SPEEDING UP AND AUGMENTING MOBILE DEVICE APPLICATIONS USING ACTORS AND CLOUD COMPUTING

by

Pratik Patel

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved by the
Examining Committee:

____
Carlos Varela
Thesis Advisor

____
Stacy Patterson, Member

____
Elliot Anshelevich, Member

Rensselaer Polytechnic Institute
Troy, New York

November 2014
(For Graduation December 2014)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to thank my advisor, Professor Carlos A. Varela, for all of his support, encouragement, and inspiration. He has been a true mentor to myself and has made my time at RPI not only fun but incredibly educational. The Worldwide Computing Lab has been a great learning experience and I would like to thank all of my colleagues for their input and dialogue. The RPI professors and staff have helped me immensely throughout my courses and have taught me interesting material. A big thanks to my friends here at RPI who have made my time here extremely enjoyable. Last but not least, I would like to thank my family for their endless support and encouragement during my RPI experience.

ABSTRACT

The use of mobile devices has been steadily increasing, for example, smartphones are expected to be used by 69.4% of people worldwide by 2017 [1]. User expectations have also increased as the computational capabilities of mobile devices improve. As a result, software applications need to perform ever more complex and data-intensive tasks to address user expectations. Because of resource limitations in mobile devices (e.g., battery, limited network connectivity) we investigate the cloud computing paradigm as a means of augmenting mobile device capabilities.

In this thesis, we first study the potential reduction in computation time for a mobile device application that offloads part or all of its execution to remote resources, such as a tablet, a laptop, a desktop, or a private/public cloud. Second, we apply the actor model of concurrent computation to reconfigure a distributed application from the mobile device to the cloud. Specifically, we use the SALSA actor programming language, which allows developers to easily create computationally intensive applications that can be broken apart and migrated to various computational resources. Since SALSA programs compile down to Java byte code, we can readily run them in the Android Operating System through an extension to the SALSA language. Lastly, we aim for a separation of concerns by specifying policies that govern when and where to move actors, separately from the functional application code.

Using our Mobile Cloud Computing using Actors (MobileCCA) approach, as applied to a face recognition task, we observed speedups on average of ~5x in the private cloud with respect to doing the computation on the mobile device. Furthermore, we were able to perform the face recognition task on a database of 1000 faces for 400 people, a task beyond the resource capabilities of the mobile device alone. MobileCCA therefore illustrates not only the potential to speedup computations in mobile devices and save battery, but also to enhance the power of mobile applications.

1. Introduction

1.1 Motivations

The mobile device market has expanded greatly over the past years and shows no indication of slowing down. As smartphones and tablets become more affordable there still proves to be growth potential in an already dominant market share. Statisticians expect that smartphones will grow to 69.4% usage worldwide by 2017 [1]. Smartphones along with tablets provide users with the ability to run various applications without the shackles of a constant power source. These devices, however, have limitations in the type of applications that can be run. Smartphones have a limited CPU/GPU compared to that of a stationary computer or cloud. Most importantly the battery can only handle so much before it requires a charge. With these limitations we can see that a group of applications known as computationally expensive can run rather inefficiently on a mobile device when compared to other computational resources available, such as desktops and private/public clouds.

Computationally expensive applications can include, face recognition, question and answering systems, video recognition, etc. As the world moves towards mobile devices' the requirement to run such applications is essential. The problem that arises is how can we run these computationally expensive applications on mobile devices efficiently? Mobile Cloud Computing provides an elegant solution: the ability to offload various tasks to private or public clouds for computation lessens the onus on mobile devices' capabilities and introduces the idea of shared computing.

Mobile Cloud Computing, however, requires application developers to choose when and where to offload certain computational tasks. There are a number of Cloud-based augmentation techniques that provide different ways for a system to handle a computationally expensive application [2]. This thesis focuses on the use of the actor-oriented programming language SALSA with Mobile Cloud Computing [3]. We introduce a Cloud-based augmentation approach we call *Mobile Cloud Computing using Actors (MobileCCA)*. The idea behind the MobileCCA approach utilizes the dynamically reconfigurable property of SALSA actors when running mobile applications.

1.2 Contributions

We have developed a cloud-based mobile augmentation strategy, known as MobileCCA. By utilizing the actor model, we are able to create dynamically reconfigurable distributed applications that can be run on various computational resources. The natural mobility of actors allows for a cohesive use in Mobile Cloud Computing. In devices that are dependent on battery and network connectivity, the ability to move actors dynamically can be important. We have seen that because actors are an encapsulated state, the distribution and migration of actors can be accomplished much easier than with other techniques.

We have created a number of policies to distribute actors based on certain requirements. MobileCCA advocates for a separation of concerns for application developers. Developers now only need to focus on the implementation of their applications. The MobileCCA policies, separately specified, can then decide when and where to move actors. By utilizing these policies with MobileCCA, dramatic changes in the environment and application can be handled.

We have implemented the MobileCCA approach using the SALSA programming language. In order to run SALSA code on the Android device, we created a SALSA add-on to start a theater on the Android device. In this thesis, we have extended the usability of Android SALSA by creating applications that can run on the Android device. To test the advantages of MobileCCA implemented in SALSA we have created a useful application, face recognition. This example application showed that MobileCCA provided a developer friendly approach to Mobile Cloud Computing.

1.3 Structure of Thesis

The structure of this thesis is as follows: In Chapter 2, we begin with an overview of Mobile Cloud Computing, the use of computational offloading, the actor model, and the SALSA programming language. In Chapter 3, we look at the use of SALSA in mobile devices and how it can be incorporated into Mobile Cloud Computing. Then, Chapter 4 examines our approach known as Mobile Cloud Computing using Actors, and Chapter 5 examines a computationally expensive application, face recognition. In Chapter 6, we review related work, and finally, Chapter 7 concludes this thesis and discusses future work.

2. Overview

2.1 Mobile Cloud Computing

Mobile devices by nature have their limitations; they are limited by their processing power, battery, and storage. By leveraging the benefits of cloud computing, we are able to mitigate these limitations. Cloud computing by the NIST definition is the model for on-demand access to a shared pool of configurable computing resources [4].

We can intuitively see that given the diverse network capabilities of a mobile devices, they can be easily coupled with cloud computing to provide an elegant solution to the devices' limitations. Through utilizing a cloud's resources, mobile devices experience the following returns and restrictions: better battery life, enhanced processing power, increased data storage, broader network connectivity, decreased security, and increased cost.

Table 2.1: Computational Resources Statistics

Source	**CPU Speed (GHz)**	**Cores/ Unit**	**RAM (MB)**	**Storage (GB)**	**Battery (mAh)**
Samsung Galaxy S5 [5]	2.5	4/1	2048	16/32	2800
Samsung Galaxy Tab Pro 12.2 [6]	1.9/1.3	4/2	3072	32	9500
Amazon EC2 C3 Large [7]	2.8	2/3.5	3840	32	Constant
Amazon EC2 C3 X-Large [7]	2.8	4/3.5	7680	80	Constant

2.1.1 Better Battery Life

One of the most important benefits from Mobile Cloud Computing is the ability to preserve battery on a mobile device. From Table 2.1, it is obvious that one limiting factor for any mobile device is its battery. Whether it is performing various computations, transmitting data, or simply sitting on a table, when a mobile device is unplugged from its charger it is consuming battery. By computational offloading, the mobile device is able to alleviate the workload of its CPU and therefore reduce battery consumption. Computation offloading is not trivial because by offloading to a cloud, we incur a network overhead. Depending on the type of data that is being transmitted and the type of computation being performed, offloading may not be the best solution all of the time.

2.1.2 Enhanced Processing Power

Through private and public clouds, we obtain the use of high performance computational resources. The natural mobility of mobile devices allows any user with network connectivity the access to a vast supply of computational power. From Table 2.1, we can see that the Galaxy S5 and Tab Pro 12.2, a smartphone and tablet respectively, two top devices to date, still lag behind an average Amazon EC2 instance. Amazon Elastic Compute Cloud (Amazon EC2), is a web service that provides resizable compute capability in the cloud [8]. Through offloading to the cloud, mobile applications can experience faster processing time. For example in Chapter 5, we look at experimental results for the face recognition application. To get a sense of the differences in processing power for each computational resource, we compare the face recognition application run solely on the mobile device vs. private cloud.

2.1.3 Increased Data Storage

In Table 2.1 although the mobile devices have comparable data storage to an average Amazon EC2 instance (C3 Large), it is important to note that it is not dedicated to the targeted application. Mobile devices are constantly experiencing I/O whether it is background processes from the OS or other applications run by the user. Mobile devices' data storage can also be limited by personal data use: music, photos, and videos can take up a large portion of the storage and limit the amount of space an application can use. For example in the face recognition application, a database of face images is required to train over. In the mobile device the number of images that can be stored may be significantly lower than that in the private or public cloud, which leads to erroneous and high uncertainty results, as shown in Chapter 5.

2.1.4 Broader Network Connectivity

The benefit of using a mobile device is the access to a number of wireless networks and telecommunication technologies. Whether it is wireless internet, 3G, 4G, etc., within reason there is always some sort of network connectivity available to a user at all times. The 3G technology provides a greater coverage than 4G but is slower; 4G coverage has been growing as of late and provides similar speed with wireless internet but can consume more battery. Wireless internet allows for high speed communication while having the lowest battery consumption amongst all options [9]. Compared to wireless networks, 3G and 4G allow for greater mobility, but can incur restrictions from the phone provider such as data limitations and bandwidth throttling. Wireless networks can allow for fast connections for devices under the same network but can experience bandwidth and congestion problems.

2.1.5 Decreased Security

A pitfall of using Mobile Cloud Computing with mobile devices is the risk of security vulnerabilities. When data is migrated to the cloud there is always the danger of losing it to adversaries. Although cloud distributors are constantly trying to enhance their security, attacks as simple as phishing and brute force of personal accounts can lead to the loss of vulnerable data.

2.1.6 Increased Cost

With cloud computing you are paying for resources on-demand. This brings the factor of budget into consideration when using Mobile Cloud Computing. Although public clouds offer an increase in performance when dealing with applications, it is important to look at the cost-benefits of offloading. Some applications can experience a marginal gain in performance when moving to public resources, and for a strict budget user this move can be illogical.

2.2 Cloud-based Mobile Augmentation

Cloud-based Mobile Augmentation is the model that leverages the use of cloud resources to optimize the computing capabilities of mobile device applications [2]. In this thesis we look into the approach of computational offloading for Mobile Cloud Computing.

2.2.1 Computation Offloading

Computational offloading involves migrating computational tasks from the mobile device to another computational resource and performing the computation there, sending the results back when complete. Defining tasks to offload can be done prior to execution or dynamically during runtime. The question that arises is when to offload computation based on the statistics of the application and surrounding environment?

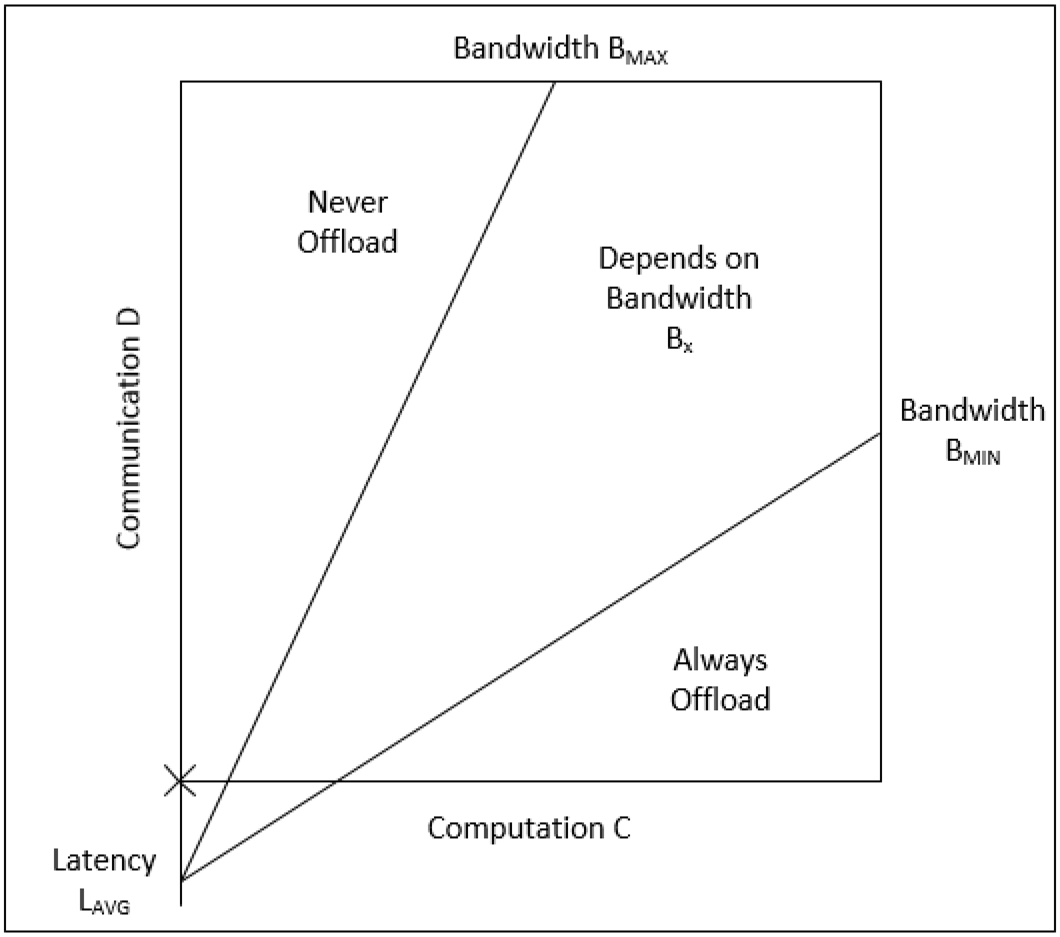

Figure 2.1: Offloading Decision Graph [10]

Table 2.2: Cloud Application Feasibility Matrix [11]

Applications	Compute Intensity	Network Bandwidth	Network Latency	Offload Decision
Web-mail	Low	Low	Low	Never
Social Networking	Low	Medium	Medium	Never
Web browsing	Low	Low	Low	Never
Online Gaming	High	Medium	High	Depends
Chess	High	Low	Low	Always
Face Recognition	High	Medium	Medium	Always
Video Recognition	High	High	High	Always
Question and Answer	High	Medium	Medium	Always

Figure 2.1, extended from [10], represents a decision graph for offloading computation. The decision equation for offloading computation is:

$$\text{if } \frac{C}{f_{remote}} + \frac{D}{B} + L < \frac{C}{f_{local}} \text{ then offload.} \tag{2.1}$$

The goal is to ensure that the time it takes to finish the computation on the remote machine plus the overhead of sending the data is less than doing the computation locally. For Figure 2.1, we assume that performing the computation on the remote machine occurs instantaneously. Therefore we can reduce Equation 2.1, to –

$$\text{if } \frac{D}{B} + L < \frac{C}{f_{local}} \text{ then offload.} \tag{2.2}$$

Now we strictly compare the network overhead time to the local computation time. There are three regions in the graph: the region that indicates you should never offload, the region where you should always offload, and the middle region where offloading depends on the bandwidth of your network connection [10]. In the “never offload” region, the communication overhead is large while the computation is fairly low. This range represents network driven applications such as web browsing and social networking. In the “always offload” region, computation is large and communication overhead is fairly low. Applications that would benefit from offloading include face and video recognition, and question and answer services. Finally, the "middle region" depends on the application at hand. For example for online gaming, responsiveness is crucial; therefore, we should only offload computation when it doesn't affect gameplay. The type of game can be a factor when offloading especially when on a mobile device. Action and adventure games can lead to a large amount of data being transferred. Therefore offloading can reduce the responsiveness of the game. Puzzle games, chess specifically, can benefit from offloading as very little data is being transferred, and the computation of chess moves can greatly benefit from being processed on a larger computational resource.

