maps) for identified buildings to the user.

Program execution is graphically depicted in Figure 3.1. Execution starts from the file manager downloading the list of all buildings of a relatively large area (e.g., the list of all buildings within Western University) from a cloud server and the location service retrieving the user's current GPS location. The GPS location is used together with a pre-set radius of 100 meters (which is typical for a mobile camera view range) by the location analysis function in the mobile application to filter the buildings list and generate a small pool of nearby building candidates. The file manager then downloads all classifier files that are related to these candidate buildings from the cloud server. Meanwhile, the detection software starts looping through each camera frame. The downloading of files is parallel to the detection. When there are no classifiers downloaded, the detection software detects nothing and a normal camera frame is rendered to the user. As classifiers are downloaded one after another, the detection software is able to detect more and more buildings in the camera frames. When the building is detected, the detection results are used by both the file manager and user interface. The file manager uses the result to download additional feed information about the building and the UI uses the result to display on the mobile screen.

3.3 Architecture of a Cloud Version

The cloud version refers to the partitioning of the application where only minimum functionality of the application resides on the mobile device and a major part on a cloud server or cloudlet. This includes managing building lists, feeds, classifiers, processing camera stream and object detection.

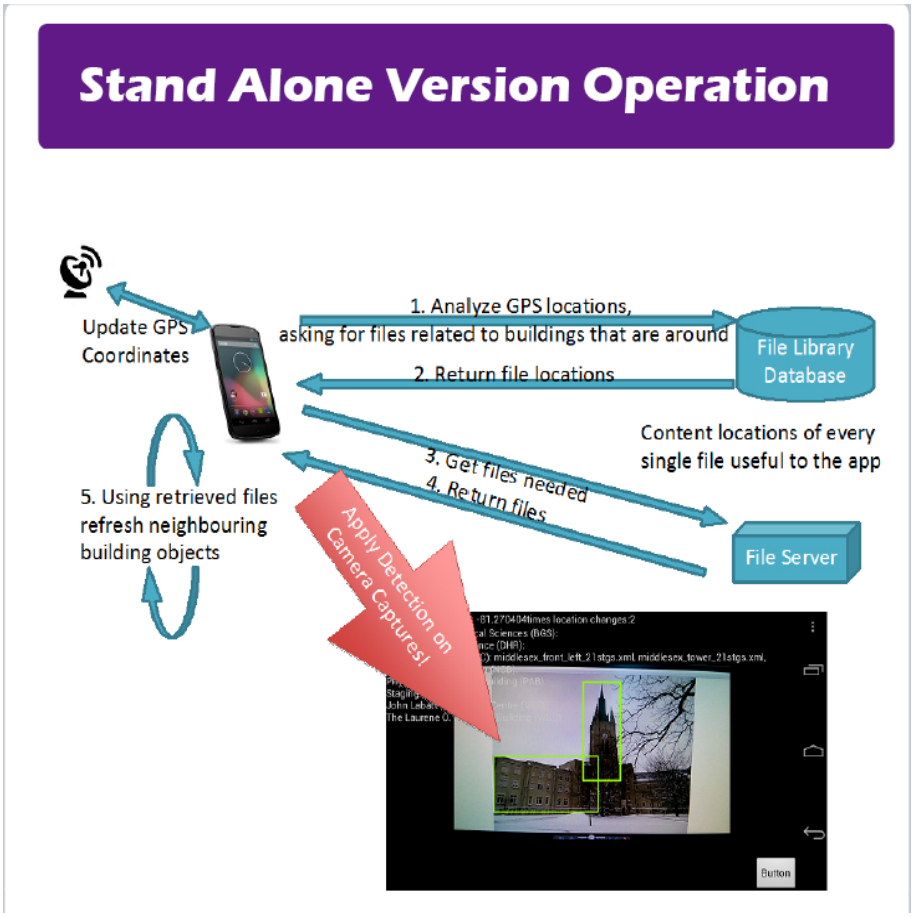

Figure 3.1: process of stand alone version

3.3.1 Client

The location unit is responsible for retrieving the user's current location coordinates. The location coordinates are sent to the server. The file manager downloads the extra feeds of buildings when the detection results are returned from the server.

The client side needs three units that were not required for the stand alone application. This includes the outbound streaming unit, the inbound streaming unit and the messenger unit. The Messenger unit is responsible for the data exchange between the client and the server. The types of messages sent by the messenger include messages to establish a streaming connection between client and server, periodic GPS coordinate update of current client location, and the detected building list which the server, upon detection of buildings, sends to clients. The Outbound streaming unit is responsible for transmitting the mobile device's camera view to the remote server. The Inbound streaming unit is responsible for receiving the video stream returned from the server.

The stream sent is the original video stream with detected buildings marked out on each video frame. The reason why the live video stream is used over transmitting only a series of images between client and server is because the mobile camera captures on average 30 frames per second. Each frame can be considered as a single image which can be used for detection. Although the client and server can manage the transmission of 30 images per second, the 30 frames in a second can be highly compressed, and thus we use video streaming to packetize the video into packets and then transmit the packets between the client and server, which is a more efficient way of transmitting the data.

3.3.2 Server

The server part of the application detects the buildings, filters buildings based on location coordinates sent by the client, and a file manager for pre-fetching data based on the location of the cloudlet from the cloud. In addition, there are Outbound streaming, Inbound streaming and messenger units which are similar to those found in the clients. The server part also includes a transcoding unit which converts the video stream into formats that are used by each of the outbound stream, inbound stream and messenger.

3.3.3 Full Program Cycle

A full program cycle is graphically depicted in figure 3.2 which starts with the server side initializations. The server initialization pre-fetches files needed for the application through the file manager (including building list, each building's classifier files for buildings that are in the building list) from the cloud. This speeds up the detection process since servers have more computational resources and thus can process data faster. The server also initializes the outbound streaming service, client reception service and waits for a client connection. For each client connection, the server side creates a session for communication with this client. When a client successfully connects to the server, it sends session data information to the server. It will then stream its camera view towards the server. Using the session information received, the transcoding unit on the server side is able to decode the incoming packets into video frames. The video frames are fed to the detection unit. The clients update the user's latest location coordinates every 10 meters moved or after every 1 minute. This is a pre set threshold used to determine the frequency that a client updates a server of its GPS location. Higher frequencies comsume energy while a frequency that is too low may result in inaccurate user location information. The coordinates are used on the server to generate a list of nearby buildings. Based on the list of nearby buildings, the detection unit updates classifiers loaded for detection.

After going through detection, each video frame is then converted back to H264 format and streamed back by the server's outbound streaming unit. Lastly, the client's inbound streaming unit receives the video stream back which is displayed on the screen.

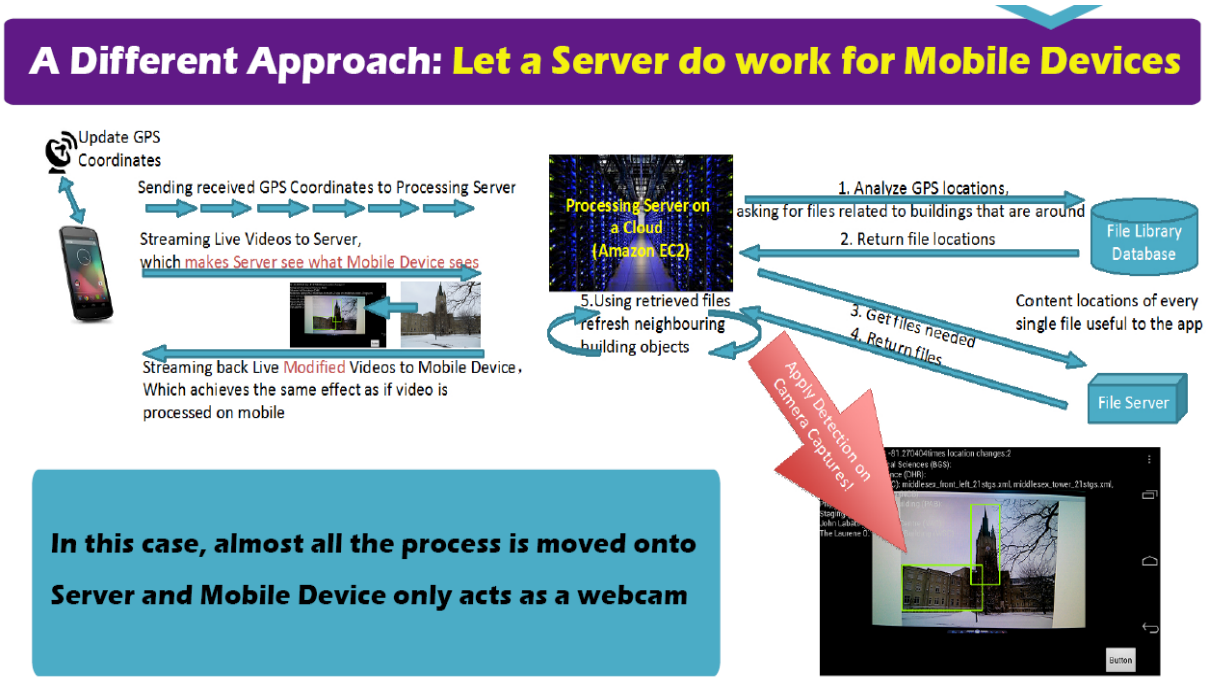

Figure 3.2: process of Client-Server version

This example is not unique and can be generalized into many different stand alone/cloud based application architectures. Applying cloudlet-based network architectures to applications has several benefits. First, all application related data (in the case of the example, this includes: sample images and classifiers) can be pre-fetched onto cloudlets. This releases storage of mobile devices from storing data locally while still providing a relatively faster access of the data than having to access cloud servers directly. Secondly, since cloudlets possess computing power, the mobile requests is resolved at cloudlets instead of forwarding to cloud servers. This reduces workload on a cloud as well as reduces network traffic in the network core.

Additionally, application data often possess important context information which can be used to improve pre-fetch efficiency and accuracy. In the example scenario, the context information could include the physical locations, average number of mobile devices around the area etc. For a cloudlet, prioritizing pre-fetched data related to the same area that the cloudlet is located may be more efficient since mobile devices accessing the cloudlet are more likely around the same area and therefore more interested in the data of the

area than data related to other areas.

Chapter 4

Experimental Environment

This chapter describes the experimental environment.

4.1 Application

The application used in the experiments identifies buildings and landmarks. Once a building is detected the application will use coloured rectangles to mark out the detected buildings/landmarks on screen. In addition, it displays extra feeds (including images gallery, video clips and short descriptions of identified buildings on to the side of the screen).

There are two versions of the application. The stand-alone version runs on the mobile devices and only downloads data from the cloud. In the cloud/cloudlet version, the mobile device sends live video stream of its camera to the cloud/cloudlet server where the software for identifying building/landmarks take place. The cloud/cloudlet server also pre-fetches application related data to avoid time cost on downloading them during actual application runtime.

4.2 Object Recognition

The application requires object recognition algorithms for identifying a specific object in a digital image. In this work, we used object recognition algorithms based on supervised machine learning which uses training data with labels to learn a model of the data. The training data represents features that includes edges, gradients, histograms of oriented gradients (HOG), Haar wavelets and linear binary patterns [31]. Object recognition algorithms are implemented in OpenCV [32]. The preparation for the machine learning

process includes:

1. Collection of Samples: An average of 250 sample images were gathered for each target object representing a building/landmark. The reason why the number of sample images may vary from building to building is becuse different buildings have different features. Some have only one significant feature while others have multiple features. Each feature needs to be trained separately and therefore some buildigns end up having more sample images than others. These images are used as training data by the OpenCV machine learning algorithm: *opencv_traincascade*, which supports both Haar wavelets and LBP (Local Binary Patterns) features [33].