We can see that as the bandwidth of the connection increases, the "always offload" portion of the graph increases as well. We can compare this to Mobile Cloud Computing by assuming B_{MAX} = WiFI, and B_{MIN} = 3G. Latency is another factor for offloading decision. If the average latency of an application is large, we can see that L_{AVG} will move down on the y-axis causing the portion of the "always offload" to decrease.

2.2.2 Current Approaches

There are various cloud-based mobile augmentation approaches that use computational offloading in Mobile Cloud Computing.

CloneCloud [12] is a cloud-based mobile augmentation approach that involves cloning an entire mobile devices platform to the cloud. This allows mobile applications to run locally on the device as well as remotely on the cloud's VM. This approach aids the developer as the same code is run on the device and cloud simultaneously. Problems that arise are that cloning a mobile device can be expensive and data vulnerabilities appear when data is moved to the cloud. *CloneCloud* is also strictly used on distant fixed clouds, like an Amazon EC2 instance. This can hurt budget users as they cannot run the computation on their hardware or other mobile resources.

MOCHA [13] is a hybrid cloud computing approach that involves application partitioning and migration. *MOCHA* has the ability to partition computationally intensive applications through two algorithms, and distribute them to various cloud resources. The fixed algorithm will distribute work equally to all resources, while the greedy algorithm will distribute to resources with the fastest response time. Unlike *CloneCloud*, the ability to migrate partitioned applications reduces the overhead of migrating an entire mobile clone. The problem of partitioning the application, however, is left to the developer.

Hyrax [14] is an approach that focuses on a cluster of mobile devices for computation offloading. Similar to MapReduce [15], *Hyrax* allows a large dataset to be processed over a cluster of resources, in this case mobile devices. Mobile devices are connected to a central server and issued various tasks to complete. Devices can communicate with the central server to send back results, as well as with other mobile devices in the cluster. This approach allows applications to be distributed to other mobile devices rather than cloud resources. With this, however, the drawback of lower computational power and storage occurs.

Table 2.3: Cloud-based Mobile Augmentation Approaches

Approach	**CPU Alleviation**	**Battery Prolonging**	**Network Overhead**	**Developer Concerns**	**Security**
CloneCloud	Medium	Medium	High	Low	Low
MOCHA	High	High	Medium	High	Medium
Hyrax	Low	Low	Medium	Medium	Medium
MobileCCA	High	High	Medium	Low	Medium

In Chapter 4 we describe the approach, *MobileCCA*, Mobile Computing using Actors, which involves the use of actor-based programming languages and the migration of actors.

2.3 Concurrent Computing

As opposed to sequential computing that requires tasks to end before a new one begins, concurrent computing allows tasks to be executed simultaneously during overlapping time periods. However, the ability to separate tasks and execute them in parallel can sometimes reduce overall runtime. Therefore, concurrent computing offers both advantages and disadvantages. The benefits of concurrent systems is that tasks do not need to wait on others to complete to make progress [16]. Each computational task in a given system has a separate execution point or “thread of control”. Disadvantages of concurrent programs can occur based on the ways tasks interact with each other. For programs that require each tasks to access shared memory, precautions need to be taken to prevent race conditions and deadlock.

Concurrent programming consists of four basic approaches: sequential programming, declarative concurrency, atomic actions, and message passing. In this thesis we focus on message passing. Message passing sends messages to “actors” or “agents”, and assumes the infrastructure will allow the object to select and execute the appropriate code.

2.4 Actors

2.4.1 Actor Model

The actor model is one model of concurrent computation. An “actor” is a unit of concurrent computation, where in response to a message the “actor” can: send messages to other “actors”, create “actors”, make local decisions, and designate its behavior for the next message it receives.

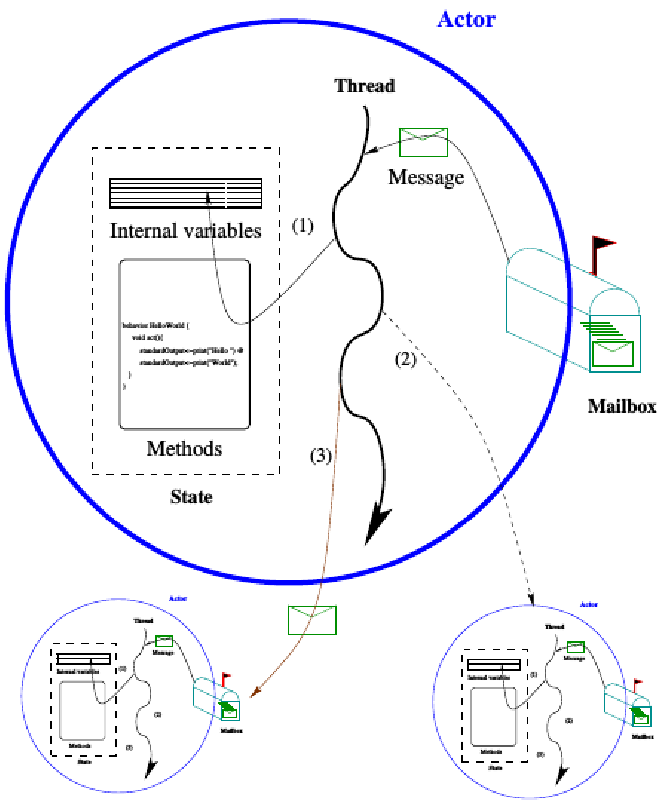

Figure 2.2: Actor Model Diagram [3]

Figure 2.2 [3] illustrates an actor is an encapsulated state with its own thread of control and mailbox. When in an encapsulated state there is no shared memory between actors, thus actors can migrate to different locations easily. Actors can communicate by sending asynchronous messages to one another's mailboxes. Actors can reside in different locations, and therefore in order for actors to communicate in a distributed system they need to possess the mailing address of the other actors.

The actor model is an ideal candidate for use in Mobile Cloud Computing. Actors can be created in various computational resources and executed concurrently. Actors ensure a uniformity across all resources as the same behavior can be assigned to multiple actors.

2.4.2 SALSA Language

SALSA (Simple Actor Language, System and Architecture) is a concurrent programming language that is based on the actor model. Employing Java as its base language, SALSA has adhered to similar concepts.

Table 2.4: Java vs SALSA Concepts

Java	**SALSA**
packages	modules
classes	behaviors
objects	actors
methods	messages

Similar to package structuring in Java, actors with similar behaviors are grouped together in modules. Behaviors represent the definition of actors, similar to classes representing the definition of objects. The way objects call methods in Java, actors send messages, either to itself or to other actors. For distributed systems SALSA actors are identified by a specific Universal Actor Name (UAN) and Universal Actor Location (UAL). When an actor receives messages there is no guarantee that it will process them in the order at which they were sent. The actor model naturally includes non-determinism, and SALSA provides solutions for this. Token-passing and join continuations allows messages to block until the token has been passed from one actor to another. When dealing with distributed applications the overhead of actor operations becomes a factor. This includes actor creation, message sending, and migration.

Table 2.5: SALSA Operation Overhead [3]

Operation	**Type**	**Time**
Actor Creation	Local actor creation	386 μs
Message Sending	Local message sending	148 μs
	LAN message sending	30 - 60 ms
	WAN message sending	2 - 3 sec
Actor Migration	LAN minimal actor migration	150 - 160 ms
	LAN 100 Kb actor migration	250 - 250 ms
	WAN minimal actor migration	3 - 7 sec
	WAN 100 Kb actor migration	25 – 30 sec

3. SALSA Language on Android Devices

3.1 SALSA in Mobile Cloud Computing

The benefits of using SALSA in Mobile Cloud Computing are: the number of devices SALSA can run on, the ability to dynamically reconfigure applications, and ease for developers.

3.1.1 Heterogeneity in Devices

The world is naturally heterogeneous, especially the technical world. There are a mixture of operating systems, hardware, and devices. Finding a way to perform computations on everything can be difficult. SALSA provides a reasonable solution to this problem. SALSA actors can be created or migrated to virtual machines, or theaters, on different computational resources. A theater runs on the Java Virtual Machine (JVM); this implies that SALSA can run anywhere Java can be run. This means SALSA actors can be on public and private clouds, desktops, laptops, and even tablets and smartphones. Whether it is Windows, Linux, or Mac OS X, Java can be run on virtually everything. Mobile devices are a little tougher; out of the top Mobile OS's, Android, Windows, and IOS, Android is the only one that can run Java without problems. Android [17] is a mobile operating system that runs Java-flavored code on the Dalvik Virtual Machine (DVM). The DVM is a register-based virtual machine that aims to run on low memory, which makes it ideal for mobile devices.

This chapter is to appear in: S. Imai, P. Patel and C. A. Varela, "Developing Elastic Software for the Cloud," in *Encyclopedia on Cloud Computing*, Hoboken, NJ, USA: Wiley, 2014.

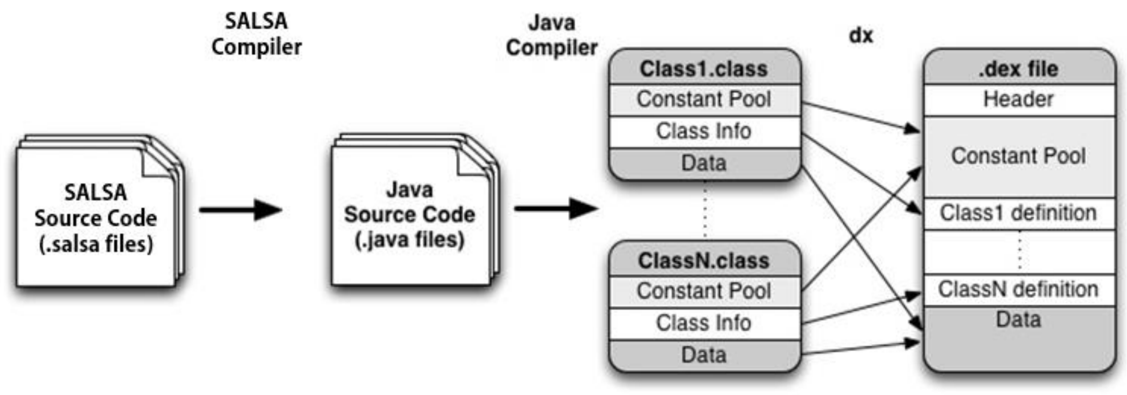

Figure 3.1: SALSA to Java to Dalvik [18]

Figure 3.1, edited from [18], represents the conversion of SALSA source code to Java source code. It then converts from Java source code to Dalvik byte code that can run on the DVM. Since SALSA actors run on the JVM, they can be made to run on Android devices. With Android dominating the mobile device market share at 84.7% [19], SALSA can run on the majority of computational devices. As technology is evolving, workarounds can be discovered to open SALSA to IOS and Windows Phones as well.

3.1.2 Dynamic Reconfigurability

Since SALSA actors have an encapsulated state, they can be migrated at ease without making adjustments to the overall application. A migrate message can be sent to an actor, and when the actor receives and processes the migrate message they will serialize their state, mailbox, and thread of control. When the actor migrates to the new UAL, the actor will de-serialize its state and mailbox, restart its thread, and begin to process messages again. Applications can take advantage of this by load-balancing theaters that are not being fully utilized. For example, Cloud Operating System (COS) [20] is a middleware for SALSA applications that is able to perform autonomous scalability based on the current workload of theaters.

3.1.3 Ease for Developer

When compared to other concurrent options, SALSA code can be written very optimally in terms of lines of code.

Table 3.1: Lines of Code Comparison [3]

Application	**Foundry**	**SALSA**	**Java**
Shared Code	40	10	34
Basic Multicast	146	27	115
Acknowledged Multicast	60	21	134
Group-knowledge Multicast	73	24	183
TOTAL	319	82	466

Table 3.1 [3] represents the comparison of lines of code for four different applications. As shown above, SALSA is able to write the same application in the smallest amount of lines. SALSA applications in turn will be more compact and therefore can be easier to comprehend.

3.2 Example Application

3.2.1 Distributed Face Recognition

Face recognition is an application that can be vastly improved by leveraging cloud computing resources. Rather than using a single device, such as a smartphone in this case, we can offload parts of the image processing to the cloud [2]. By using cloud computing, we can save battery in the mobile device, and we can also consider larger data sets. Using the SALSA programming language and the `FaceRecognizer` API from *OpenCV* [21], we can design a mobile phone application to recognize a face in a given image using a database of faces (see Figure 3.2).

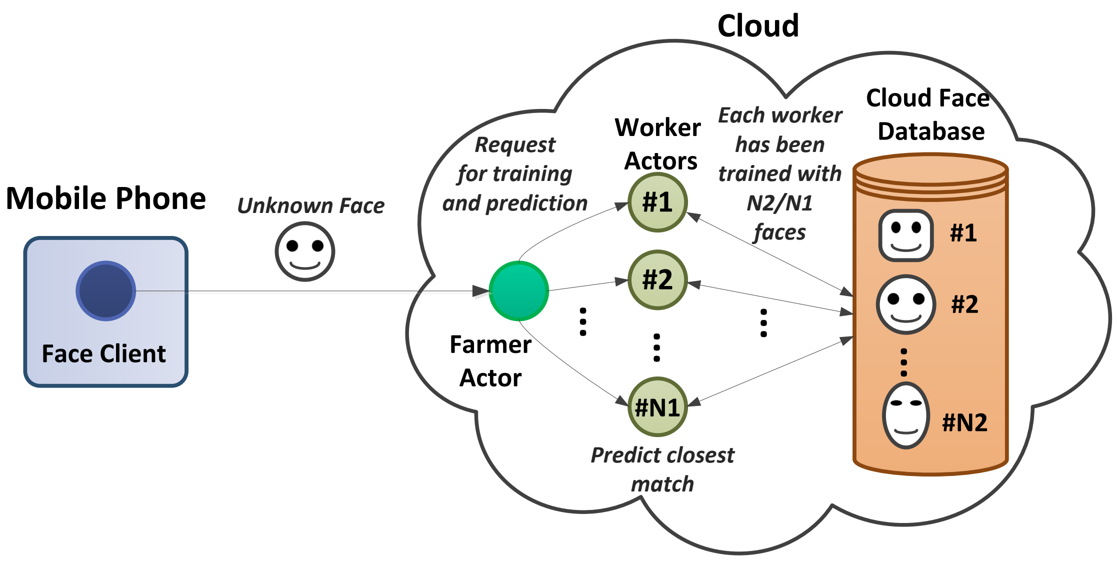

Figure 3.2: Distributed Face Recognition using SALSA Actors

The face recognition application consists of two stages, the training stage, and the prediction stage. The training stage trains a database of faces using the `FaceRecognizer` method defined in OpenCV. The prediction stage predicts a given face using the desired method with a certain confidence, 0 being the exact face, and infinity being completely unlike. When the database of faces is small, there is no need to offload computation because the phone can process the faces locally just as fast as it would take to offload to the cloud and process remotely. As the database grows, we run into the limitations mentioned above and offloading to the cloud can be beneficial.

The distributed face recognition model consists of a "farmer actor" that creates *N1* "worker actors" in the cloud. While the farmer and worker actors reside in the cloud, a client on the mobile phone requests the farmer actor to recognize an unknown face. The farmer actor assigns each worker actor a range of faces (*N2/N1* each) to train from the cloud face database containing *N2* faces. The worker actors then predict the closest match to the unknown face based on their assigned database. The farmer actor collects the closest match from each worker actor and calculates the best candidate for the unknown face. Pseudo code for the client, farmer, and worker SALSA programs are shown in Figure 3.3, Figure 3.4, and Figure 3.5 respectively.

```
behavior FaceClient {
    void act(String[] args) {
        FaceFarmer farmer = (FaceFarmer)
            FaceFarmer.getReferenceByName(
                "uan://nameserver/facefarmer");
        Image unknownImage = new Image(args[0]);
        farmer<-predictAll(unknownImage)@
            displayImage(token);
    }
}
```

Figure 3.3: SALSA Pseudo Code for Face Recognition Client Actor

```
behavior FaceWorker {
    FaceRecognizer faceRecognizer;
    void train(Image[] assignedDatabase) {
        faceRecognizer.train(assignedDatabase);
    }

    ImageMetaData predict(Image testImage) {
        // return the closest match
        return faceRecognizer.predict(testImage);
    }
}
```

Figure 3.4: SALSA Pseudo Code for Face Recognition Worker Actor

```
behavior FaceFarmer implements ActorService {
    //CloudFaceDatabase contains N2 faces.
    void act(String[] args) {
        faceWorker[] workers = new faceWorker[N1];
        for (i = 0; i < N1; i++) {
            workers[i] = new faceWorker(new UAN(…));
            workers[i]<-migrate(new UAL(…))@
            workers[i]<-train(
                    N2/N1 faces from CloudFaceDatabase);

        }
    }
    Image predictAll(Image testImage)
        join {
            for (i = 0; i < N1; i++)
                workers[i]<-predict(testImage);
        }@getBestMatch(token)@currentContinuation;
    }
    Image getBestMatch(ImageMetaData[] closestMatches) {
        // Find best match with the highest confidence.
    }
}
```

Figure 3.5: SALSA Pseudo Code for Face Recognition Farmer Actor

In Figure 3.5, note that a `join` block in the `predictAll` message handler is used to synchronize all the worker actors executing `predict`. After all the workers finish processing `predict`, the join block returns an object array `ImageMetaData[]`, which contains prediction confidence from the workers. Finally, `getBestMatch` takes the array and finds the best matching image with the highest confidence. In Chapter 5, we look at the results of the face recognition application.

4. MobileCCA

4.1 Approach

We describe Mobile Cloud Computing using Actors (MobileCCA) as a cloud-based mobile augmentation strategy that uses the actor model for the offloading of computational tasks. Specifically, we use SALSA as the actor programming language for the distribution of actors to different computational resources.

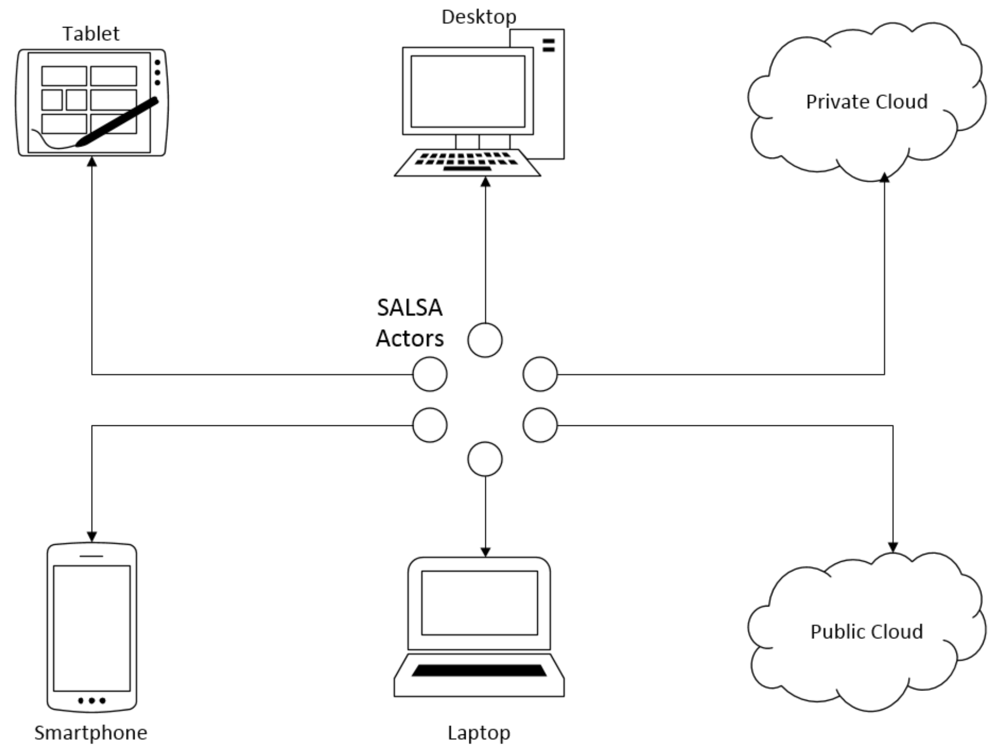

Figure 4.1: SALSA Actors Distribution

We introduce a SALSA add-on for the Android OS that will run SALSA code on Android devices. Application developers can now create SALSA applications that will be able to migrate to all SALSA compatible devices. Although there are inherit limitations from Mobile Cloud Computing, through the use of actors, these limitations can be mitigated. Battery issues can be resolved by controlling the movement of actors to the mobile device. Storage capacity and processing power can be increased by utilizing actors in the private and public cloud. Security vulnerabilities can be decreased by controlling the location of actors. Finally, the cost of using the public cloud can be controlled by migrating actors to the cloud only if there is an allotted budget. Policies can be implemented to control the distribution of actors and thus limit the computation on select devices and change the quality of service.

4.2 Quality of Service Policies

This section looks at several possible policies that can be used with MobileCCA. For example, these policies can include: battery-minded, security-minded, deadline-minded, budget-minded, history-driven-minded, or a combination of a few.

4.2.1 Battery-Minded

The most important factor for Mobile Cloud Computing is reducing the consumption of battery. MobileCCA's approach to this is the migration of actors away from the mobile device. Based on the current battery level of the device, actors can make decisions where to perform the computation. For example as battery level deteriorate, the actors processing on the mobile device may begin to migrate away towards other resources. A simple Battery-Minded policy can control the distribution of mobile actors with the following equation:

$$A_{Mobile} = \frac{B}{R * A_{Total}} \tag{4.1}$$

This controls the number of actors on the mobile device by its current battery status B. For R number of computational resources, we can divide the rest of the remaining actors equally. Figure 4.2 shows an experiment with a mobile device and private cloud as the battery deteriorates.

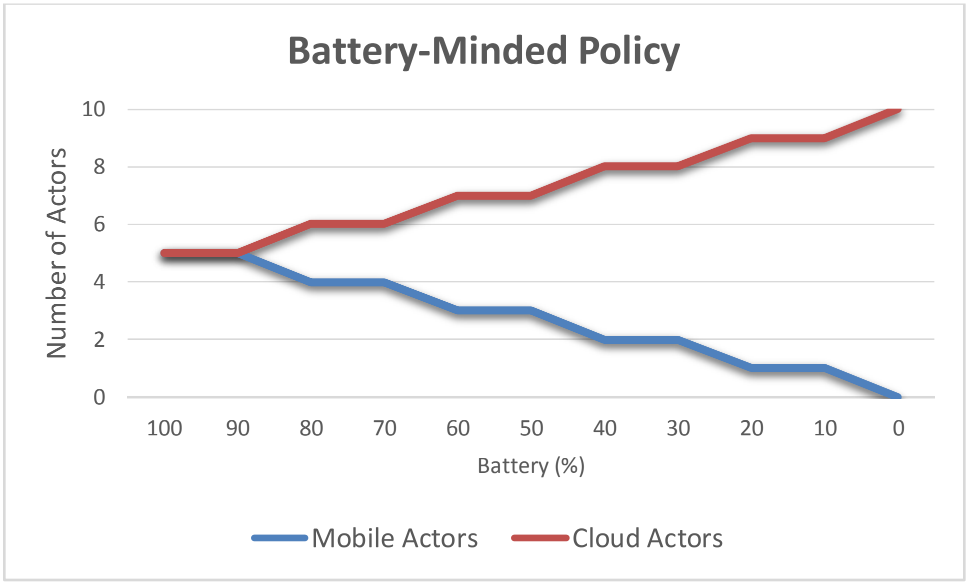

Figure 4.2: Battery-Minded Policy Experiment

4.2.2 Security-Minded

When data is moved to the cloud there is always the possibility of vulnerabilities. The ability to control the location of the actors means that the user can control where their data will go. Users confident with public cloud providers can allow the migration of actors to public resources, or restrict actors to stay within private boundaries.

4.2.3 Deadline-Minded

Applications that have an inherent state of progress can update the user with their estimated time of completion. Therefore if an application is progressing faster than estimated it may be beneficial to scale down the computation to save money and energy. On the other hand, if an application is slower than expected, the computation can scale up and out towards public resources where the computation can complete faster. A simulation of this policy consists of initially designating a number of actors for each computational resource. In Figure 4.3, we designate five actors each to the mobile device and public cloud. As the application progresses the actual completion percentage becomes lower than the estimated percentage by a certain threshold. In this case we migrate some of the mobile actors to the cloud. Eventually when the actual completion percentage goes ahead of the estimated percentage by a certain threshold, we can migrate the public cloud actors back to the mobile device to save cost.

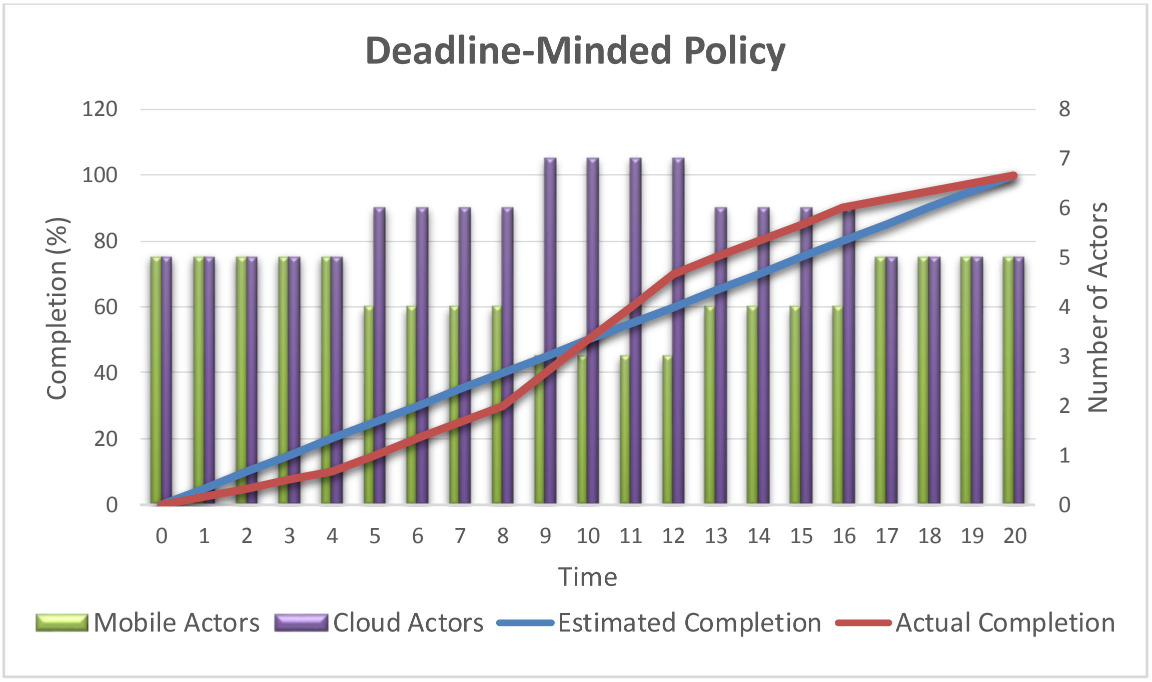

Figure 4.3: Deadline-Minded Policy Simulation

4.2.4 Budget-Minded

Using the public cloud means that the user will incur a cost for the rented resources. Budgeted users can indicate a threshold of money that they would be willing to spend for an application. A user would ideally like to spend as little as possible to complete an application, but the higher the budget ensures more leeway for the application to complete in a faster time. Resource prediction algorithms can be used to estimate the progress of an application based on a subsection of work completed for a given monetary value. WECU [22] (Workload-Tailored Elastic Compute Units) is an extension of Amazon's Elastic Compute Units, and is a cost-optimal resource prediction method. By using WECU, we can provide a budget and deadline for the application. Based on the number of tasks in a given application, WECU can provide an accurate resource configuration to complete the application within the restrictions. An example of a Budget-Minded policy can be as simple as ensuring the CPU utilization for every actor in the public cloud is as high as possible. This allows the user to obtain the best "bang for their buck". Since a user is paying for resources, using that resource to its full extent is important. For example, an Amazon EC2 C3 Large instance can cost ~$0.10 per hour. The number of actors located on a computational resource can correlate to the amount of CPU each individual actor gets. Since each actor is competing for resources, sending a large number of actors to the public cloud might not be the smartest configuration. Context-switching and CPU contention can cause this configuration to provide a lower individual CPU utilization per actor, even though the total CPU utilization for the machine may be 100%.

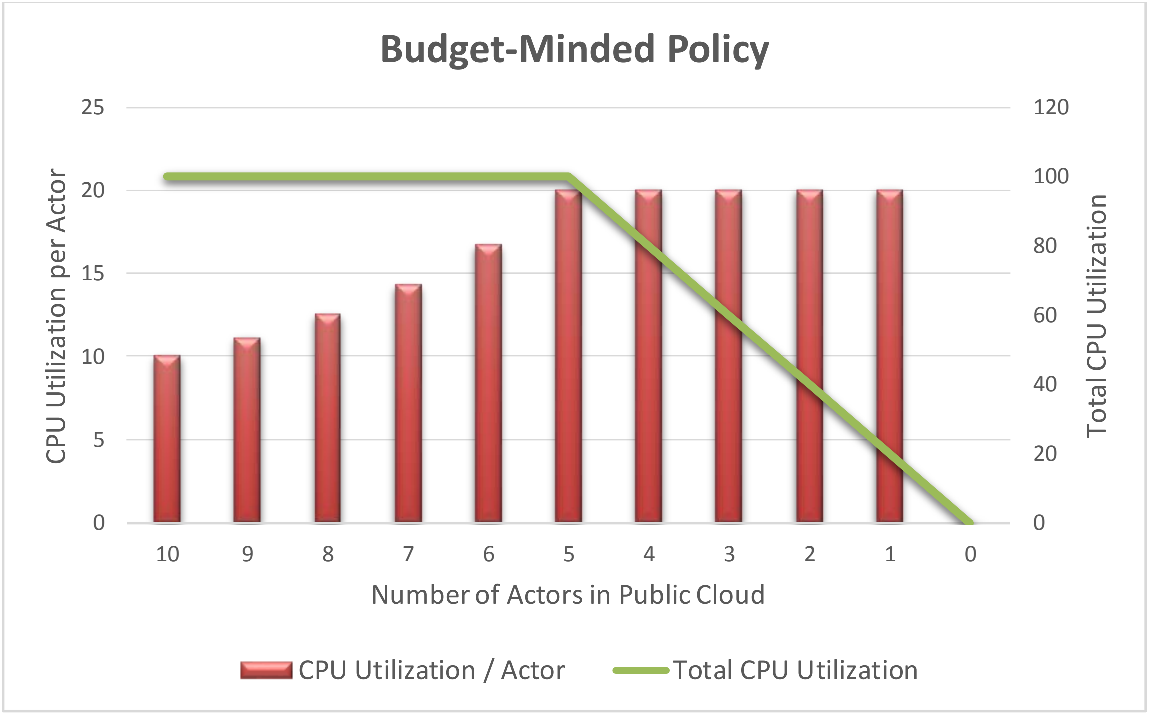

Figure 4.4: Budget-Minded Policy Simulation

Figure 4.4 represents a simulation of a hypothetical public cloud actor configuration for an application. We can see that initially having 10 actors on the public cloud provided a lower individual CPU utilization than that of 5 actors. Having a policy in place to choose the correct number of actors to place in the public cloud can allow for an optimal use of the user's budget.