2. Pre-Processing of the Samples: The sample images are pre-processed. For example, writing down target object position coordinates relative to each sample image, removing negative images, using image filters on images for contrast enhancement and image scaling to get more useful features of the target object.

3. Generation of Object Classifiers: The training data is provided to the machine learning algorithm: *opencv_traincascade*. The classifiers generated this way represent the object features and will be applied later on to identify the object from other images or views.

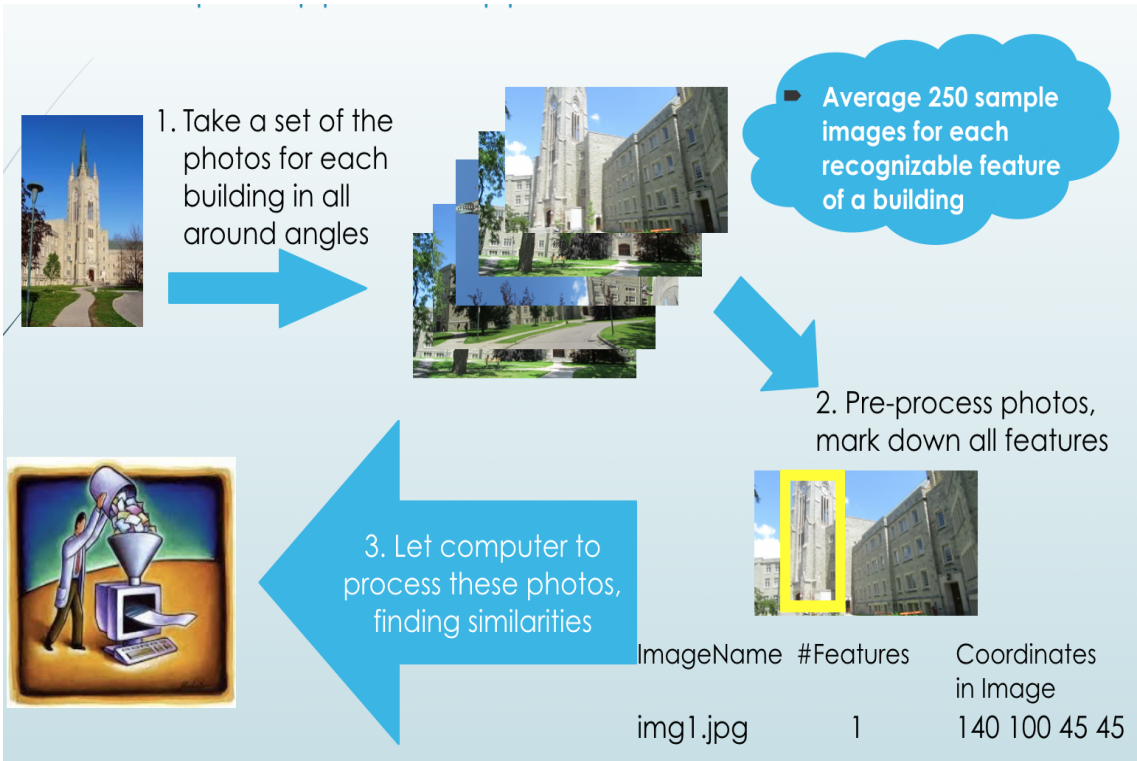

Figure 4.1: how does detection happen

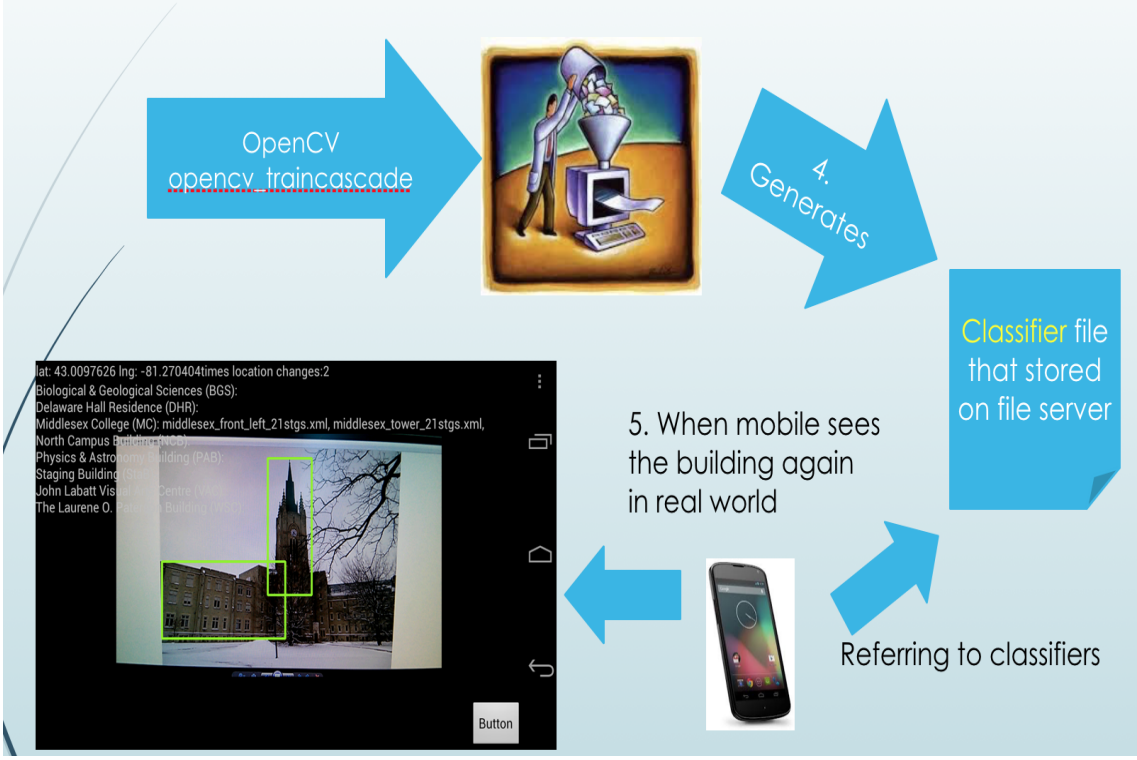

Figure 4.2: how does detection happen

4.3 Video Streaming and Transcoding

Video Streaming and transcoding technologies are applied in the cloud/cloudlet version prototype for the purpose of sending camera captured views from mobile devices to the cloud. It is also applied in the reverse process which mobile devices receive the detected & modified camera views from cloud. The H264 standard [34] is used when compressing camera views into packets and sent to the designated server.

4.4 Application implementation

The application is centered around the building/landmark detection functionality. As briefly introduced in Section 4.2, in order to identify buildings from a camera view, the application needs each building's generated classifier files. In reality, a mobile camera's view range is limited, therefore, the potential building candidates that the detection function needs to go through given a physical location is limited. Thus, before running the actual detection, we first apply a location filter to select the potential candidates from the pool of all known buildings(a building is considered known if its classifiers have

been generated). This concept is applied in both versions of the prototype.

4.4.1 Stand Alone Version implementation

Programming languages and OS

The stand alone version is programmed in an Android OS 5.1.1 Lollipop environment using Android Studio 2.0. The external library OpenCV-android-sdk 3.0 [32] is used for handling object detection.

4.4.2 Cloud Version Implementation

Programming languages and OS

The mobile side is programmed in an Android OS 5.1.1 Lollipop environment using Android Studio 2.0. The external libraries used include libstreaming [35] for handling outbound streaming from mobile devices and vlc-android-sdk [36] for handling inbound streaming to mobile devices.

The server side is programmed in both Linux(CentOS and Ubuntu) and a Mac OSX environment. The external libraries include OpenCV 3.0 for Macosx/Linux [32] for handling object detection, FFMPEG libraries [37] for handling inbounding video streaming as well as video transcodings, Live555 [38] for handling outbound streaming from the server, and libcurl 7.41.0 [39] and rapidxml 1.13 [40] are used for handling file downloading and file content parsing.

Additionally, we uses Session Description Protocol (SDP) to pass along video streaming port and type of video codex. between mobile devices and servers.

4.5 Testbed

The testing environment consists of two remote cloud servers and a local Macbook Pro running OS X Yosemite (version 10.10). Both cloud servers use Linux. The reason for having two remote servers is to demonstrate that distance between mobile devices and remote servers affects the speed of data transfer. Therefore, one server running CentOS release 6.3 is close to the testing location (at Western University, London, Ontario), the second remote server running Ubuntu 14.04 is provided by Amazon EC2 and is located in Ashburn, Virginia, US. The prototype applications are mounted on Android system 5.1.1 lollipop on a LG Nexus 4 mobile device.

4.6 Testing Goal and Metrics

Evaluation metrics include the following:

1. Latency

The latency measures the time needed for an application to download files or access data from cloud servers as well as transmitting to and receiving packets from cloud/cloudlet servers. The time is measured in milliseconds.

2. Battery usage

The battery usage measures energy consumption of an application's run. In the experiment, we let each version of the application run a certain time period and measured the energy consumption in milliwatt-hour (mWh)/milliampere-hour (mAh).

3. Frame rate

The frame rate defines the smoothness of the visual display, measured by Frames Per Second (FPS), which indicates how many video frames are updated every second.

4. CPU, RAM and GPU usage

CPU and GPU usage are measured by percentage of the total capacity while RAM usage is measured in megabytes indicating how much RAM the application needs to have reserved in order to function.

Chapter 5

Experiments

This chapter evaluates the two architectures presented in Chapter 2. The two different versions of the application enable us to perform a series of tests and comparisons around using cloud server/cloudlet.

5.1 Experimental Setup

The experimental setup is organized so that it is representative of an organization of servers that includes cloud server, cloudlet server, local server and a mobile device.

The first cloud server is provided by Amazon EC2 and is located in Ashburn, Virginia, USA. We use the name **Amazon EC2** in the rest of this chapter. The specifications of this server are the following:

OS: Linux - Ubuntu 14.04

CPU: Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz, 1Core

RAM: 1.02GB Memory

The second cloud server is located at Western University in London Ontario. We use the name **Circe** in the rest of this chapter. The specifications of this server are the following:

OS: Linux version 2.6.32-279.19.1.el6.x86_64, CentOS release 6.3

CPU: Intel(R) Xeon(R) CPU E7- 4820 @ 2.00GHz, 64 Cores

RAM: 793.97GB Memory

The local server is a laptop. We use the name **Macbook pro** in the rest of this chapter. The specifications are the following:

OS: Mac OS X Yosemite 10.10.5

CPU: Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz, 1 Cores

RAM: 8 GB Memory

The mobile device specifications are the following:

LGE Nexus 4

OS: Android Version: 5.1.1

GPU: Qualcomm Adreno (TM) 320

CPU Cores: 4 — Min-Max Frequencies: 384Mhz - 1.51Ghz

5.2 Data location vs Data Retrieval Experiments

This first series of experiments measures file downloads from cloud servers. The servers are at different distances and the files vary in size. The time to download a file is measured in milliseconds. The time to download a file is based on the difference of the timestamp taken before the request for a file download is made and the timestamp taken after the file download.

The stand alone version of the application is used in this test and all three servers are used. There are various types of files to be downloaded including: Utility file (the buildings index file size is 12 kilobytes), buildings' classifiers (file sizes are between 16 kilobytes to 41 kilobytes) and images about each building. This includes thumbnail images (file sizes are between 5 to 6 kilobytes) and full size images (file sizes are between 220-250 kilobytes). In addition we tested the time it takes for a SQL query to complete between the client to the server MySQL database. The time associated with an SQL query is measured in a similar fashion to file downloads. A client uses the SQL query to specify a building which is used to query the cloud server's database to find any video links, text informations, and the download paths for any files associated with the building.