4.2.5 History-Driven-Minded

We look at a group of applications that depend on certain data sets for computation. For example, in the Face Recognition application, recognition occurs based on the training phase over a database of faces. The type of images in these databases can lead to a different confidence result. A database of images in the mobile device can consist of friends and family, while the database in the private and public clouds can contain a broader set of images like celebrities. Testing an unknown image in all three computational resources can provide different results. The overall goal of this application is to find the highest accuracy label to the unknown image. With a History-Driven policy, actors can initially be distributed equally to all resources to obtain an overall sense of the databases. If the unknown image is a family member, it would make sense for the mobile device to provide the lowest uncertainty result, and the private and public cloud to return lower quality results. Therefore, we can migrate actors away from these lower quality resources towards the resource that provided the lower uncertainty result in the previous iteration of the application. Figure 4.5 represents an experiment using this policy. Through five iterations of the application, prior history influences the locations of actors in the next iteration. Initially, five actors are located in the mobile device, private cloud, and public cloud. In iteration one, we can see that the mobile device provided the lowest uncertainty. Therefore, in the next iteration, actors in computational resources that are a threshold away from the best confidence measure are moved to the lowest uncertainty computational resource.

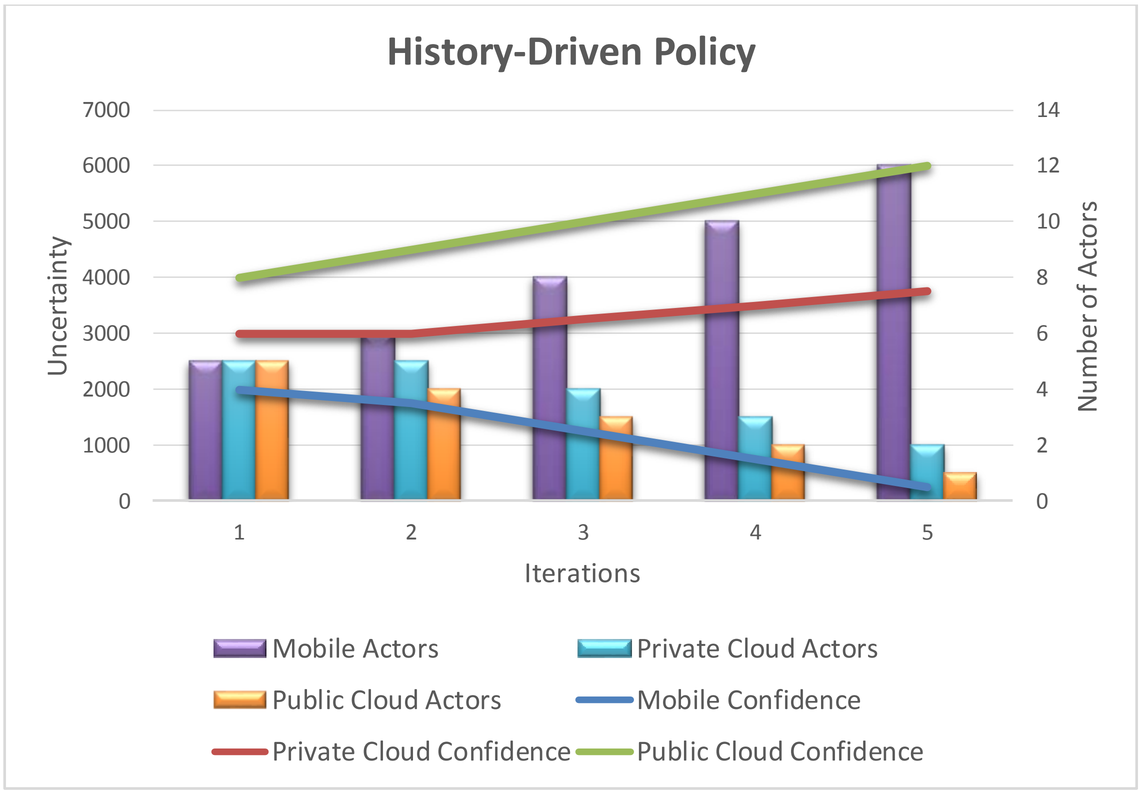

Figure 4.5: History-Driven-Minded Policy Experiment

5. Face Recognition Application and Experimental Results

5.1 Face Recognition

In this chapter we take the Face Recognition application and examine the advantages of using the MobileCCA approach. The Face Recognition application consists of a training phase that uses the Eigenface approach [23] to determine the uncertainty, or confidence, of an unknown image. Zero represents an exact image in the training database up to infinity for being completely different. The application trains over a provided set of faces in its personal database and attempts to recognize the unknown face using that database.

5.2 Experimental Results

Table 5.1: Computational Resource for Face Recognition Application

Computational Resource	CPU (GHz)	# Cores	Memory (GB)	Storage (GB)
Samsung Galaxy S4	1.9	4	2	16
Private Cloud (Shared)	4	8	16	256
Public Cloud Amazon EC2 C3 Large (Dedicated)	2.8	2/3.5	3.75	32

Using the resources in Table 5.1, we first get a baseline for the Face Recognition application by writing the application in Java and running it separately on the mobile device and private cloud. Then we use the SALSA version using 1 Actor to get a comparison. For the actual face images we will be using the Georgia Tech face database [24] with 10 people. We perform experiments with 100 faces (10 faces per person), 50 faces, (5 faces per person), and 20 faces (2 faces per person). Each image on average is ~50 kB. For each experiment we select an image from each of the 10 people and perform the face recognition application. If the correct person is identified we mark the charts with the uncertainty as a bar, if the application provides an incorrect result we do not include an uncertainty bar for that person label. The time in milliseconds is noted as a line for each person trial regardless of the recognition correctness.

5.2.1 Results on Mobile Device and Private Cloud using Java & SALSA 1 Actor

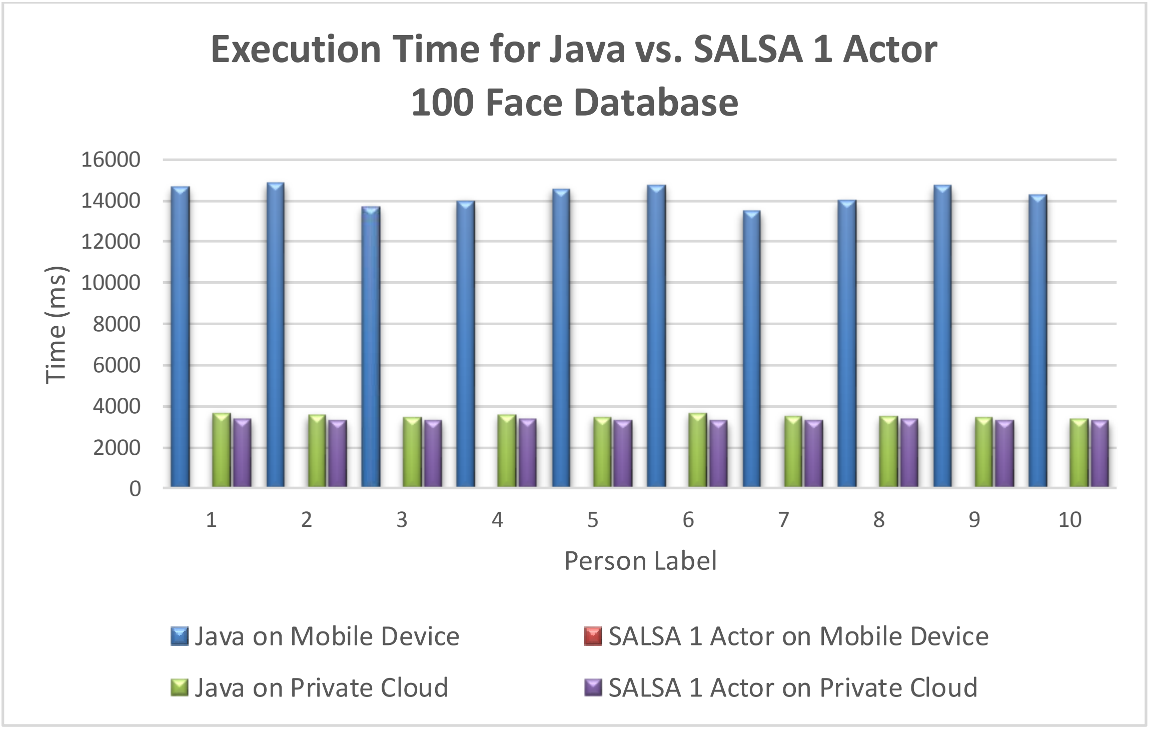

Figure 5.1: Execution Time for Java vs. SALSA 1 Actor with a 100 Face Database

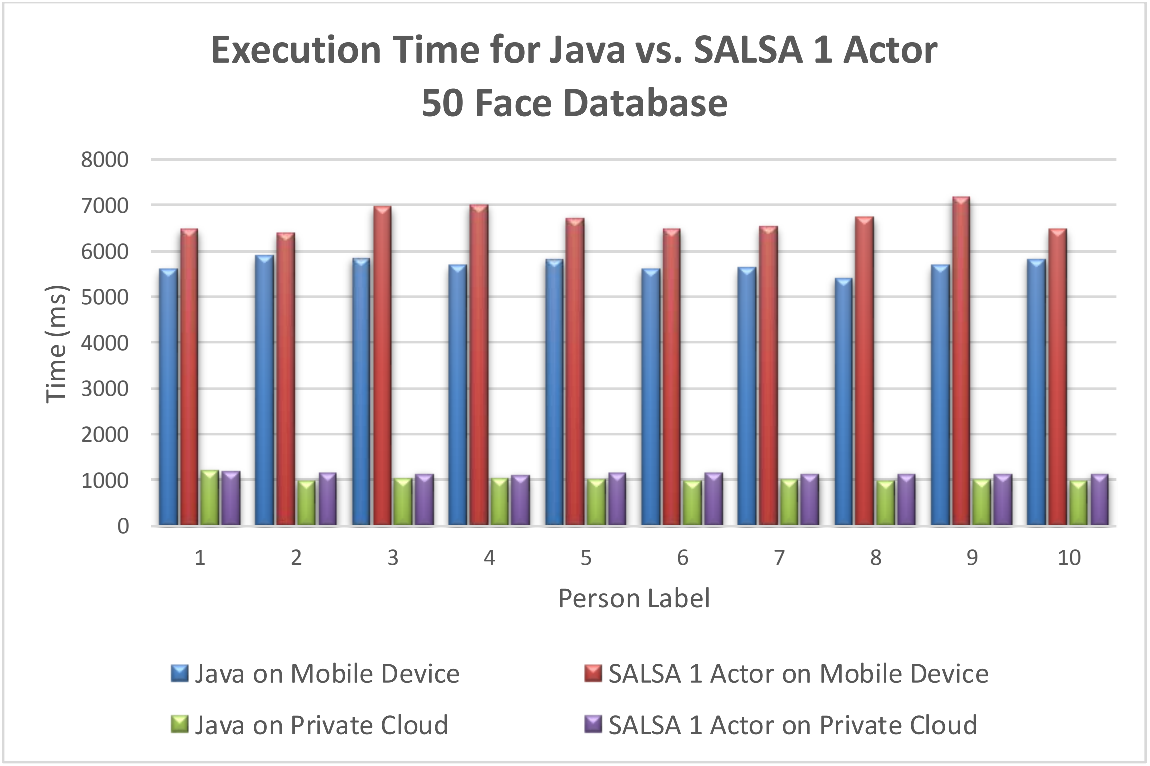

Figure 5.2: Execution Time for Java vs. SALSA 1 Actor with a 50 Face Database

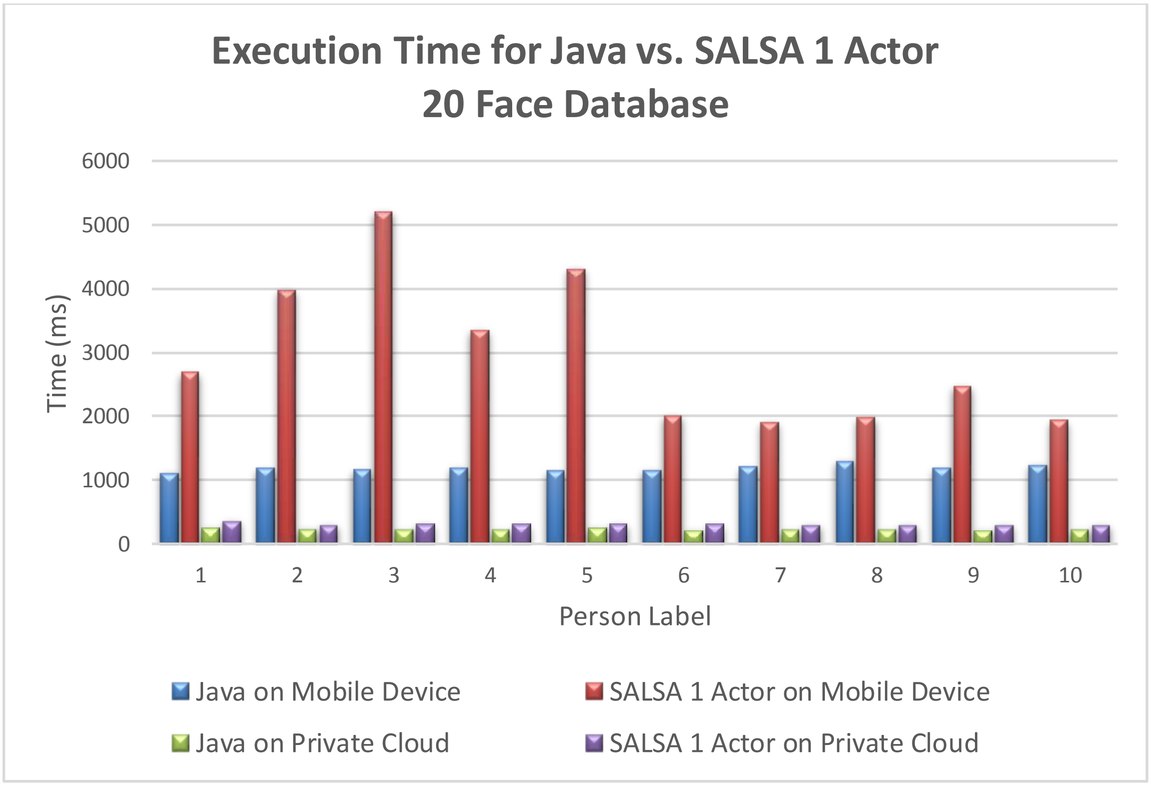

Figure 5.3: Execution Time for Java vs. SALSA 1 Actor with a 20 Face Database

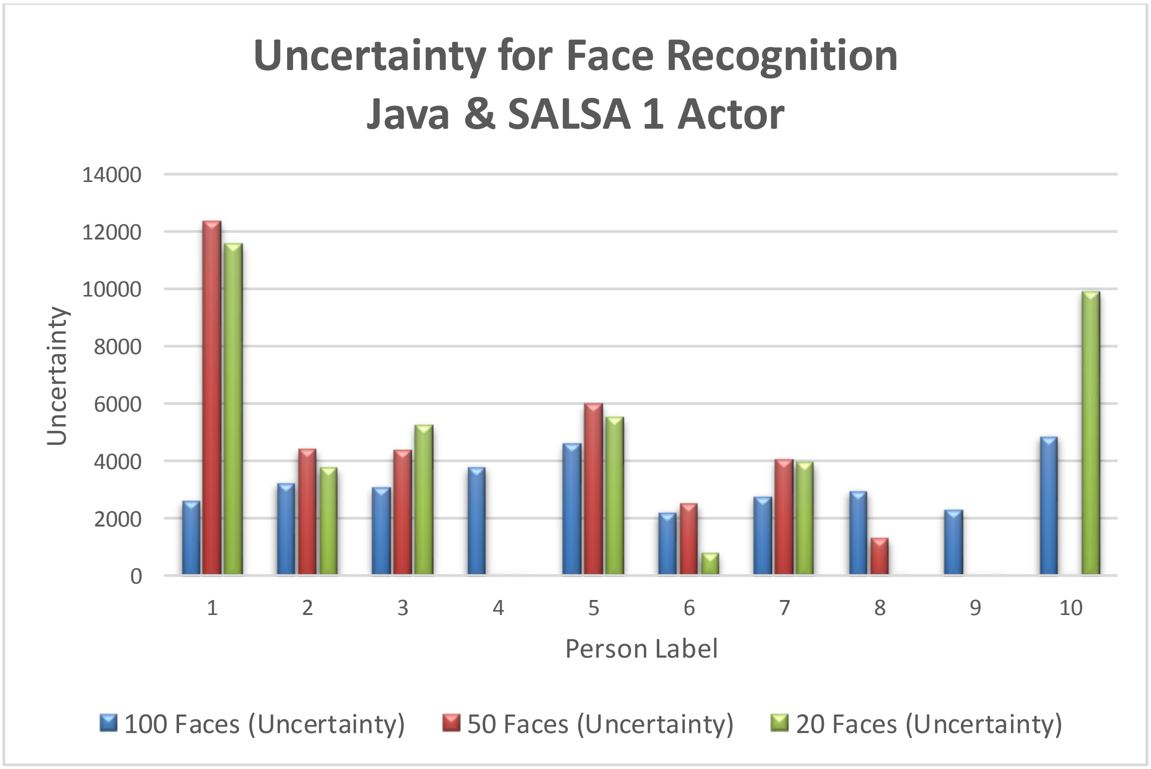

Figure 5.4: Uncertainty for the Java vs. SALSA 1 Actor Experiments

In this section we perform the face recognition application on both the mobile device and private cloud. We run the experiments using the Java and SALSA version of face recognition. In these experiments we compare the results of using a sequential Java version vs. SALSA with 1 Actor. From Figure 5.1, Figure 5.2, Figure 5.3, and Figure 5.4, we can make some deductions from the application. We see that as the number of faces in the database decreases the execution time decreases and the uncertainty on average increases. As we move computation to the private cloud we can also see a significant reduction in execution time. We can see that for a 100 face database on the mobile device we were not able to compute any of the unknown images. This could be due to the fact that the 1 actor was carrying too much data (100 references to faces) for the SALSA theater to handle on the mobile device.

The benefits of using SALSA is that with the altering of a single variable (number of actors), the application can run concurrently. This proves to be much easier than writing a concurrent version of Face Recognition in Java. In Figure 5.5 - Figure 5.12, we look at the results from using 5 and 10 actors on both the mobile device and private cloud. This entails that each actor will receive a portion of the face database to train over and return a result based on that subsection of data. When all actors have finished computation, the "Farmer" actor will decide which actor provided the lowest uncertainty result. It is important to see that we are now running completely different applications than before. Since each actor is receiving a subsection of the face database the computations occurring is different. We therefore see changes in the uncertainty of a predicated image.