5.2.1 Metric

The metric used in this experiment is the time it takes to download a file or finish a query. This is measured in milliseconds. The stand alone version takes a timestamp before initiating a download or query and after each download or query is completed. It then calculates the time gap between these two timestamps. This difference is the time it takes to download a file or complete a query.

5.2.2 Test 1

The mobile device is physically located in Middlesex College, at The University of Western Ontario (UWO), in London, Ontario. The application is executed three times with each execution retrieving data (e.g., files and additional information as described in Chapter 5.2) from a different cloud server e.g., Amazon EC2, Circe and Macbook Pro. In each run, the application made on average 12 SQL queries including one query for finding the building list file's download path and 4 to 7 queries for finding download paths for the classifiers of buildings and 4 to 8 queries for finding the download paths of additional images or text information about buildings, e.g., video links, website links or text descriptions. The application made 1 utility file download and on average 6 to 12 building classifier downloads, 25 to 36 thumbnail image downloads and 15 to 24 full size image downloads. The results in Figure 5.1 present the average time to complete a query, to download classifiers, utility files and images from three different servers. Data access is the fastest from the Circe server. This could be because the mobile device is physically located in Middlesex College and within the same subnet as Circe. We assumed that the Macbook Pro would have the fastest access time since there is a LAN connection to the mobile device. However, the results do not show this. This may be the result of subnet security at Western where it wasn't possible for the mobile to reach the Macbook Pro through the same subnet used to reach Circe or Amazon EC2. As a result, we had to switch to a different router (Cisco Linksys E1200) to perform the LAN connection test. This might result in unusual performance when the mobile device accesses the Macbook Pro.

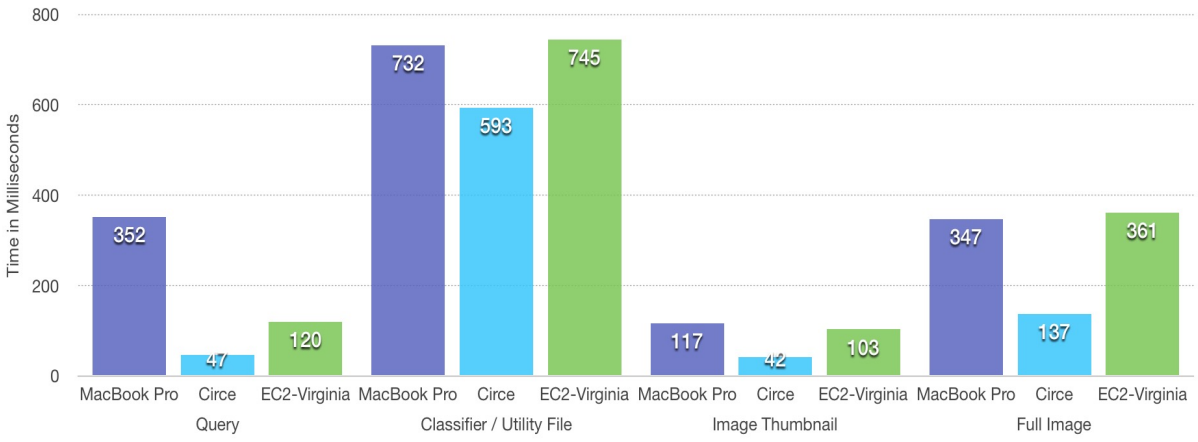

Figure 5.1: VERSION 1 - TEST 1

5.2.3 Test 2

The second test is similar to Test 1 except the mobile device is physically at a different place: 120 Cherryhill Place, London, Ontario. The distance from the mobile device to Circe increases. The results are presented in Figure 5.2 as accessing all three servers through the same home cable router. While the mobile device moves away from the subnet of Circe, the latency of accessing Circe increases significantly. The time to download a file or to complete a query exceeded that of Amazon EC2 except for full images. This result was not what we expected since the Circe server is located at a closer distance compared to Amazon EC2 in the locations for Test 1 and Test 2. The time to access data from Amazon EC2 increased slightly compared to Test 1. The Macbook Pro maintained the performance compared to Test 1, which is expected since there are no other network hops between the LAN connection of the Macbook Pro and the mobile device and therefore less variation in the time to make an SQL query or download files.

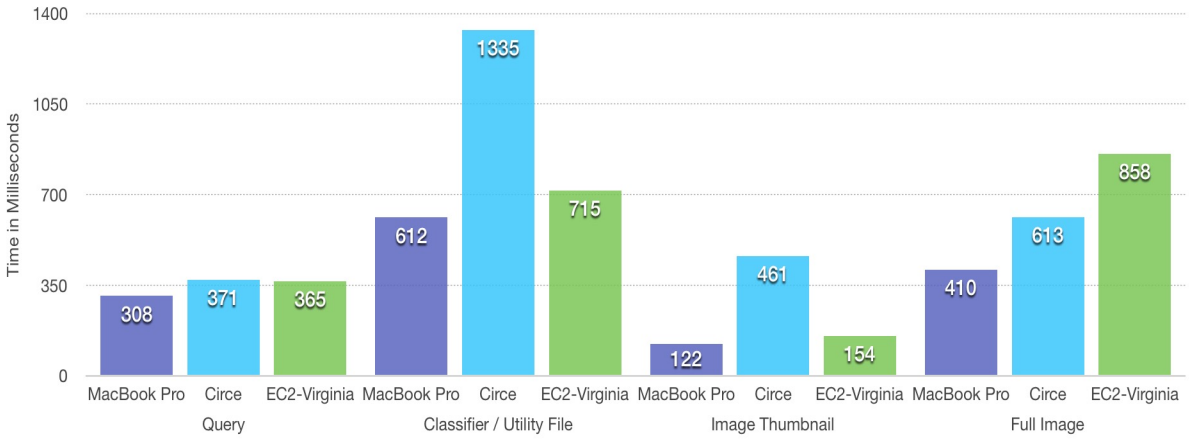

Figure 5.2: VERSION 1 - TEST 2

Figure 5.3 presents Circe's download times from Test 1 and Test 2. It clearly shows Circe's performance varies significantly as the mobile device moves from within the same subnet of Circe to outside of the subnet.

Location 1: 120 Cherryhill Place, London, Ontario (Three kilometres away from where Circe located, which is outside the same subnet of Circe)

Location 2: Middlesex College, UWO, London, Ontario (Exactly where Circe located, same subnet)

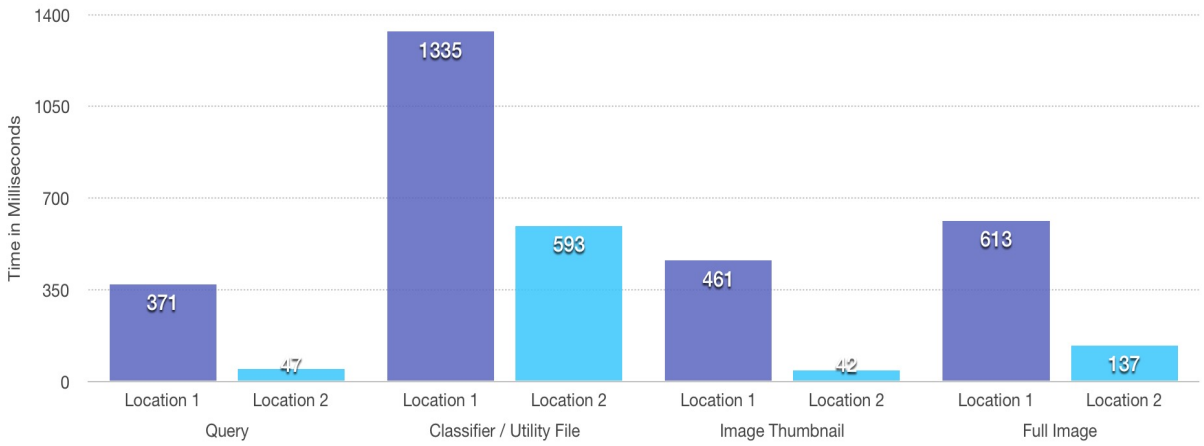

Figure 5.3: Accessing Circe from two different locations - Comparison 1

5.2.4 Test 3

Although we expected the time to retrieve data from Circe would vary between the two locations, we were unsure why accessing data from Circe takes longer than Amazon EC2 except for full sized images. This third test performed uses the same setup as Test 1 and Test 2, except that the mobile device is physically located at a different place: 50 Capulet Lane, London, Ontario. The distance from the mobile device to Circe increases further and this time we used Circe, Amazon EC2 located in Virginia, USA and an additional Amazon EC2 server located in Tokyo, Japan. The results are presented in Figure 5.4. Circe outperforms both Amazon servers for queries this time, while the Amazon server in Virginia outperforms Circe again for both file downloads and image thumbnails which is consistent to the results seen in Test 2. However, if we look at the performance of the Amazon EC2 Tokyo, we can clearly see that longer physical distance between mobile devices and cloud does impact application performance as it was observed the Circe was faster for data access except for image thumbnail downloads.

5.2.5 Discussion

Combining Test 1, 2 and 3, we observe the following:

1. Amazon EC2 - Virginia outperforms Circe in Test 3 for most types of data accesses.
2. The SQL query time for the Circe server in Test 3 outperforms Amazon EC2 - Virginia which is different than that that of Test 2. On the other hand, the full image download time of Circe out performs Amazon EC2 - Virginia but is out performed by Amazon EC2 - Virginia in Test 3.

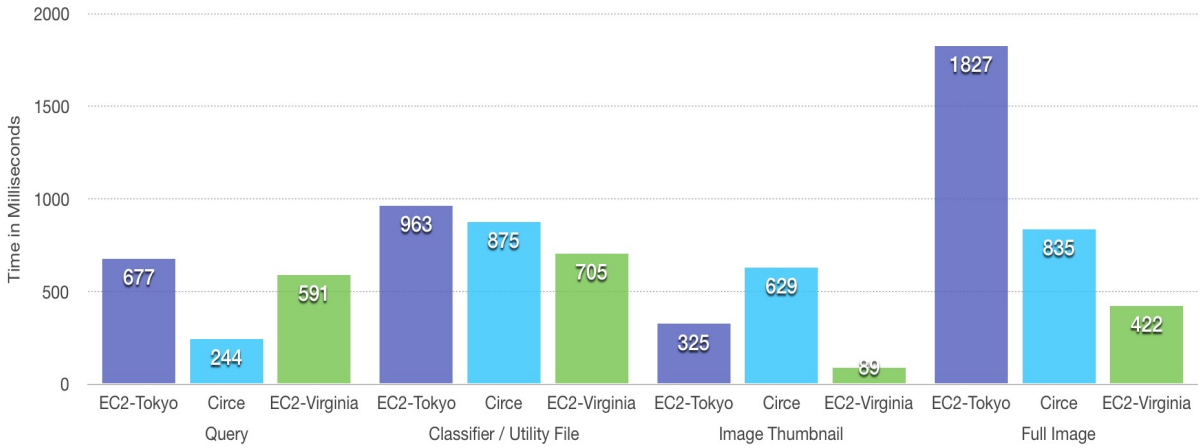

Figure 5.4: VERSION 1 - TEST 3

3. The classifiers / utility file download time of Circe varies significantly between Test 2 and 3 although the two locations (120 Cherryhill Place and 50 Capulet Lane) are fairly close to each other. Note, the WIFI connection uses the same router, same ISP and same Internet. However the available bandwidth on the connection between the home router and the ISP router may vary.