5.2.2 Results on Mobile Device and Private Cloud using SALSA 5 Actors

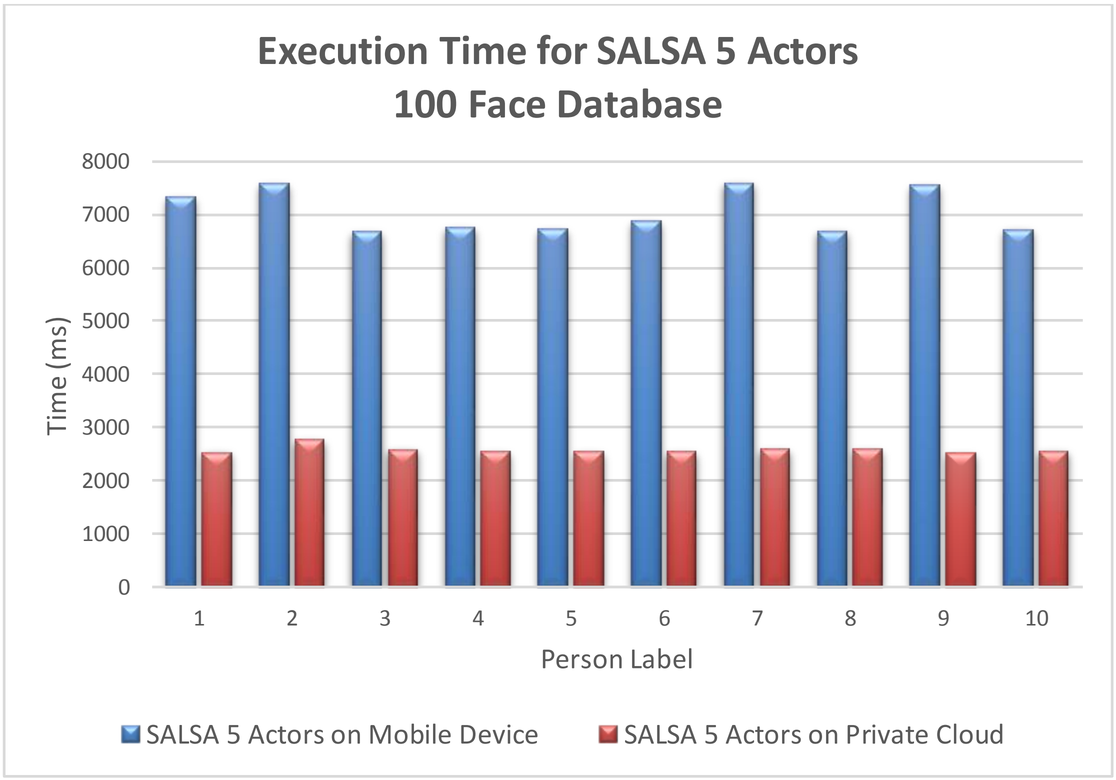

Figure 5.5: Execution Time for SALSA 5 Actors with a 100 Face Database

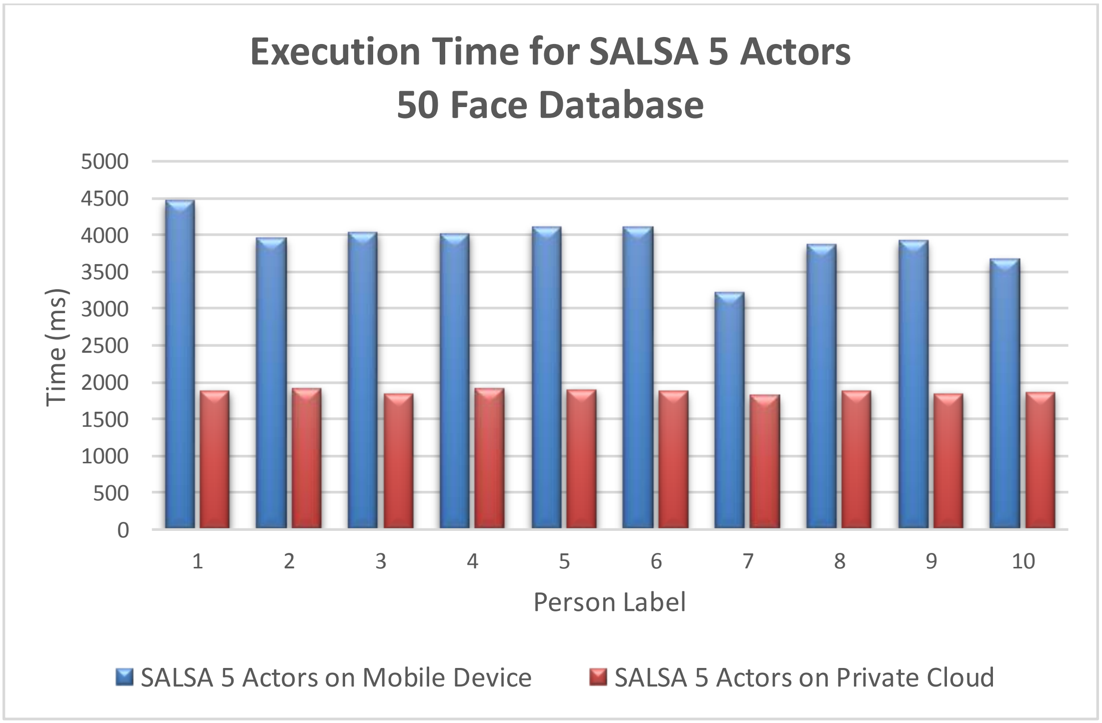

Figure 5.6: Execution Time for SALSA 5 Actors with a 50 Face Database

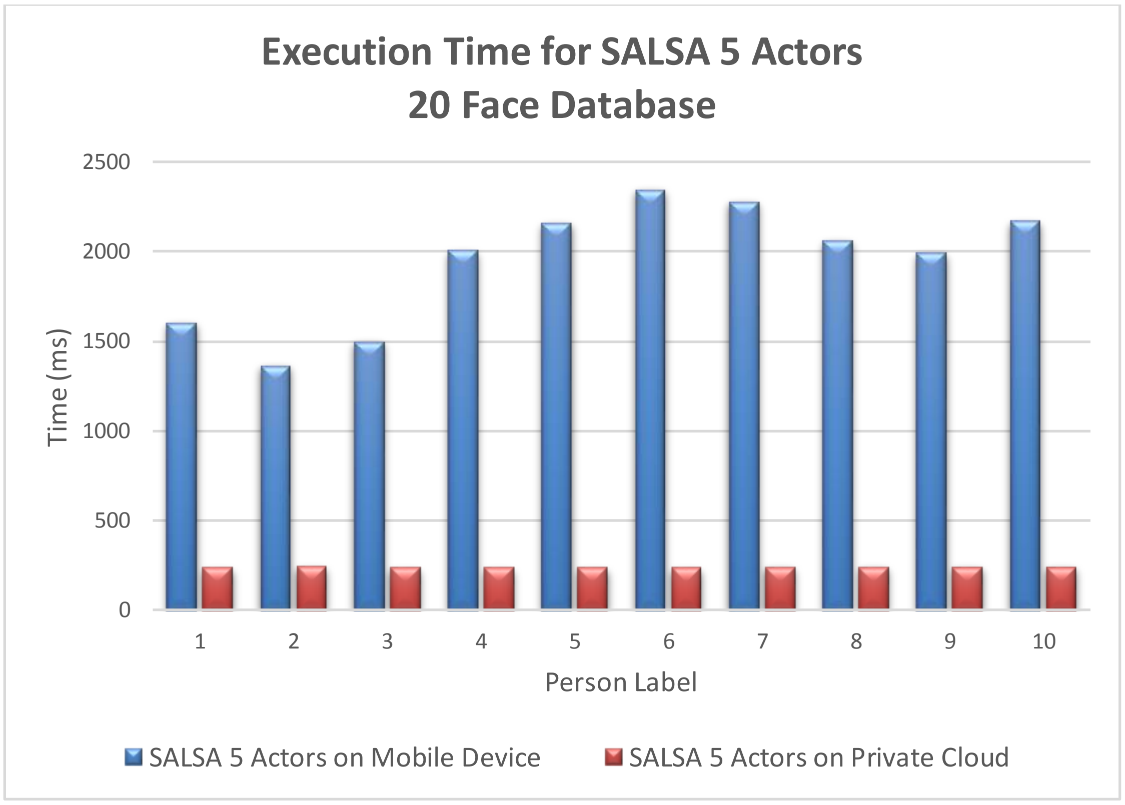

Figure 5.7: Execution Time for SALSA 5 Actors with a 20 Face Database

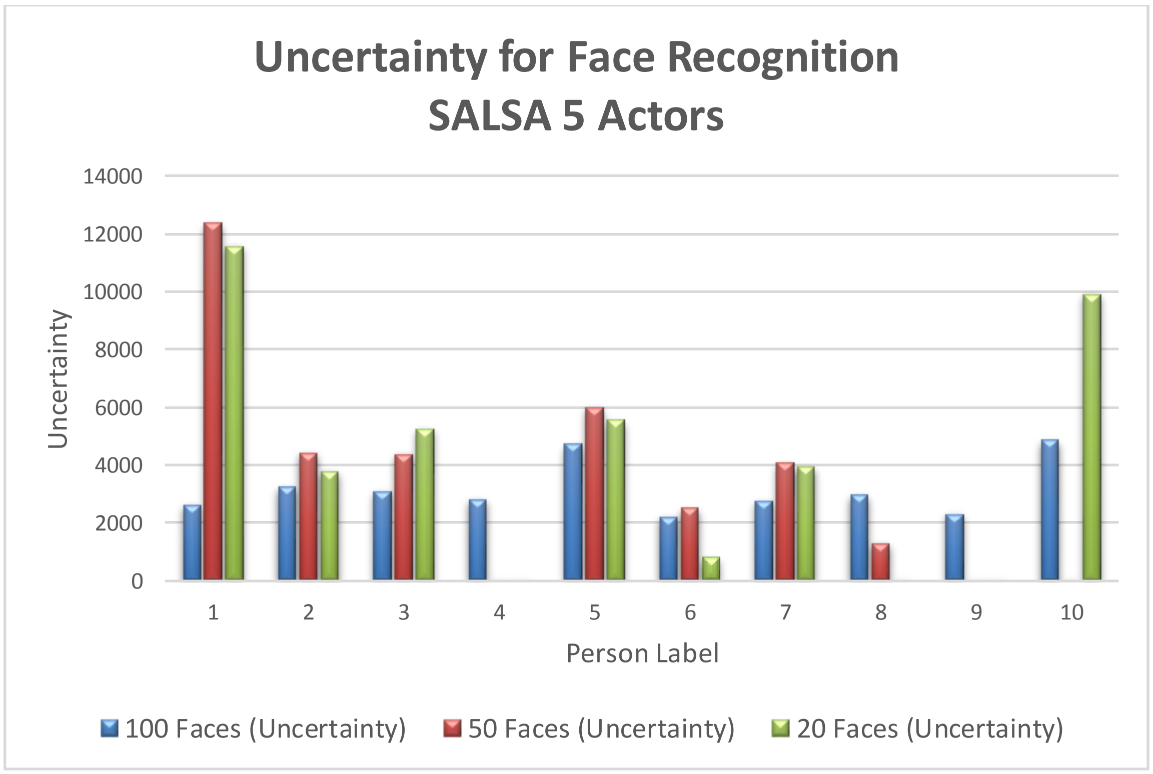

Figure 5.8: Uncertainty for the SALSA 5 Actors Experiments

5.2.3 Results on Mobile Device and Private Cloud using SALSA 10 Actors

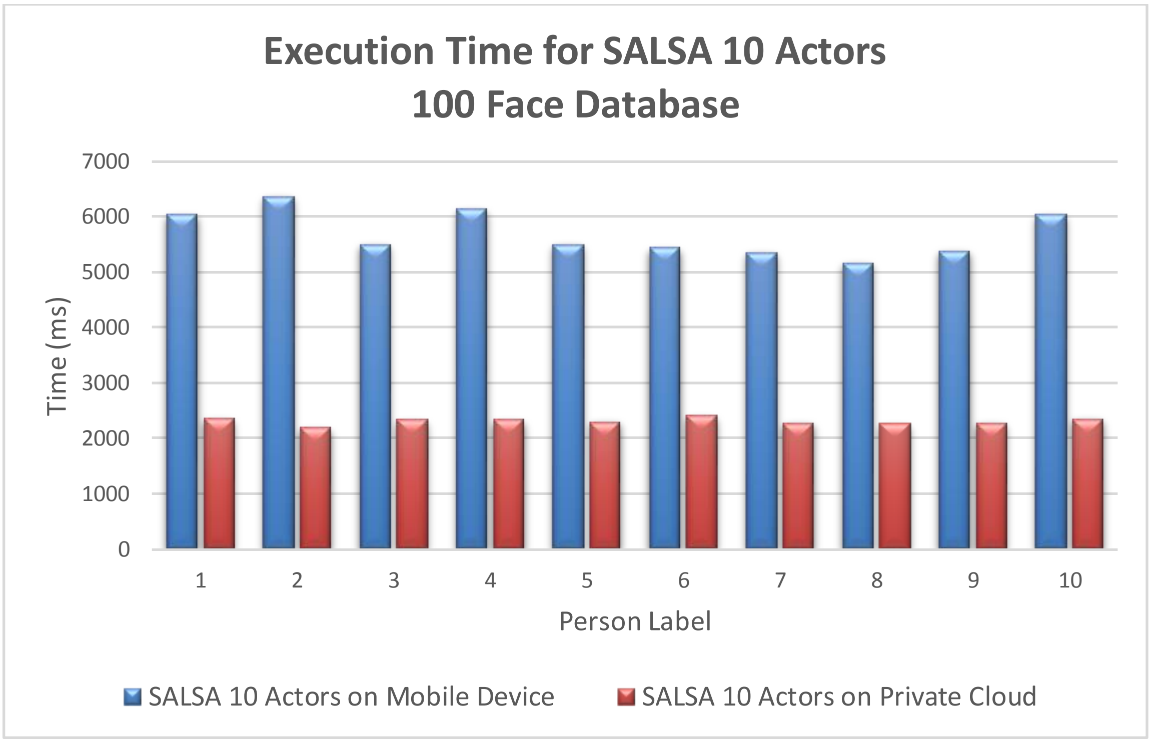

Figure 5.9: Execution Time for SALSA 10 Actors with a 100 Face Database

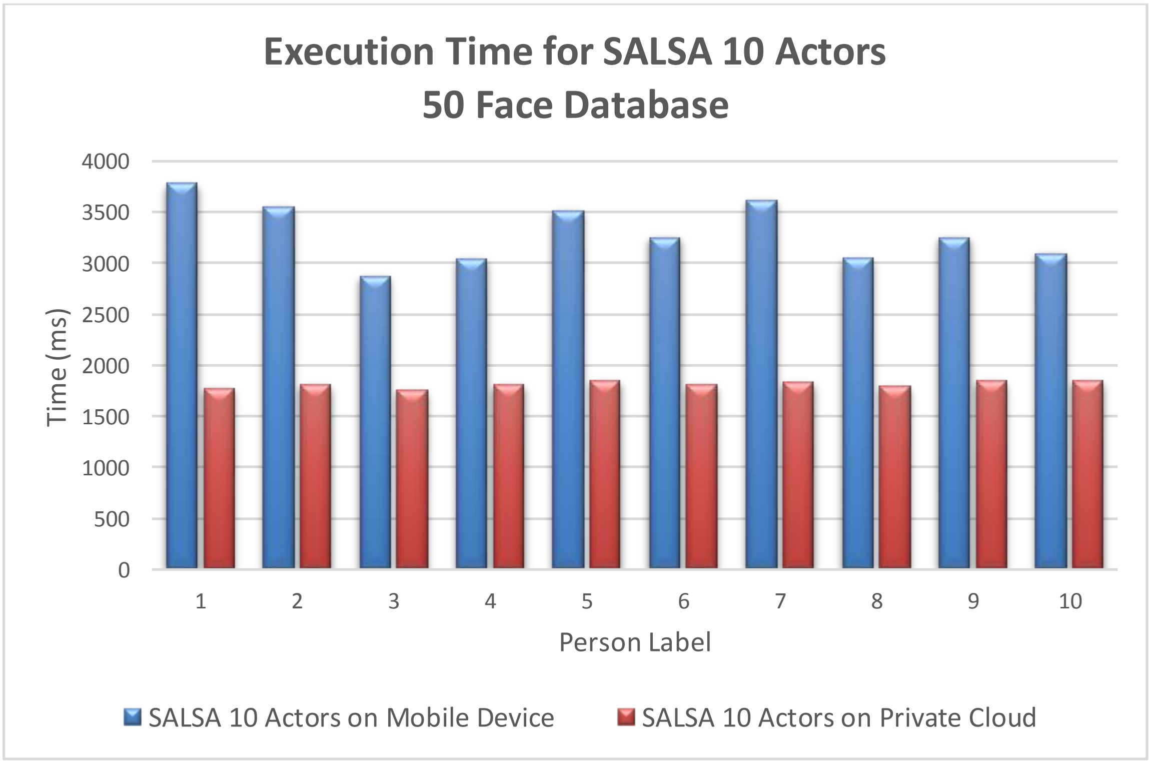

Figure 5.10: Execution Time for SALSA 10 Actors with a 50 Face Database

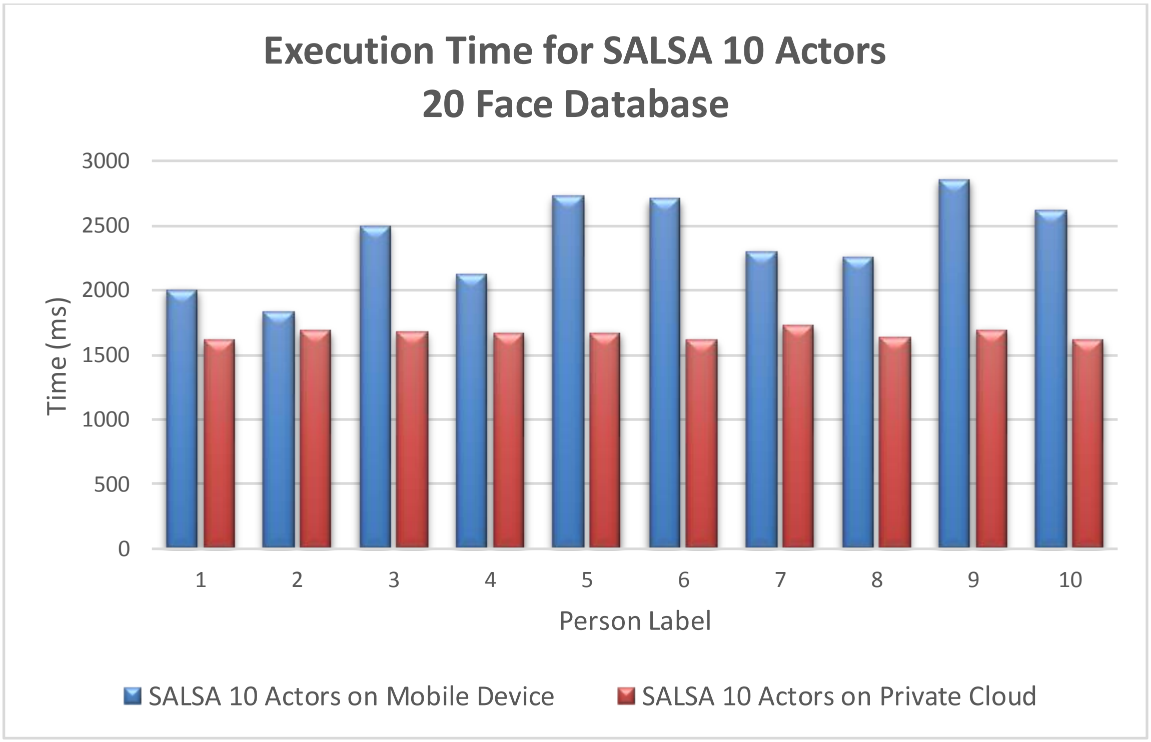

Figure 5.11: Execution Time for SALSA 10 Actors with a 20 Face Database

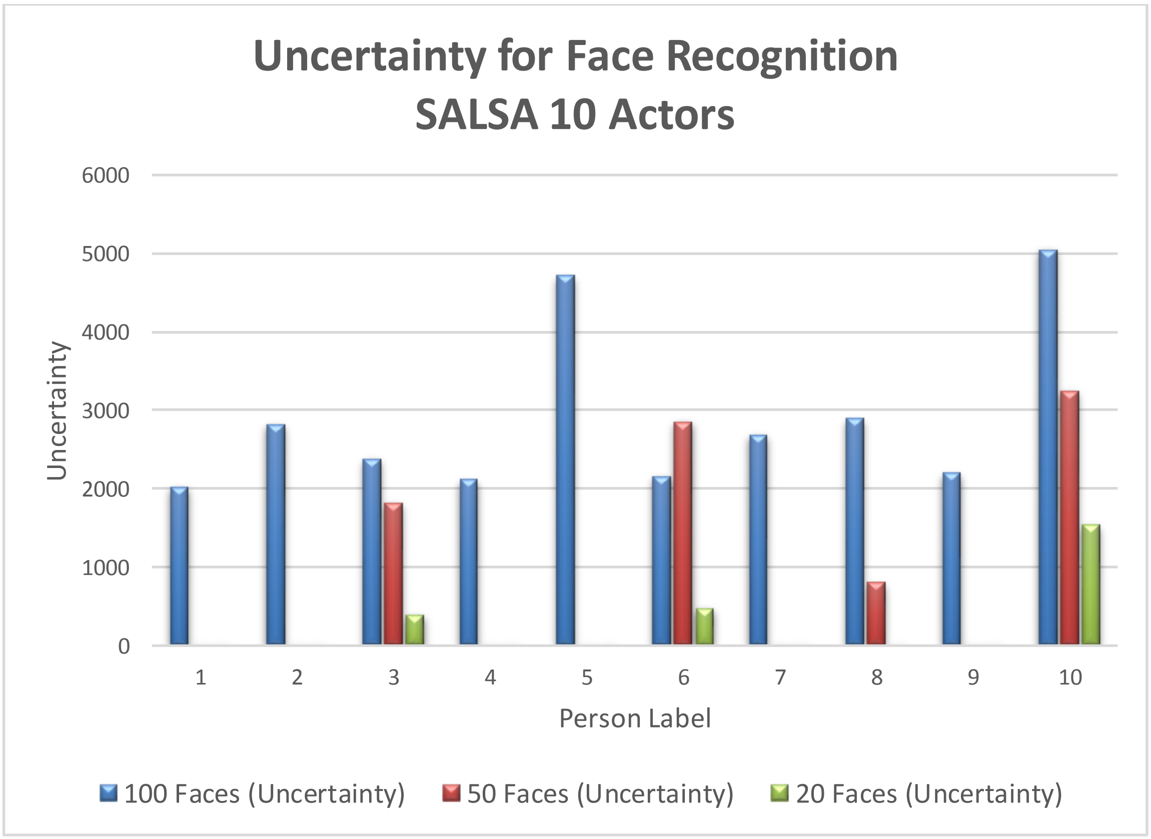

Figure 5.12: Uncertainty for the SALSA 10 Actors Experiments

5.2.4 Averaged Results on Mobile Device and Private Cloud

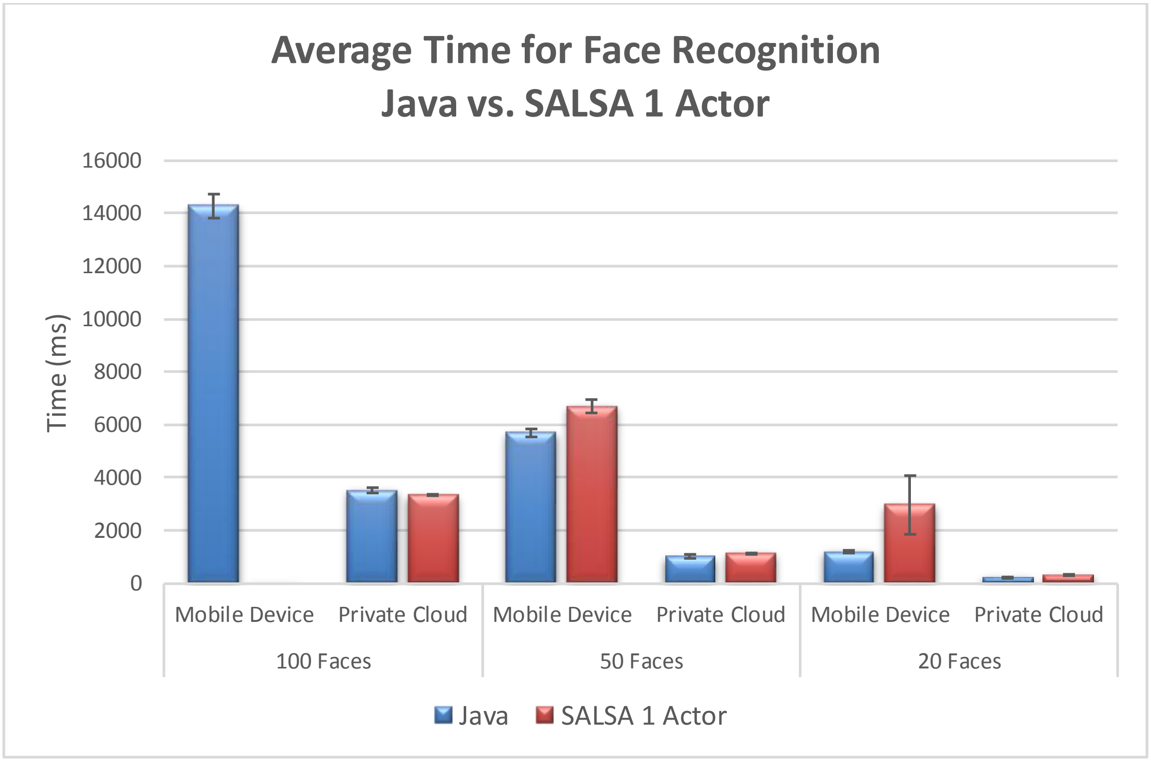

Figure 5.13: Average Time for Java vs. SALSA 1 Actor for 100, 50, and 20 Faces

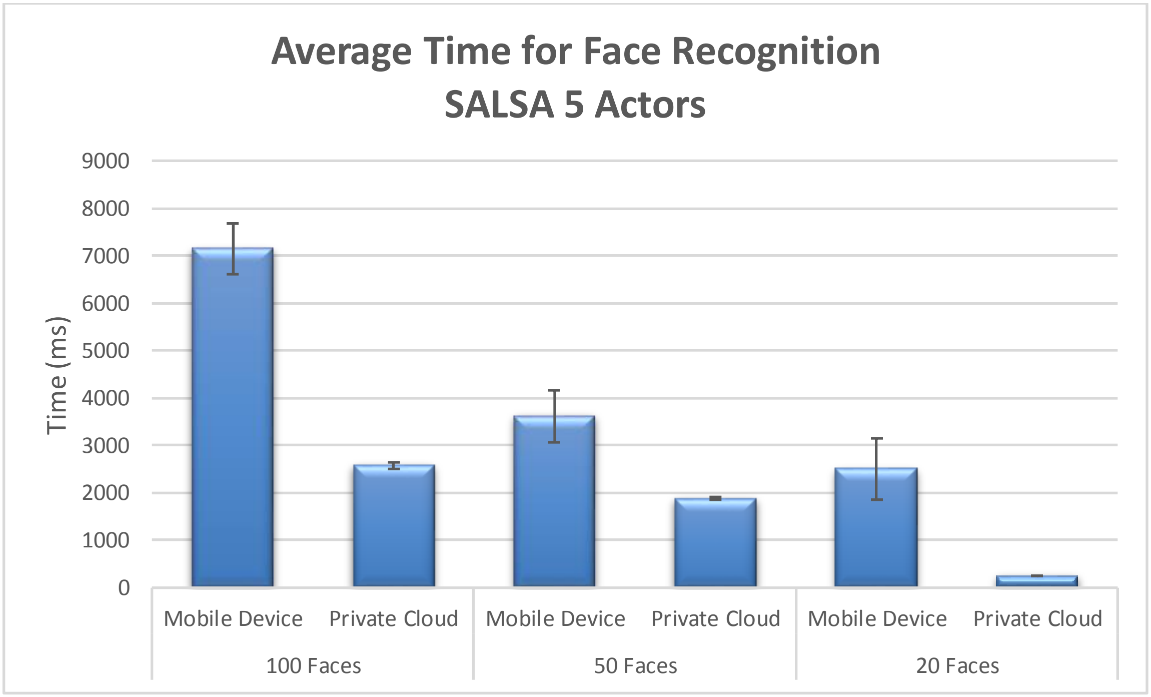

Figure 5.14: Average Time for SALSA 5 Actors for 100, 50, and 20 Faces

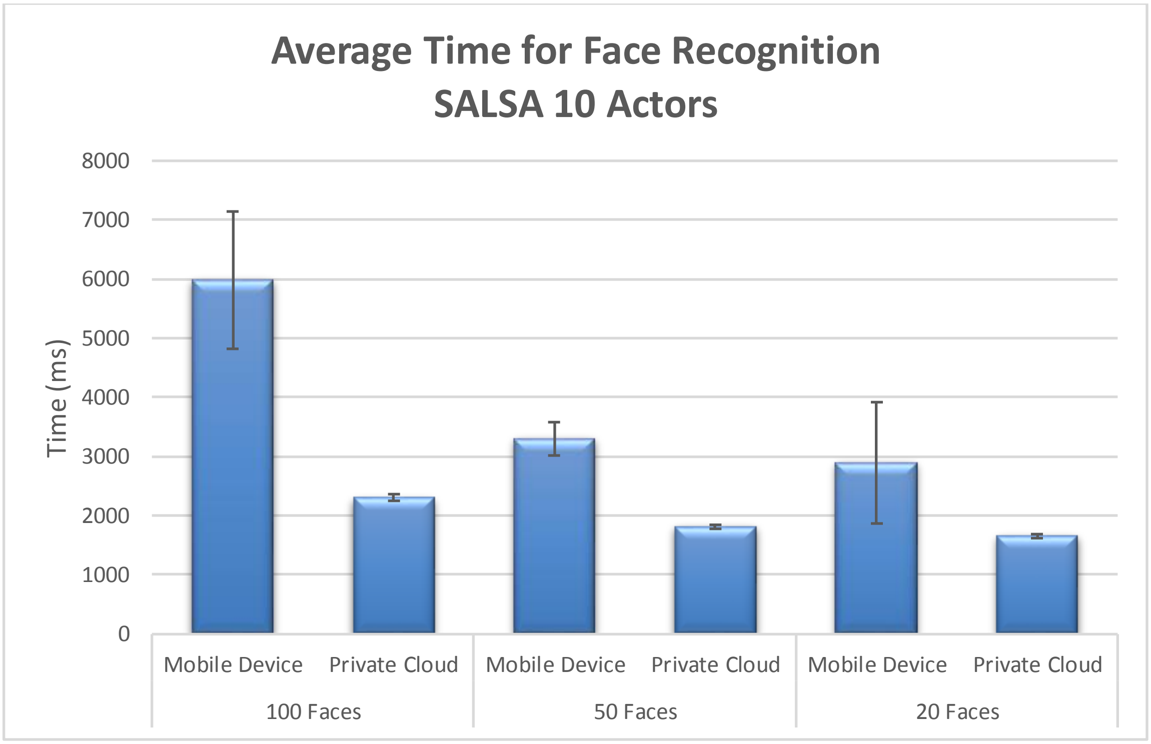

Figure 5.15: Average Time for SALSA 10 Actors for 100, 50, and 20 Faces

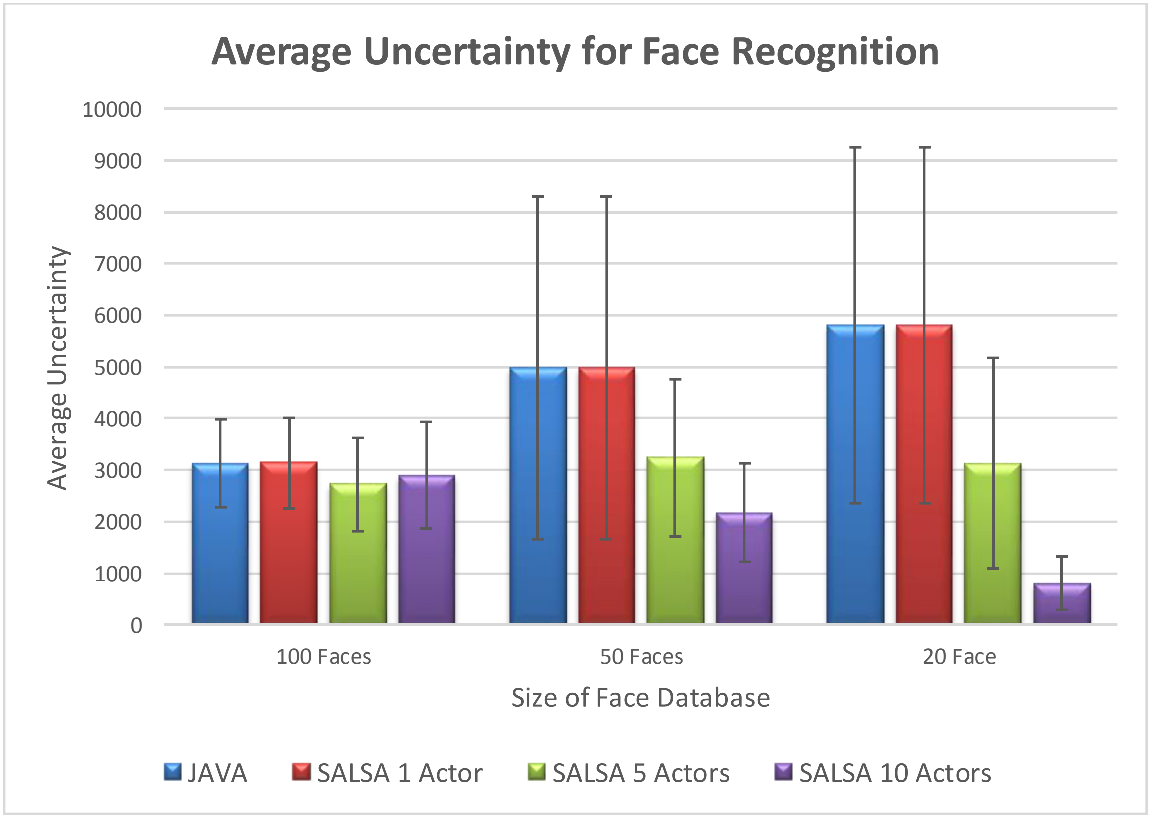

Figure 5.16: Average Uncertainty for 100, 50, and 20 Faces