First of all, we can not eliminate network fluctuations during the experiments. Therefore, there are factors that could affect the results. First, there are outliers where one or several queries or file downloads takes a significant amount of time compared to the average. Second, because some of the data are relatively small in size and their download times are measured in milliseconds, even a small network fluctuation could result in significant changes in download times. Therefore, to better understand the data, we should consider the following:

1. Amazon - Virginia out performs Circe and both outperform Amazon - Tokyo in most types of data accesses.

2. Circe performance varies based on if it is located in the same subnet as the mobile device.

3. Amazon - Virginia outperforms Circe in both Test 2 and Test 3 for most types of data accesses. One possible theory is that Amazon EC2 - Virginia is relatively close and the amount of data is not significantly large.

5.2.6 Additional Comparisons

Tests 1 and 2 show the average time to download data. We present four additional comparisons (Figure 5.5, 5.6, 5.7, and 5.8) which present download times when the mobile device is physically located at two different places and downloads data from the three different servers. Each dot in the graphs stands for the download time of a single image file (thumbnail image sizes are between 5 to 6 kilobytes and full size image sizes are between 220-250 kilobytes), the y-axis stands for the exact download time in milliseconds and the x-axis means nothing expect simply laying out the results in a evenly distributed manner so that every file download result is shown without overlapping. The curved lines are polynomial regression of the dots generated by Mac Software Numbers. This is used to make it easier to make visual comparisons of download times from the three different servers.

Comparison 1: Figure 5.5 shows the time to download thumbnail images from each server while the mobile device is at 120 Cherryhill Place, London, Ontario.
Comparison 2: Figure 5.6 shows the time to download full images from each server while the mobile device is at 120 Cherryhill Place, London, Ontario.
Comparison 3: Figure 5.7 shows the time to download thumbnail images from each server while the mobile device is at Middlesex College, UWO, London, Ontario.
Comparison 4: Figure 5.8 shows the time to download full images from each server while the mobile device is at Middlesex College, UWO, London, Ontario.

5.3 Streaming cost incurred in Cloud Version Application

The above series of experiments presents the analysis of download times for the stand alone version to retrieve data from different servers in various locations. In the cloud version, since the detection process is executed on the server side, there is no need for mobile devices to download the building list or any classifier files. Although the server has more powerful computing resources compared to the mobile device and there is no need to download classifier files or a building list, there is a need for the mobile device to send its camera view as video streams to the server as well as the server to return the resulting video stream back to the mobile device. This bi-directional video stream transmission adds additional time to the application cycle comparing to the stand alone version where there is no need to transmit video stream with server. This section presents

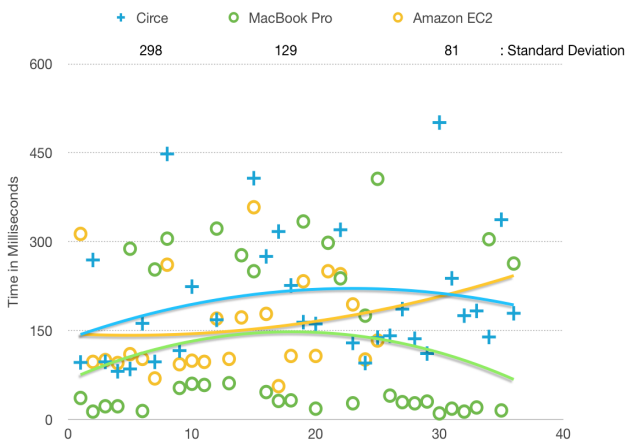

Figure 5.5: Accessing three servers - Comparison 1

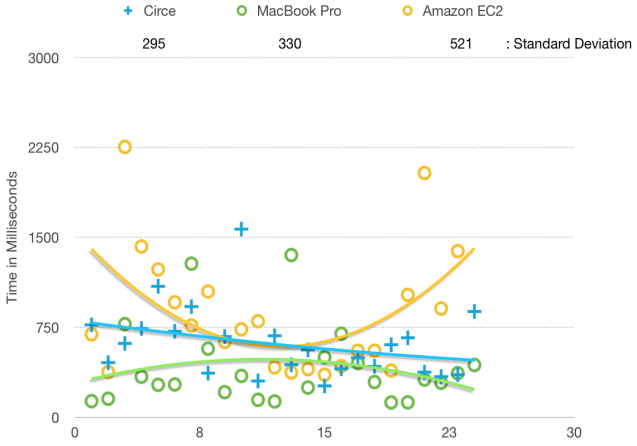

Figure 5.6: Accessing three servers - Comparison 2

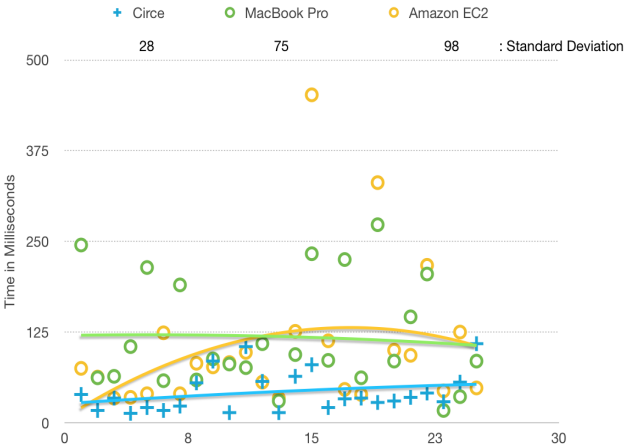

Figure 5.7: Accessing three servers - Comparison 3

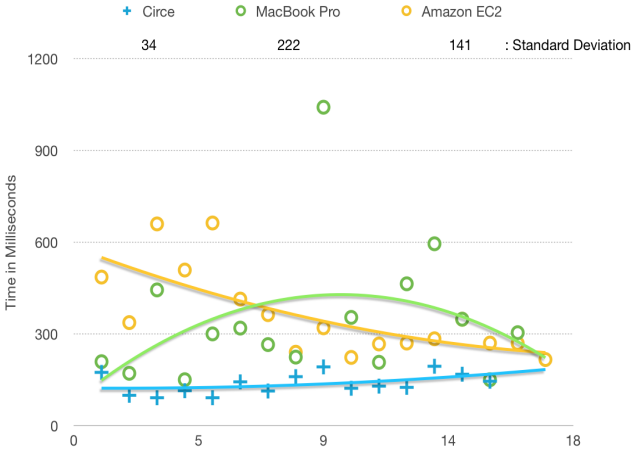

Figure 5.8: Accessing three servers - Comparison 4

our experiments related to the cloud version of the application.

Two servers are used for comparisons: The Circe server and the Macbook Pro server. The mobile device is at 50 Capulet Lane, London, Ontario. We measure the time between sending the camera view of the video stream to the server and the receiving of the video stream from the server. Two timestamps are used. The first is before the *streamToServer()* function which is used to send the camera view. The second timestamp is placed before the *setSurfaceSize()* function which sets the video display size and starts rendering the modified video stream from the server. The difference is referred to as latency. We use the Android function *System.currentTimeMillis()* for the timestamps.

Figure 5.9 shows the latency when the server is the Macbook Pro. The application is executed 5 times and Figure 5.9 shows the average latency incurred. The average latency

is 5242 milliseconds with standard deviation of 230 milliseconds. Figure 5.10 assumes the server Circe. The average latency is 6680 milliseconds with standard deviation of 823 milliseconds.

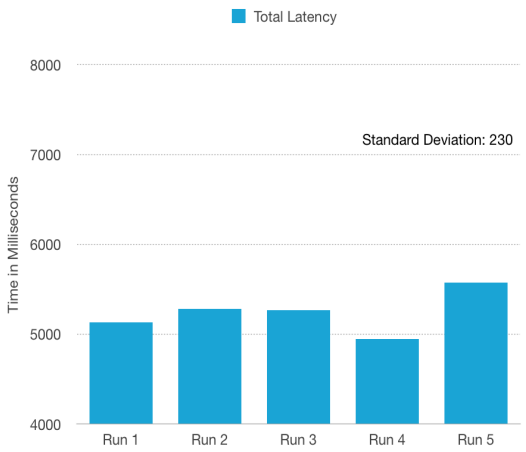

Figure 5.9: Mobile - Server Latency - MacbookPro

Figure 5.10: Mobile - Server Latency - Circe

To compare this result with stand lone version, we uses the same two servers and from the same location to measure stand lone version's time between downloading the initial building list and rendering the first detected camera view. Two timestamps are used. The first is before the *fileManager.startFilePreparation(ACTION_GET_GENERAL_FILE, "buildings")* function which is used to download the building list file. The second timestamp is placed after the *PrepareFile()* function which used to download all classifiers from server. We also use the Android function *System.currentTimeMillis()* for the timestamps.

In stand lone version, the nearby building intensity and each building's different features decide how many building classifiers files need to be downloaded. The more classifiers that needed to be downloaded is expected to incur a longer latency before users are able to see detected results. To also experiment on this, we use the above setup to perform two tests. First includes only single building as nearby buildings and second includes four buildings as nearby buildings. The one building only requires 1 to 2 classifier files to be downloaded while the four buildings requires total of 6 classifier files to be downloaded.

Both test is executed 5 times, Figure 5.11 shows the latency of single nearby building when the server is the Macbook Pro. Figure 5.12 shows the latency of four nearby buildings when the server is the Macbook Pro. The average latency incurred with single nearby building is 2453 milliseconds with standard deviation of 264 milliseconds. The average latency incurred with four nearby buildings is 6123 milliseconds with standard

deviation of 304 milliseconds. Figure 5.13 and Figure 5.14 shows the latencies of single and four nearby buildings when the server is Circe. The average latency incurred with single nearby building is at 3135 milliseconds with standard deviation of 437 milliseconds. The average latency incurred with four nearby buildings is 8332 milliseconds with standard deviation of 1503 milliseconds.

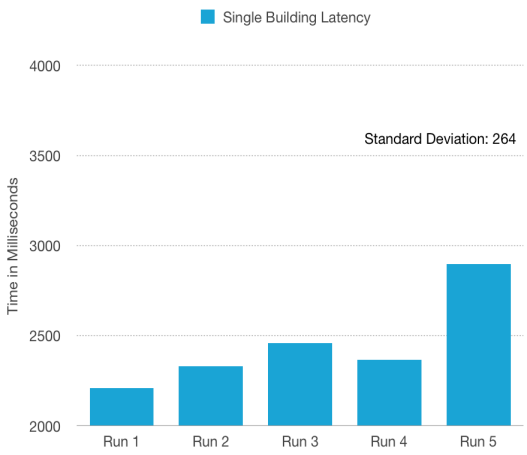

Figure 5.11: MacBookPro - Mobile StandAlone - Single Building

Figure 5.12: MacBookPro - Mobile StandAlone - Four Building

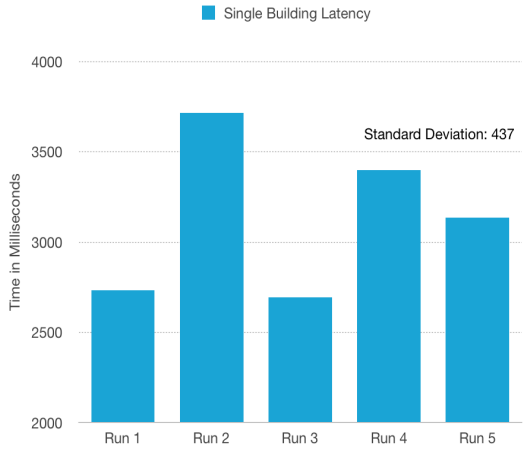

Figure 5.13: Circe - Mobile StandAlone - Single Building

Figure 5.14: Circe - Mobile StandAlone - Four Building

5.3.1 Discussion

The stand alone version needed average of 2000 to 3000 milliseconds to get the classifier files for single building before the detection software on the mobile device can be executed (with one additional building list file). When there are four buildings, the required time averages 6000 to 8000 milliseconds.