In Figure 5.13, Figure 5.14, Figure 5.15, and Figure 5.16, we can see that as we increase the number of actors, the execution time reduces. We do, however, incur more false predictions, especially as the number of faces decreases. This could be because each actor has a smaller database of faces to train over and therefore the predications become less accurate. The more faces in the database provides a smaller standard deviation for uncertainty and therefore leads to more accurate results. For databases of size 50 and 20, as the actors increase although we may incur more false predictions the predictions that are correct have an extremely low uncertainty.

5.2.5 Averaged Results on Private Cloud and Public Cloud

For this experiment, we look at the Face Recognition application on the public cloud. Using an Amazon EC2 C3 Large instance, we perform the Java vs. SALSA experiments on the cloud. We can see that using the same face database (Georgia Tech face database with 100, 50, and 20 faces [24]), would provide results similar to that of the private clouds. Instead we can utilize the increased data storage capacity of the cloud. We perform the Face Recognition application using 1000 faces from the Labeled Faces in the Wild database [25]. The images in this database average 20 kB in size. Unlike the previous experiments where the database had an equal amount of faces per person, this face database has ~400 people with a random number of faces per person. When running the applications on the mobile device it is interesting to see that every experiment failed with an `OutOfMemory` error. This supports our notion that mobile devices cannot run all applications due to the lack of computational power. Using the same conditions we run the same experiment on the private cloud. Due to the enhanced processing power of this particular private cloud, the results favored this computational resource. This establishes the notion that public clouds are not necessarily the most powerful computational resource in every case.