In the cloud version, the user of the device will receive a marked up video after 5000 to 6000 milliseconds. However, it prefetches classifier files so there is no download time during application execution.

Essentially the results suggest that the more buildings there are within camera view, the more classifiers that need to be downloaded to the mobile device. This would result in the stand alone version being slower whereas the cloud version would not be affected by this change.

5.4 Stand Alone Version and Cloud Version Performance Comparison Experiment

In this section we examine the performance difference between the stand alone version and the cloud version. We look at the performance comparison from five different aspects: Battery consumption, Frame rate, CPU Usage, RAM Usage and GPU Usage. The performance is measured by a third party software called GameBench [41].

5.4.1 Metrics

The total battery consumption is measured by milliwatt-hour (mWh)/ milliampere-hour (mAh). Both versions of the application are executed for 16 minutes (which is the minimum required duration for battery consumption observation by GameBench) and measures battery consumption. The second metric is the frame rate which is measured by frames per second (FPS) that indicates the number of frames rendered in a second. A higher frequency indicates smoother video display. The third metric is CPU usage, which is measured by the percentage of total CPU capacity needed for running the application. The fourth metric is RAM usage, which is measured in megabyte indicating how much RAM the application needs to reserve in order to run. The last metric is GPU usage, which is measured as the percentage of total GPU capacity needed for graphically rendering the application screen.

Stand Alone version performance:

1. Battery consumption:

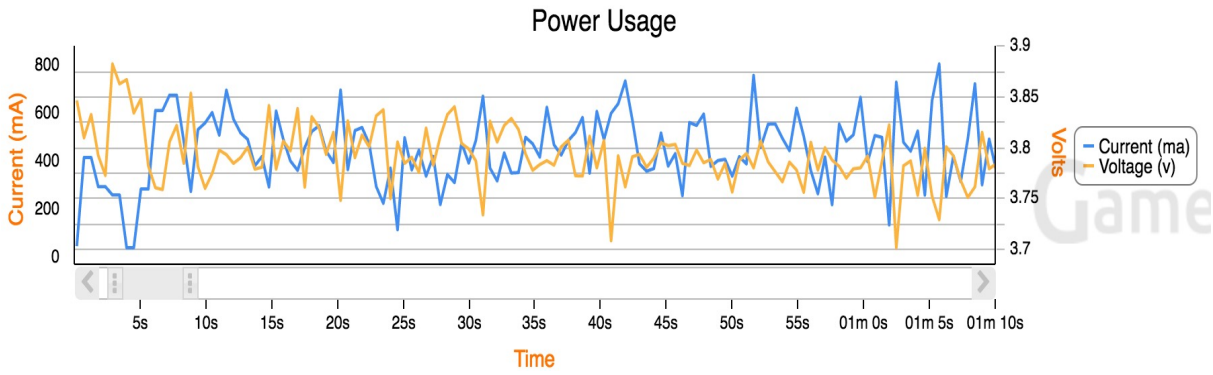

Figure 5.15: Stand Alone Version Performance - Battery

The total battery consumption is 564.40 mWh

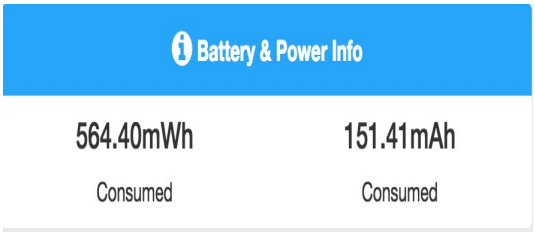

Figure 5.16: Stand Alone Version Performance - Battery - Sum

2. Frame rate: The stand alone version has an average frame rate of 21 fps and 99% FPS Stability where the FPS Stability measures how stable the frame rate keeps at the average frame rate through out the application run

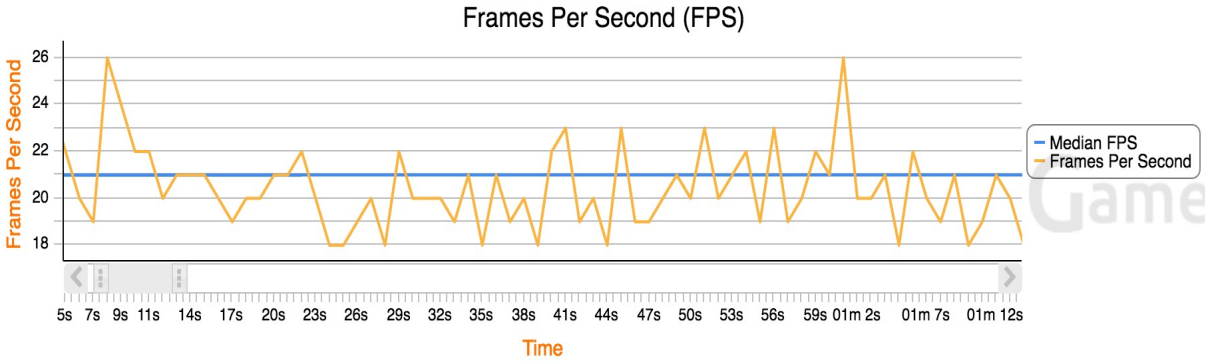

Figure 5.17: Stand Alone Version Performance - FrameRate

3. CPU Usage: The stand alone version has an average CPU usage of 21%, a 66.09% peak CPU ssage and 384Khz - 1.51Ghz minimum and maximum CPU frequencies

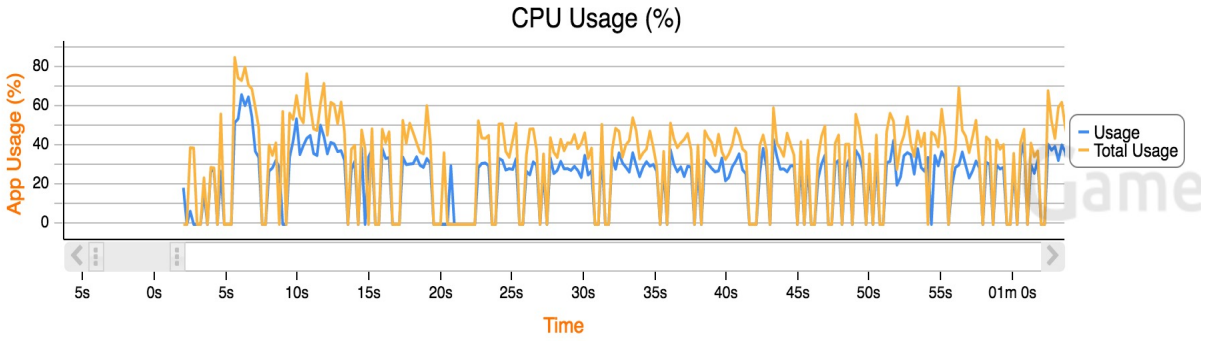

Figure 5.18: Stand Alone Version Performance - CPU

4. RAM Usage: The stand alone version has an average memory usage of 54MB and 60MB peak memory usage

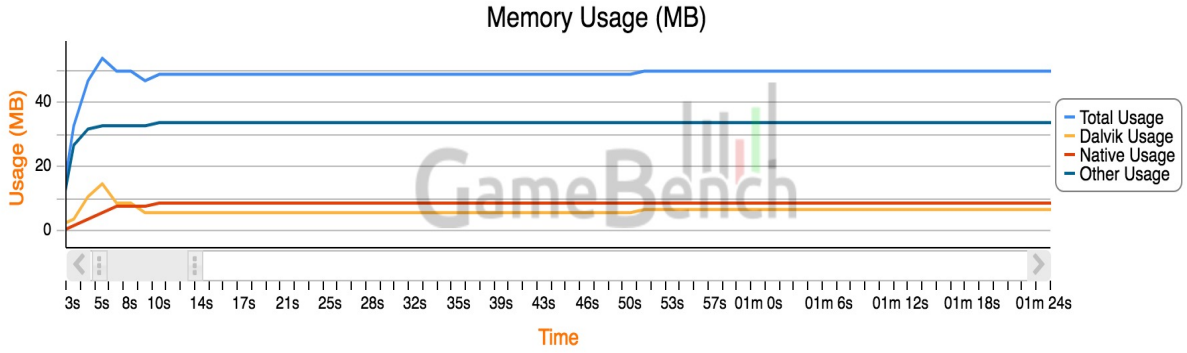

Figure 5.19: Stand Alone Version Performance - RAM

5. GPU Usage: The stand alone version has an average GPU usage of 10.22% and 31.22% peak GPU usage

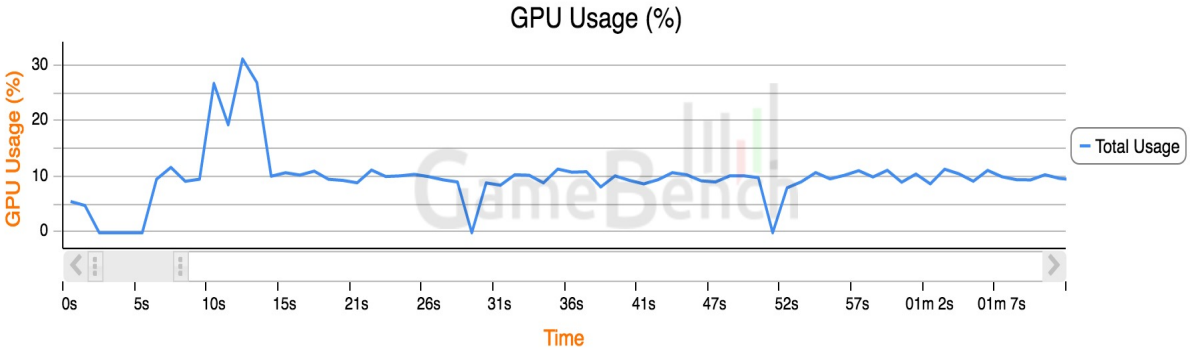

Figure 5.20: Stand Alone Version Performance - GPU

Cloud version performance:

1. Battery consumption:

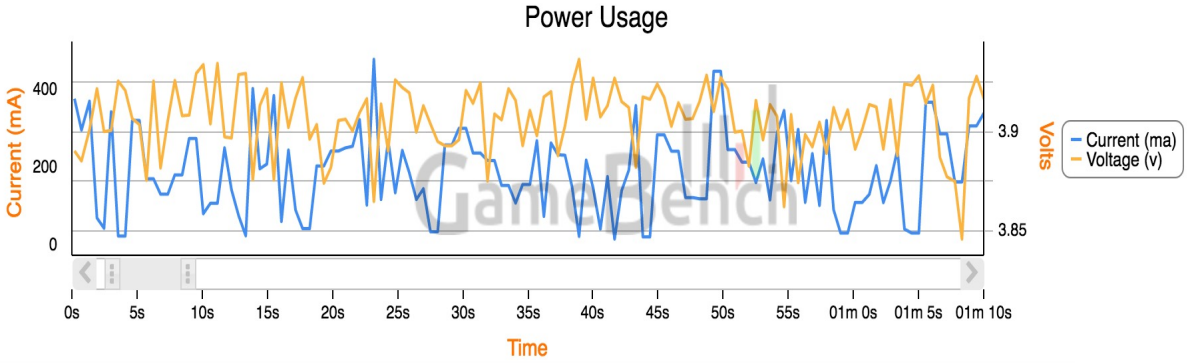

Figure 5.21: Cloud Version Performance - Battery

The total battery consumption is 195.75mWh

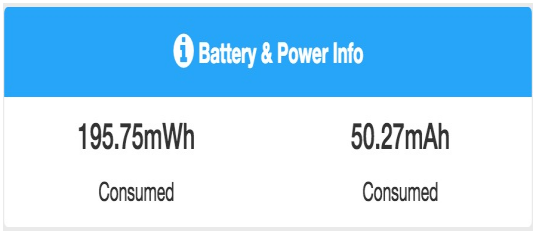

Figure 5.22: Cloud Version Performance - Battery - Sum

2. Frame rate: The cloud version has an average frame rate of 30 fps and 97% FPS Stability

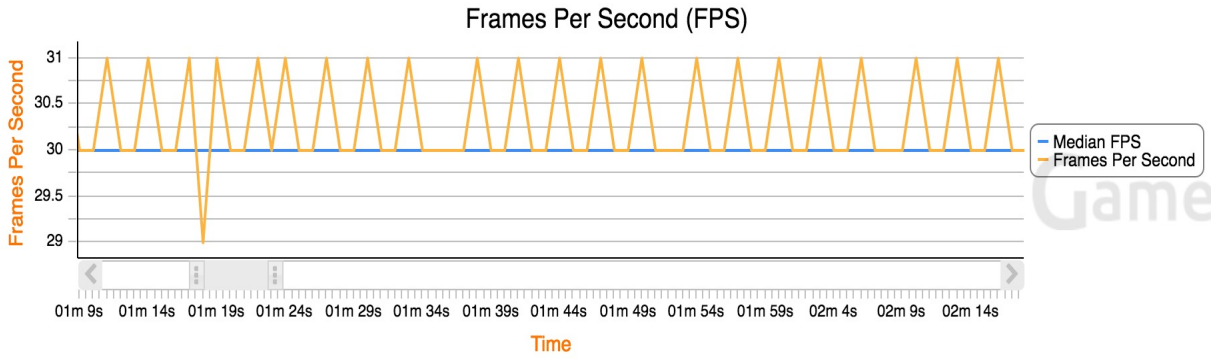

Figure 5.23: Cloud Version Performance - FrameRate

3. CPU Usage: The cloud version has an average CPU usage of 5.29%, 27.16% peak CPU usage and 384Khz - 1.51Ghz minimum and maximum CPU frequencies

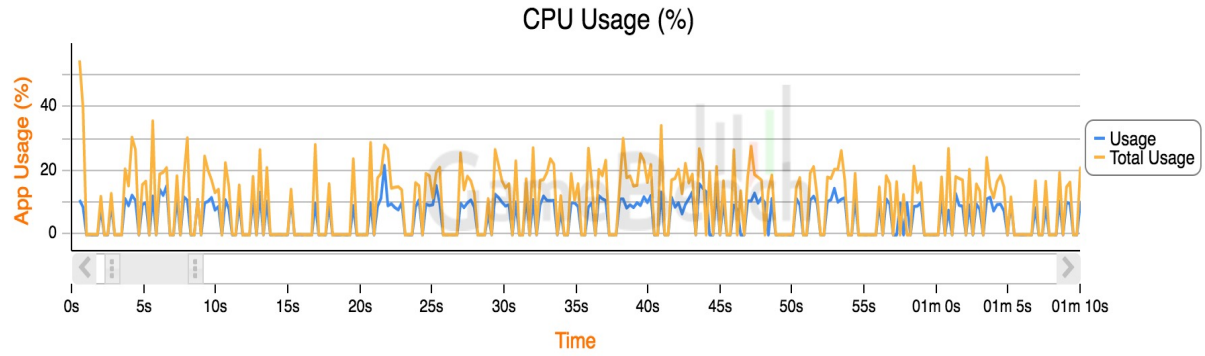

Figure 5.24: Cloud Version Performance - CPU

4. RAM Usage: The cloud version has an average memory usage of 48MB and 50MB peak memory usage

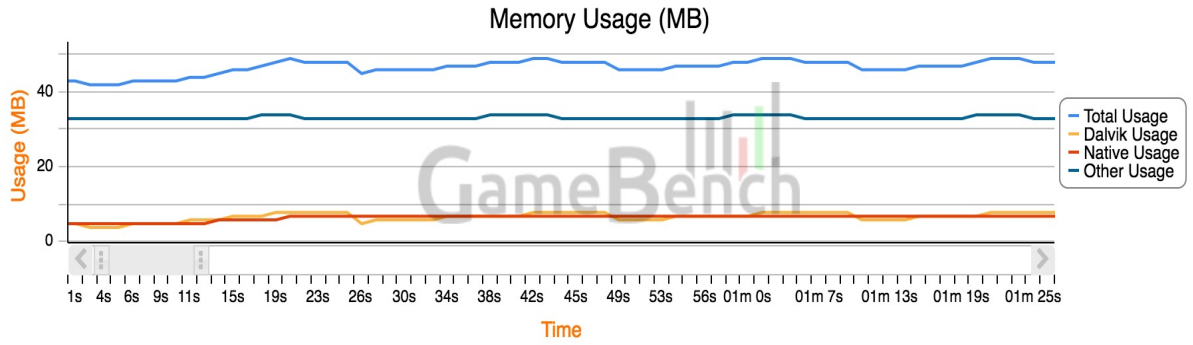

Figure 5.25: Cloud Version Performance - RAM

5. GPU Usage: The cloud version has an average GPU usage of 2.11% and 5.47% peak GPU usage

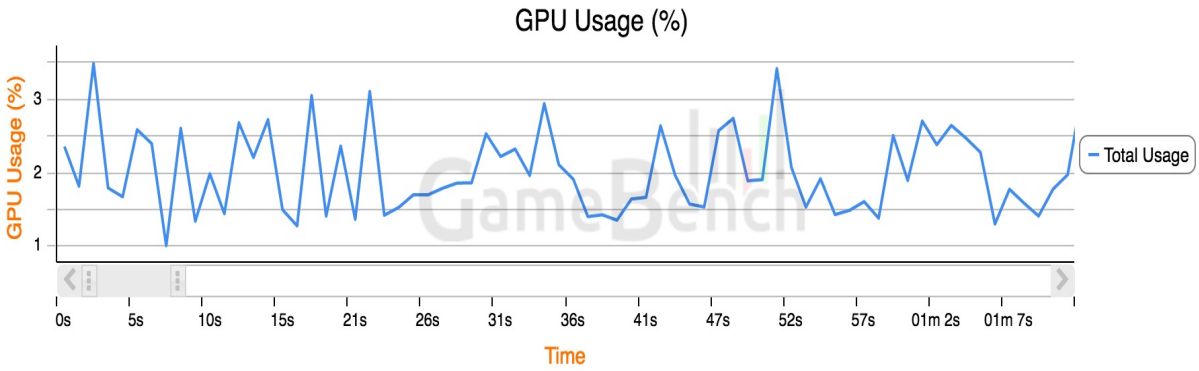

Figure 5.26: Cloud Version Performance - GPU

There's a clear improvement of performance from running everything locally to offloading major tasks onto cloud server/cloudlet. These improvements include reduced CPU and memory usage for the mobile device and reduced battery consumption for the mobile device to keep the application running.

5.5 Weaknesses

Throughout the series of tests, we were not able to obtain information on upload and download speed limits and the available bandwidth for accessing the testing servers. This could be a factor that contributes to the inconsistency of results and why some of the

results were not as we expected prior to the execution of experiments. The other weakness of these test is that we only considered one mobile device as the testing device (Nexus 4). A more powerful device like iPhone 7 or others could result in better performance of the stand alone version application, including latency.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we designed and implemented a live building detection mobile application in a stand alone version and a cloud version. Using the two versions of an applications as test programs, we set up a cloudlet architecture to experiment on the effectiveness of cloudlets. Our work suggests that cloudlets can improve performance of mobile applications in two ways: First, offloading computationally expensive functions to cloudlets can significantly reduce the resource requirements on mobile devices which includes reduced battery usage, reduced CPU/GPU usage as well as reduced memory usage. Second, the cloudlet's pre-fetch ability provides mobile devices with a faster and more stabilized data access without the need to store additional data on the local storage of mobile devices.

6.2 Future Work

Because of the nature of our prototype applications there potentially is a need for a large amount of data storage where the data are content and context based (In our application, we only had buildings data for Western campus only. This could certainly grow to a much larger scale such as a city, a country or even globally). Also worth noting is that the application is only in prototype stage. We can put more work into improving its performance. Additionally, this brings us to the future work of exploring a more generalized cloudlet architecture to adapt to a broad range of mobile applications and sophisticated context and content based cloudlet data pre-fetching which we hope would result in more data pre-fetching efficiency and accuracy.

Bibliography

[1] M Ribicre and P Charlton. Cisco visual networking index: Global mobile data traffic forecast update." cisco, inc., 2014-2019.

[2] Peter Lynch. The origins of computer weather prediction and climate modeling. *Journal of Computational Physics*, 227(7):3431–3444, 2008.

[3] A Ericsson. Ericsson mobility report, on the pulse of the networked society. *Ericsson, Sweden, Tech. Rep. EAB-14*, 61078, 2012.

[4] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[5] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[6] Byung-Gon Chun and Petros Maniatis. Augmented smartphone applications through clone cloud execution. In *HotOS*, volume 9, pages 8–11, 2009.

[7] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[8] Tao Guan, Ed Zaluska, and David De Roure. A grid service infrastructure for mobile devices. In *2005 First International Conference on Semantics, Knowledge and Grid*, pages 42–42. IEEE, 2005.

[9] Tao Guan, Ed Zaluska, and David De Roure. Extending pervasive devices with the semantic grid: A service infrastructure approach. In *The Sixth IEEE Interna-*

tional Conference on Computer and Information Technology (CIT'06), pages 113–113. IEEE, 2006.

[10] Dejan Koavchev, Yiwei Cao, and Ralf Klamma. Mobile multimedia cloud computing and the web. In *Multimedia on the Web (MMWeb), 2011 Workshop on*, pages 21–26. IEEE, 2011.

[11] Paramvir Bahl, Richard Y Han, Li Erran Li, and Mahadev Satyanarayanan. Advancing the state of mobile cloud computing. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, pages 21–28. ACM, 2012.

[12] Debessay Fesehaye, Yunlong Gao, Klara Nahrstedt, and Guijun Wang. Impact of cloudlets on interactive mobile cloud applications. In *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*, pages 123–132. IEEE, 2012.

[13] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. The smartphone and the cloud: Power to the user. In *International Conference on Mobile Computing, Applications, and Services*, pages 342–348. Springer, 2010.

[14] Raghunath Rajachandrasekar, Rupak Nagarajan, G Sridhar, and G Sumathi. Job submission to grid using mobile device interface. In *Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, pages 170–174. IEEE, 2009.

[15] Cynthia Taylor and Joe Pasquale. Towards a proximal resource-based architecture to support augmented reality applications. In *2010 Cloud-Mobile Convergence for Virtual Reality Workshop (CMCVR 2010) Proceedings*, pages 5–9. IEEE, 2010.

[16] Mahadev Satyanarayanan, Zhuo Chen, Kiryong Ha, Wenlu Hu, Wolfgang Richter, and Padmanabhan Pillai. Cloudlets: at the leading edge of mobile-cloud convergence. In *Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on*, pages 1–9. IEEE, 2014.

[17] Paolo Bellavista, Marcello Cinque, Domenico Cotroneo, and Luca Foschini. Self-adaptive handoff management for mobile streaming continuity. *IEEE Transactions on Network and Service Management*, 6(2):80–94, 2009.

[18] Anubis GM Rossetto, Vinicius CM Borges, Alexandre PC Silva, and MAR Dantas. Summit-a framework for coordinating applications execution in mobile grid environments. In *2007 8th IEEE/ACM International Conference on Grid Computing*, pages 129–136. IEEE, 2007.

[19] Francisco Rodrigo Duro, Javier Garcia Blas, Daniel Higuero, Oscar Perez, and Jesus Carretero. Cosmic: A hierarchical cloudlet-based storage architecture for mobile clouds. *Simulation Modelling Practice and Theory*, 50:3–19, 2015.

[20] Sally E El Khawaga, Ahmed I Saleh, and Hesham A Ali. An administrative cluster-based cooperative caching (accc) strategy for mobile ad hoc networks. *Journal of Network and Computer Applications*, 69:54–76, 2016.

[21] Edward E Pingoy and Osvaldo Gervasi. An effective and innovative streaming model for videos in mobile computing. 2013.

[22] Sohail Sarwar, Zia Ul-Qayyum, and Owais Ahmed Malik. A hybrid intelligent system to improve predictive accuracy for cache prefetching. *Expert Systems with Applications*, 39(2):1626–1636, 2012.

[23] Emmanouil Koukoumidis, Dimitrios Lymberopoulos, Karin Strauss, Jie Liu, and Doug Burger. Pocket cloudlets. In *ACM SIGPLAN Notices*, volume 46, pages 171–184. ACM, 2011.

[24] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[25] Jeffrey Richter. *CLR via c#*, volume 4. Microsoft Press Redmond, 2006.

[26] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: a computation offloading framework for smartphones. In *International Conference on Mobile Computing, Applications, and Services*, pages 59–79. Springer, 2010.

[27] Kiryong Ha, Yoshihisa Abe, Zhuo Chen, Wenlu Hu, Brandon Amos, Padmanabhan Pillai, and Mahadev Satyanarayanan. Adaptive vm handoff across cloudlets. Technical report, Technical Report CMU-CS-15-113, CMU School of Computer Science, 2015.

[28] Memcached. https://memcached.org/. Accessed: 2016-10-19.

[29] Jim R Parker. *Algorithms for image processing and computer vision*. John Wiley & Sons, 2010.

[30] Google goggles. https://support.google.com/websearch/answer/166331. Accessed: 2016-10-19.

[31] Mathworks object recognition. http://www.mathworks.com/discovery/object-recognition.html?requestedDomain=www.mathworks.com. Accessed: 2016-10-19.

[32] Opencv. http://opencv.org/. Accessed: 2016-10-19.

[33] Opencv. http://docs.opencv.org/2.4/doc/user_guide/ug_traincascade.html. Accessed: 2016-10-19.

[34] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003.

[35] fyhertz@gmail.com. https://github.com/fyhertz/libstreaming. Accessed: 2016-10-19.

[36] Videolan. https://code.videolan.org/videolan/vlc-android.git. Accessed: 2016-10-19.

[37] Ffmpeg. https://ffmpeg.org/. Accessed: 2016-10-19.

[38] Live networks. http://www.live555.com/. Accessed: 2016-10-19.

[39] Curl. https://curl.haxx.se/. Accessed: 2016-10-19.

[40] Marcin Kalicinski. http://rapidxml.sourceforge.net/index.htm. Accessed: 2016-10-19.

[41] Gamebench. https://www.gamebench.net/. Accessed: 2016-10-19.

Appendix A

Application Preparations and Detailed Implementation

A.1 Preparations

A.1.1 Generate pre-defined building classifiers for Computer Vision functions

1. Gather building source images: Taking one building as an example, it may have several distinguishable features, such as pointy roof, glass wall, colourful paint. Each pre-defined building classifier represents a single feature. Therefore, a building may have several related classifiers. To generate each classifier, firstly a video clip is taken to capture one building feature from different angles all around. Then, the video clip is chopped into images using the ffmpeg function:
FPS Video Filter: Output one image every second, named out1.png, out2.png, out3.png, etc.

ffmpeg -i input.flv -vf fps=1 out%d.png

2. Mark-up source images: After obtaining an image group about a building feature, these images are then to be marked-up by recording the top-left pixel coordinate and the bottom right pixel coordinate that the building feature are bounded within a rectangle box. Additionally, we prepared a group of negative images(background images). These images do not contain the target building feature and is for the purpose of comparison in later classifier training algorithms. Finally, all these images are fed through the OpenCV cascade classifier training functions to generate classifier file named after the building's feature.

Firstly create samples:

```
opencv_createsamples
  -vec positive_example/middlesex.vec
  -info middlesex_img/info.txt
  -bg negative_example/bg_bg.txt
  -num 162
  -w 300
  -h 300
```

-vec is the output samples' destination folder after running opencv_createsamples command

-info is path to a file that contains a list of each positive image's file name and the corresponding coordinates of the feature position in the image

-bg is path to a file that contains a list of negative images

-num is the total number of positive images

-w & -h are the width and height of the output samples of the feature in pixels

After running opencv_createsamples, it generates an output sample group bases on all the images that were fed to the algorithm. Next, runs the training command:

```
opencv_trancascade
  -data middlesex_classifiers/
  -vec positive_example/middlesex.vec
  -bg negative_example/bg_bg.txt
  -numPos 20
  -numNeg 2
  -numStages 8
  -w 300
  -h 300
```

-data is the output classifier destination folder

-vec is the sample folder previously generated by opencv_createsamples command

-bg is path to a file that contains a list of negative images

-numPos & -numNeg are the number of positive and negative images that choose to be used in each stage of the training

-w & -h are the width and height of the output samples of the feature in pixels

It takes hours for training algorithm to finish, and in the end the classifier is generated and stored as an XML file. The file is named after the building name concatenates with the description of the significant part of the building. Such as, MiddleSexCol-

lage_MainTower.xml. The same process repeats for other features of the same building as well as other buildings.

A.1.2 Record down each building's geographic location

As previously stated in the application design, geographic locations of the application user is vital. In order to use a user's geographic location, a pool of buildings' geographic location is created and stored in a XML file format where each building's name pairs with its geographic location. This is the building list file used in both version of application.

A.2 Application source code

A.2.1 Stand Alone Version Components

Constants.java: It pre-defines file types that is used in application(image, utility, classifier, building info), types of messages that helps communication between separate function groups, and database & file server address that application connects to.
Camera_Activity.java: The main activity of the application. This stand alone version can be grouped into 7 functional groups: **Messenger**, **File Manager**, **GPS Locator**, **Camera**, **Building Info**, **OpenCV Detection** and **UI**. Each will be stated separately for clearer presentation.

Messenger and its handler:

Messenger and handler in this app is designed to let different functional group communicates with Main Activity(Main Thread), so that Main Activity can direct the next process based on what the return message contains. Upon Main Activity's onCreate function, a main_activity_messenger is initialized. Together, it set-ups its handler to main_activity_message_handle where it states how to handle different type of messages. There are a total of 6 different types of messages in the current version, and could easily expand more following the same loop design(more of future works to perfect the application):

MAIN_ACTIVITY_MESSAGE: The only purpose of this message is to let other functional groups know where to pass back return messages. It carries no message but the *replyTo* attribute being *main_activity_messenger.*

FILEMANAGER_MESSAGE: This message's purpose is to let File Manager func-

tional group to notify Main Thread that a previously initialized remote file download task is complete. The application currently consists of 4 different types of files: *UTILITY_FILE, CLASSIFIER_FILE, BUILDING_INFO and IMAGE_FILE*. A helper class *FileCompletionMessage* is also created so that File Manager can send Main Thread the same file completion message but with different file type specified.

BULDING_INFO_ONCLICK: This message's purpose is to notify Main Thread what is user's interaction with *UI*. As buildings to be recognized by background algorithms, there will be extra information about these buildings showing to user. These informations are shown in a form of *UI* elements(such as image thumbnails, web links. etc..), where users can click or expand on them for details. This message is to tell Main Thread exactly which element user clicked on. Additionally, since current app has 4 types of different extra information about a building (referred as artifacts of building), a helper class *ArtifactOnClickMessage.java* is created to allow *BULDING_INFO_ONCLICK* to send the same message but with different artifacts as content.

UPDATE_EXPANDABLELIST_MESSAGE: This is the actual UI update message to tell Main Thread a list of buildings informations to ready for shown on *UI*.

GPSLOCATOR_MESSAGE: This *GPSLOCATOR_MESSAGE* serves for the GPS Locator service to communicate with Main Thread and let it know user's latest GPS location.

REFRESH_UI_MESSAGE: This is only for debug purpose telling Main Thread to show which building classifier file is ready and loaded on use.

File Manager & functions

FileManager.java: Although this application is to run as stand alone, it is designed in the way that needs to find files from server if they are not present locally. As previously mentioned in *FILEMANAGER_MESSAGE*, such files are categorized into mainly 3 types plus 1 type of string data that describes building artifacts.

UTILITY_FILE: utility files are what essential for the application to start up. Currently there's only one file listed in this category, which is the building list file. It listed all the buildings that the application is able to recognize.

CLASSIFIER_FILE: as previously explained in preparation, each classifier file corresponds to one feature of a building

IMAGE_FILE: images about the building, which will be shown as extra informations about a building after algorithm successfully detects the building on scene

BUILDING_INFO: This is the string data which, application will firstly query server's database, and get the result of a list describing each artifacts about a building as of what image links, video links, text description about the building.

ExecuteQuery.java: This is just a helper class where enables android to send SQL queries and receive result with data server.

GPS Locator & Location Analysis function

This is the function group that periodically retrieves user's GPS location. As has been previously stated, this is for the purpose of reducing non necessary calculations for the Classifiers & OpenCV Detection group. There are two ways to retrieve the user's GPS location, either by Wifi/Cell Tower or by satellites. Here implements both and the more accurate will be chosen. Additionally, after the first retrieve, there are two factors that will trigger location update: how many seconds in time that has passed since last retrieve, and how far the mobile has moved from last retrieve location. Currently they are set to 1 minute and 10 meters. Also, this function group covers the location comparison function, which is the *LocationAnalyser* class.

GPSLocator.java: This is the major class in GPS Locator function group, which runs as a background service of android on a separate thread. It specialized in finding the mobile's latest GPS locations through the use of Android's *LocationManager* Class. Also, by implementing its the *onLocationChanged* function, it sends every updated location to Main Activity through Messenger functional group.

GPSCoordinateGenerator.java: This is a dummy class created for the purpose of testing and debugging, instead of actually finding user's GPS locations, it just randomly generates some relevant locations from the building pool.

LocationAnalyser.java: This class's job is to do the location comparison. Upon creation of this class, it loads the *UTILITY_FILE*: building list, parses all the xmls into a buildings pool, containing each building's name and their corresponding GPS locations(Latitude and Longitude). On each location update, the Latitude and Longitude is provided to this class's findNearbyLandmarks function, it returns with an ArrayList of all nearby buildings that listed in the pool. Currently, the radius of nearby buildings is defined as 100 meters.

Camera & functions

This is the function group that connects to the mobile camera and generates view of camera stream on mobile screen for user. The functions are implemented directly in Main Activity: *Camera_Activity*. The most important function in this group is the *onCameraFrame* function, which acts as the bridge between mobile camera and the actual view on mobile screen. Because of this bridge, it enables the Classifiers & OpenCV Detection group to intercept mobile camera frames and modify the frame accordingly base on detection result.

Classifiers & OpenCV Detection

This is the core function group of the application. Its job is to detect buildings from what mobile camera stream captures and mark them out accordingly. As mentioned just above, because of the bridge set up by the *onCameraFrame* function, this function group is able to interpret each frame captured by camera before they are shown on phone screen.

SingleBuildingClassifierSet.java: This is the class containing each building's classifier information. Such as: building's name, number of features in the building and each of this building's classifier file name etc..The major purpose of this class though, are two of the followings:

1. Prepare/Loading the actual classifier object from reading the specific building's classifier file.

2. Provides a detection function which, given a single camera frame, it can determine specifically whether this building exists and save the result for later frame modification use.

BuildingClassifierSetList.java: This class groups up each *SingleBuildingClassifierSet* object and provides group functions to execute *SingleBuildingClassifierSet*'s functionality. The reason behind is, when a list of building detection candidates are generated by Location Analysis function, it is often comes with few buildings. Therefore, with group functionalities provided, it ease the coding logic as well as providing a cleaner structure.

DetectionBasedTracker.java: Here is the OpenCV native detection algorithm with Java wrapper. This class's functionality get included in SingleBuildingClassifierSet, initialized when loading classifiers and is executed when SingleBuildingClassifierSet's detection function runs.

Given the 3 major classes in Classifiers & OpenCV Detection functional group, in order for the detection process to work, it also closely rely on 3 other functional group. The GPS Location group, File Manager group as well as the Camera group. Recall that on Location group's update, the *LocationManager* can generate a list of buildings that are within 100 meters range of user's current location. The return of this list then triggers a instance of *BuildingClassifierSetList* to be generated base on the list. Then File Manager is to download each of these building's classifier files. This design is based on the assumption that the user shouldn't ever need all the classifier files of each single building in the building pool. Because when building pool grow larger, for example containing all buildings and landmarks of total 10 cities. The user may ever visit couple cities and only couple small areas of a single city. Therefore, download accordingly whenever the files are needed saves much storage space on mobile application. As that being said, after each classifier file successfully downloaded, *BuildingClassifierSetList* will

load each of them into the corresponding building's *SingleBuildingClassifierSet*. In the mean time, through the bridge that created by Camera group's *onCameraFrame* function, *BuildingClassifierSetList* will run group detect function, if any building is detected, they will be mark out on that frame, and then the frame will be shown to user on screen.

Buildings & Building Artifacts informations

This is the function group which provides all the informations about each building.

BuildingLocation.java: Generated from building list pool, contains each building's name and GPS location.

Building.java, Artifact.java and BuildingList.java: Building contains information the building that is to show to users. Such as images, links, videos..where Artifact is a parent class of all these types of information and BuildingList is the container for group of Buildings. These 3 classes are made in order to work together with *UI*, for easier presentations on mobile screen when such a building is detected.

UI

ColorPool.java, ExpandableListAdapter.java, ImageAdapter.java: This is the UI function group that responsible for rendering building's informations in cleaner way as well as letting users to interact with information such as expanding images, click to watch video links and visiting related websites.

Prototype version 2 components

Because is it a variant version comparing to stand alone version, some of the components functions in the same or similar way, and some functions are just simply mirrored from mobile onto server. So those components will not be restated again in detail, but rather focus on how the new stuff works.

Server Entity:

SingleBuildingClassifierSet, BuildingClassifierSetList, BuildingLocation, GeoPoint, LocationAnalyser, CurlFileDownloader, MySqlCpp: These are mirrored functions from stand alone version.

Server_Main: This is the Main thread of the server entity. As stated in application cycle, it generates the 2 major functional groups: **Streamer** and **Client Reception**. In addition, the Client Reception springs off another major functional group: **Video Modification**. These 3 groups completes most of the application cycle on server entity.

Client Reception

Client Reception is the front tier functional group responsible for communicating with each connecting client. For each connected client, it spawns a unique session for the

specific client to interact with. Session interactions includes:

1. Sending & receiving messages: inbound video session description protocol(sdp), inbound client's GPS location, outbound detected building's information.

2. Spawning a unique Video Modification functional group which maintains a private live stream port for receiving inbound client's video stream, run OpenCV detection functions on the stream, as well as re-encode the modified video ready for stream back.

3. Notice Streamer functional group to be ready for an additional live stream back to client.

MsgReceiverServer.cpp: This is the class that responsible for listening on new client connection request. When a new client initializes a connection, This class creates a new *ClientMsgConnection* instance, hands off the new client to this newly created session and then keeps on listening to the next client connection request.

ClientMsgConnection.cpp: This is the class that provides unique session and functions for each client connected. Its *recieveMessage* function constantly listens to what message client sends next and hands them off to *parseMessage* function which determines what type of message it is and handles accordingly. As mentioned just above, there are currently 3 types of messages:

1. Inbound video session description protocol(sdp): which contains informations about how to read the inbound stream that this client is sending. When this message is received, ClientMsgConnection spawns off a Video Modification session as well as notifies Streamer group to be ready for streaming back a new video stream.

2. Inbound client's GPS location: When the new client location is received, it updates this client session's unique LocationAnalyser and generates a new list of nearby building candidates. Major process on this one is mirrored from stand alone version.

3. Outbound detected building's information: This is not a inbound message, instead, when the session's OpenCV detection functions that run by Video Modification group detects some building within any video frames, it generates a list of these buildings as a message. This list will be sent to client's mobile through ClientMsgConnection's sendMessage function. Which helps mobile to render side informations about detected buildings accordingly.

ClientMsgParameters.cpp: This is just a helper class to let *ClientMsgConnection's parseMessage* running on a separate thread from *recieveMessage* function. So that parsing message does not block any inbound message receiving.

Video Modification

Video Modification is another key functional group which enables client to outsource video stream building detection functions. It enables server to read inbound video packet

stream of a given client session, decoding the stream into actual frames, which in turn makes it possible to run OpenCV detection functions and modifications on frames. Then re-encode frames into packets and pass along to server's Streamer functional group for streaming back.

VideoState: This is the video information class which reads the SDP message that is passed along and obtain this particular stream's property. Such as: video's format context, stream index, frame rate..etc.. Additionally, base on these information, it also prepares the video codec ready for both decoding and encoding this stream.

PacketQueue & FrameQueue: Before going into video decoding and encoding, we needs to first look at two helper data structure, *PacketQueue* and *FrameQueue*. The whole video stream handling process can be summarized as: reading video packets from stream, decode packets into frames, detect and modify frames, finally re-encode frames into packets. In this process, these two data structure are in place to make sure that each step of the process can run in parallel without having to wait on previous step as much as possible. The content of these two data structures are name implied. *PacketQueue* stores stream packets and *FrameQueue* stores stream frames.

RTPReceiver: This is the hand off destination from *ClientMsgConnection* after receiving SDP information. It implements ffmpeg libraries of video processing. Including first setup network functionalities through *avformat_network_init* so that client's live stream can be read. It also initializes *VideoState*, creates *VideoDecoder* and *VideoEncoder*, as well as creating corresponding *PacketQueue* and *FrameQueue* to connect the Decoder and Encoder process.

VideoDecoder: As its name implies, this class is to decode inbound stream packets into actual frames. Current decoding codex is H264. After *RTPReceiver*'s initialization on network component, *VideoDecoder* is able to read inbound video stream base on a given specific SDP. It runs two separate threads, *readPktLoop* keeps reading packets off stream and save them to Decoder *PacketQueue*, while *decFrameLoop* keeps reading from Decoder *PacketQueue*. After reading each packet from Decoder *PacketQueue*, *decFrameLoop* function mainly does three steps: Firstly decode packets into actual video frames, then it runs OpenCV detection on the decoded frame and finally save the modified frames onto a Video *FrameQueue* that ready for *VideoEncoder* to fetch. To note, that in *VideoDecoder* there's also 2 helper function *avframe_to_cvmat* and *cvmat_to_avframe* which can convert the video frames between ffmpeg compatible format and OpenCV compatible format. Which is convenient during a complete *decFrameLoop* run.

VideoEncoder: Here is the final destination of the client's video stream in the whole Video Modification process. VideoEncoder reads from the Video FrameQueue that previ-

ously queued by VideoDecoder and compresses each video frame back into H264 packets for later Streamer's use. The encoder process's name is encFrameLoop and it is the only function in VideoEncoder.

Streamer

Streamer is a separate functional group on server. It initializes on server's application start and it is responsible for streaming modified video back to corresponding clients. It is a separate group because it does not interact with other groups except that when each client's sdp is received, the *ClientMsgConnection* register a unique session on Streamer base on SDP information together with the designated Video *FrameQueue* where modified video will be queued. Streamer group then only listening to client's rtp request for streaming session's video back. The Streamer function group is implemented base on an open source video streaming library Live555.

VideoStreamer: This is the main hub of Streamer group. It creates a *RTSPServer*(Live555 native library class) instance on start, which provides the service of outbound streaming multiple sessions of videos. For each video stream, it has a unique RTP URL. Clients connects via the given URL then *VideoStreamer* will start to sending packets towards client.

StreamerParameters: This is just a helper class for passing *VideoStreamer* necessary parameters. Parameters including: each session's stream name, the source *PacketQueue*, video statistics.

VideoLiveServerMediaSubsession: Inside each SDP generated stream sessions, there are also sub-session of stream. They are distinguished by source codec, source streaming method as well as stream index. In this case, it is H264 codec, and live stream(oppose to existing video clip), and its a video stream(oppose to audio stream). So *VideoLiveServer-MediaSubsession* represents this exact sub-session of video stream. Since the purpose of the application is outsourcing image detection component, audio sub-session is not needed.

FramedLiveSource: *FramedLiveSource* is a component in the sub-session stream. it specifies where the source video is, and provides function on how it should be read. In our case here, it has a reference on the *PacketQueue* generated by *VideoEncoder* and provides *deliverFrame* function to specify how to read each stream packet.

FramedLiveSourceParam: This is just helper class for *FramedLiveSource* to fetch each packet from *PacketQueue*.

Mobile Entity:

Constants.java, GPSCoordinateGenerator.java, GPSLocator.java: These are mirrored functions from stand alone version.

MainActivity.java: Because of the structural design that allows mobile entity to outsource most of the processes onto server entity, the functional groups on mobile entity is much simpler. The GPS Locator, which is a mirror as in stand alone version, the Outbound Streaming group and Inbound Streaming group. As the name implies, Outbound Streaming enables mobile to stream camera views onto server in the form of live video stream, and Inbound Streaming group queries server to get the modified video stream back on screen view after server's detection process.

Outbound Streaming

Outbound streaming is implemented by using libstreaming library. Libstreaming is an open source API that allows outbound stream of an android OS device. The streaming uses RTP with UDP on transport layer, with H264 video compression method on packetizing video frames.

Inbound Streaming

Inbound streaming is implemented by using libvlc library. Libvlc is an open source API that provides VLC Player's functionality on android OS. Including the capabilities of playing stream videos over network with RTP URL link provided.

VideoPlayer.java: This is the implementation class of libvlc. It starts up by initializing an vlc instance, and provides functionalities on sizing the video, pause, resume.. etc.. on video streaming.

Curriculum Vitae

Name:	XuTong Zhu
Post-Secondary Education and Degrees:	La La School La La Land 1996 - 2000 M.A. University of Western Ontario London, ON 2008 - 2012 Ph.D.
Honours and Awards:	NSERC PGS M 2006-2007
Related Work Experience:	Teaching Assistant The University of Western Ontario 2008 - 2012

Publications:

La La