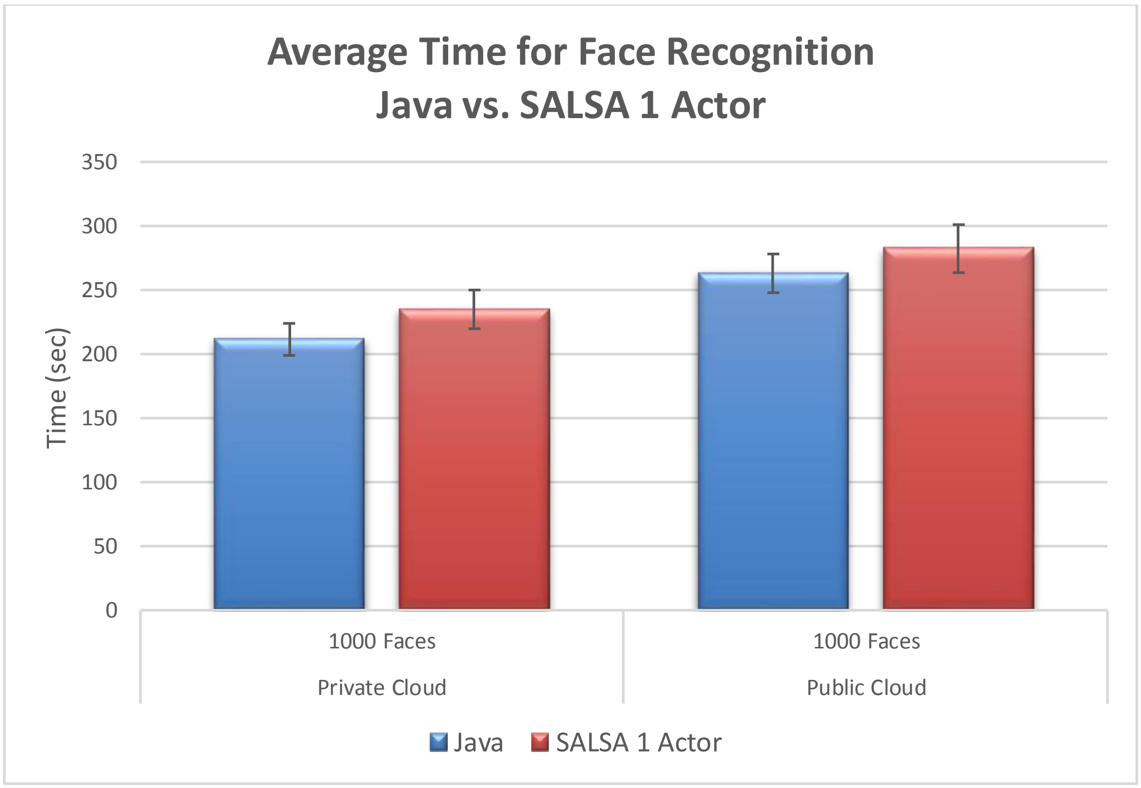

Figure 5.17: Average Time for Java vs. SALSA 1 Actor for 1000 Faces

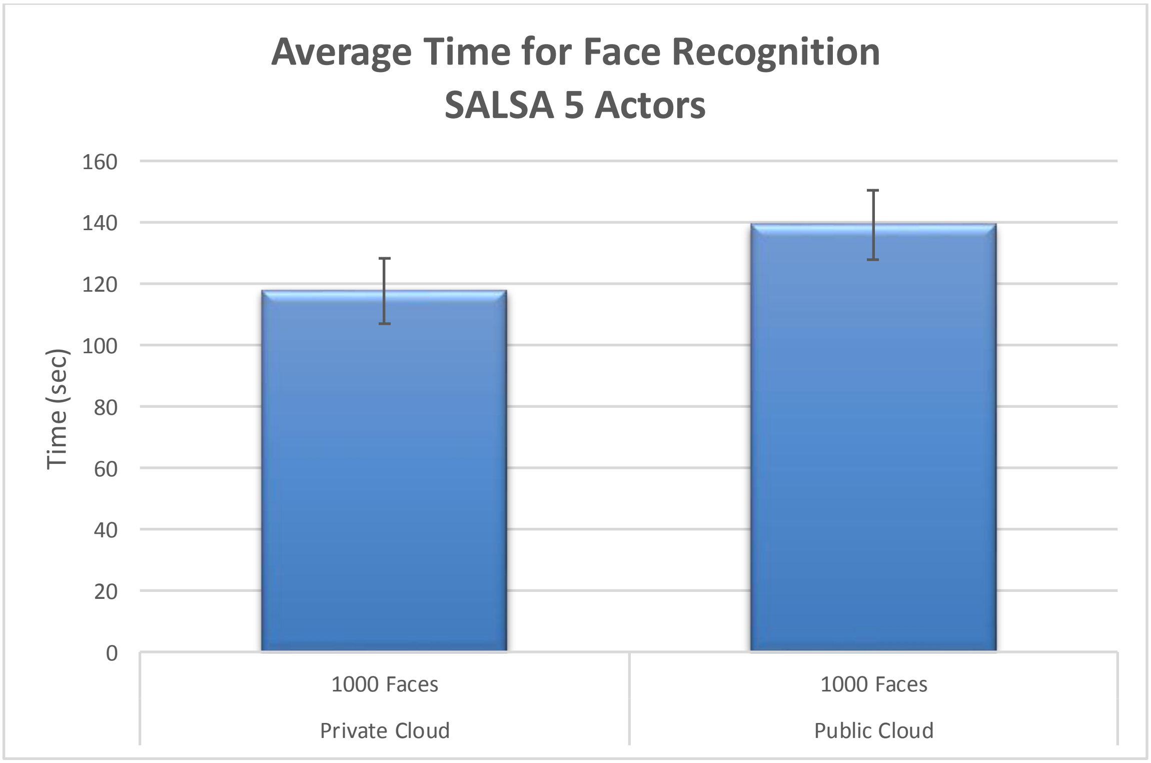

Figure 5.18: Average Time for SALSA 5 Actors for 1000 Faces

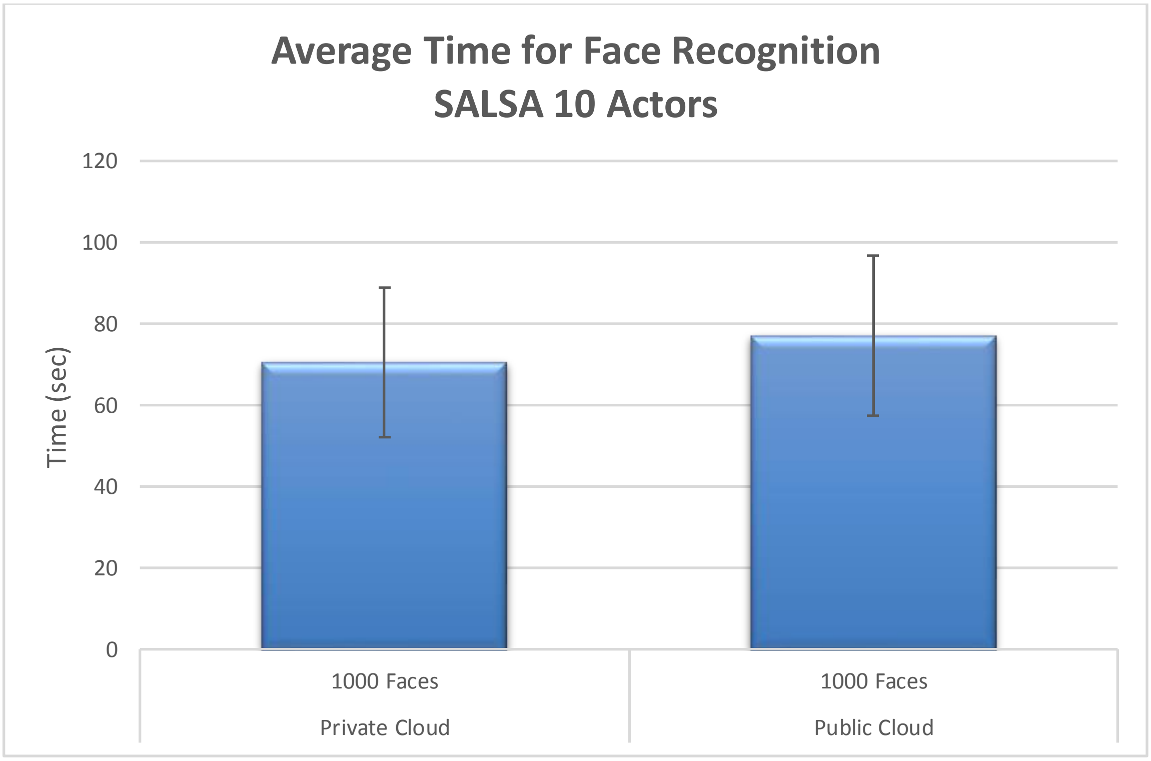

Figure 5.19: Average Time for SALSA 10 Actors for 1000 Faces

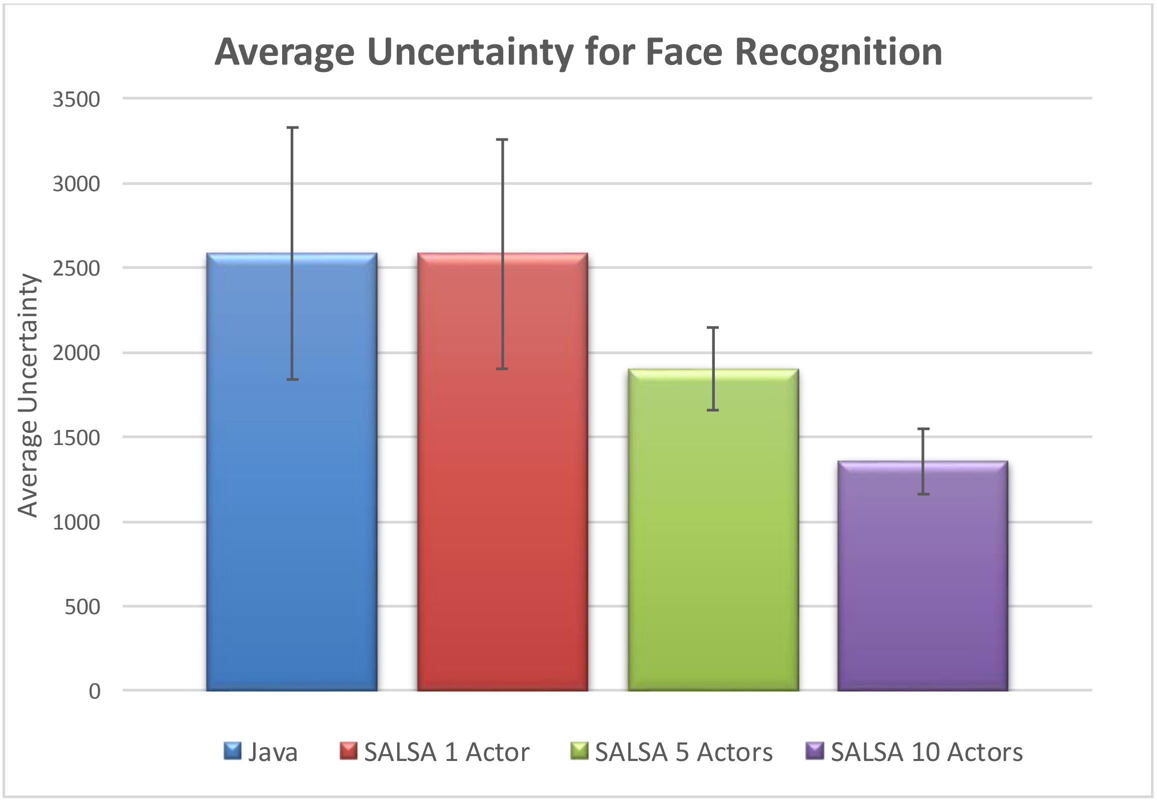

Figure 5.20: Average Uncertainty for 1000 Faces

5.2.6 Improvements by utilizing MobileCCA

Table 5.2: Speedup of Time for Java vs. SALSA

		Speedup		
Experiment	**Face Database**	**Mobile Device**	**Private Cloud**	**Public Cloud**
Java vs. SALSA 1 Actor	1000	-	0.902	0.931
	100	-	1.055	-
	50	0.851	0.897	-
	20	0.397	0.729	-
Java vs. SALSA 5 Actors	1000	-	1.795	1.888
	100	1.998	1.365	-
	50	1.575	0.946	-
	20	0.473	0.946	-
Java vs. SALSA 10 Actors	1000	-	2.999	3.413
	100	2.386	1.524	-
	50	1.724	0.558	-
	20	0.408	0.134	-

Table 5.3: Reduction of Uncertainty for Java vs. SALSA

		Reduction			
Experiment	**Face Database**	**Mobile Device**	**Private Cloud**	**Public Cloud**	**Misses**
Java vs. SALSA 1 Actor	1000	-	1	1.001	0
	100	-	0.996	-	0
	50	1.006	1	-	30
	20	0.745	1	-	30
Java vs. SALSA 5 Actors	1000	-	1.231	1.360	0
	100	1.192	1.144	-	0
	50	1.538	1.538	-	40
	20	1.380	1.854	-	60
Java vs. SALSA 10 Actors	1000	-	1.864	1.906	0
	100	1.112	1.077	-	0
	50	2.290	2.290	-	60
	20	5.408	7.267	-	70

Table 5.2 and Table 5.3 represent the speedup of time and reduction of uncertainty in terms of confidence respectively for the Java vs. SALSA experiments. In both tables we see that when comparing Java vs. SALSA with 1 actor there is no benefit since we must take into consideration the overhead of migrating the actor. We observe that when we increase the number of actors we see a speedup in time and a reduction in uncertainty of results. This is due to the fact that the application is now running concurrently and that the database of faces that each actor has is smaller. Although we incur a lower uncertainty, it is important to see that we also incur more misses since there is a smaller database of faces and thus smaller images to test against.

Table 5.4: Mobile Device to Private Cloud Average Speedup

Type	**Mobile Device to Private Cloud Average Speedup**
Java	5.00
SALSA 1 Actor	7.85
SALSA 5 Actors	5.11
SALSA 10 Actors	2.06
Total Average	5.00

Table 5.5: Private Cloud to Public Cloud Average Speedup

Type	**Private Cloud to Public Cloud Average Speedup**
Java	0.80
SALSA 1 Actor	0.83
SALSA 5 Actors	0.85
SALSA 10 Actors	0.91
Total Average	0.85

Table 5.4, and Table 5.5 represents the average speedup of moving from one computational resource to another. Based on the computational resources used in these experiments, Table 5.1, we can see a ~5x speedup by moving from the mobile device to the private cloud. Moving from the private cloud to the public cloud we actually saw an increase in execution time. This is due to the fact that this particular private cloud is more powerful than the public cloud.

Conclusions drawn from these experiments are that clearly the private and public cloud was the best choice to migrate computation. With a larger storage capacity, more images can be added to the database leading to more accurate results. We also see that having a large amount of faces on the mobile device may not be the best option. Not only is the storage capacity smaller but the computation time is much greater than that of the private cloud. What is interesting to see is that smaller amount of faces on the mobile device provides low uncertainty results when used with multiple actors. This fits well with the use and safety of data on mobile devices. Friends and family’s faces can be located on the mobiles device while broader faces can be located on private and public clouds such as friends of friends, strangers, celebrities, etc. This way actors can be sent to the phone as well as other resources and provide a broad search of the unknown face while still being efficient.

6. Related Work

6.1 Mobile Local Resources

Similar to *Hyrax*, mentioned in Section 2.2.2, N. Fernando et al. [26] introduces an approach that does not require the need to build an infrastructure to offload heavy computational work. Instead of using known cloud services such as Amazon EC2, a local cloud is created using nearby mobile devices.

This mobile cloud is a conceivable idea considering the popularity of smart phones. Smart phones have integrated Bluetooth allowing the transfer of agents to occur even when Wi-Fi or 3G is unavailable. The natural mobility of mobile devices allows this mobile cloud to appear in any locations. A "home" mobile cloud can exist for your family's mobile devices grouped together, and can easily be moved to a "work" mobile cloud with your colleague's mobile devices. The cost-benefit analysis is still used to accurately determine which devices are good candidates. Cost-benefit analysis is simply a middleware to gather statistics based on some factors including limited battery power, limited cellular signal, data access fees and availability of local resources.

This approach is similar to ours in the sense that they are creating "mobile clouds" out of various mobile devices in a vicinity, but through *MobileCCA* we do not limit ourselves strictly to this. The ability to control the movement of actors allows us to simulate these "mobile clouds" but also gives us the ability to scale out to public resources if needed.

6.2 Mobile Cloud Middleware Performance

H. Flores et al. [27] describes a design engine that is located on the mobile device to determine whether to offload data or not. This decision engine uses the same constraints such as bandwidth, size of data, and other aspects of the device. The phone offloads data using the *Mobile Cloud Middleware (MCM)*. A benefit of the *MCM* includes hiding the complexity of mobile cloud providers using different Web APIs.

This research has proved that offloading from a mobile device is beneficial. *MCM* is currently used to determine which cloud service on which to execute their application.

Expanding the *MCM* system to run the decision engine would allow all phones connecting to this middleware to use the feature.

6.3 Cloud Operating System

Imai et al. [20] present a middleware framework known as the *Cloud Operating System (COS)* that supports autonomous workload elasticity and scalability based on application-level migration as a reconfiguration strategy. Based on the actor-oriented programming language, SALSA, the framework implements reconfiguration strategy based on the workload of VMs.

When utilization on VMs are past a threshold, actors will migrate to newly created VMs. In contrast, as utilization in VMs decrease, actors will consolidate together and terminate idle VMs. *COS* provides a layer of abstraction to application developers. Rather than dealing with resource management themselves, developers can delegate this to the middleware and allow the nature of the application to control the configuration of actors.

7. Discussion

7.1 Conclusion

Approaches to Mobile Cloud Computing have circled around the idea of offloading computation to various computational resources. This idea, however, has certain restrictions, whether it is constricting applications to fit into middleware, only offloading to certain resources, or having no control in the offloading process. In this thesis we have provided an approach that solves these problems.

Our cloud-based mobile augmentation strategy is known as *MobileCCA*. By leveraging the actor model in Mobile Cloud Computing we can create dynamically reconfigurable applications very easily. This opens the ability to move actors to any computational resource that is able to run a compatible VM. Specifically in SALSA, since Java is the base language, any device that is able to run Java code can run these SALSA applications. Therefore Android devices, laptops, desktops, private and public clouds will be suitable for actor migration. In a world full of heterogeneity, the ability to write one SALSA application that is compatible on all devices is especially important because it can save a lot of time and effort. An application now consisting of actors can be offloaded to different computational resources and can experience different behaviors on each. Therefore, policies can be introduced to control the movement of actors to obtain a given goal.

We provide an example of this approach by using the actor-oriented programming language SALSA. By creating a SALSA application, Face Recognition, we were able to perform experiments on the mobile device, private cloud, and public cloud. We were able to scale up applications without the need for extensive developer involvement. By scaling up we can dramatically reduce execution time. These speedups were on average ~5x faster when moving from the mobile device to the private cloud.

With mobile devices growing sharply, finding an efficient and developer-friendly approach to Mobile Cloud Computing is important. We hope that the *MobileCCA* approach will help developers create scalable, efficient mobile applications to computationally expensive problems.

7.2 Future Work

In the future we hope that developers will use *MobileCCA* to create useful applications. For example, smartphones are now carrying onboard fingerprint sensors. A useful application can be fingerprint recognition for police officers. This application would consist of a training phase that trains over a database of fingerprints similar to the Face Recognition application. When a user attempts to recognize a fingerprint, actors can be sent to the mobile device, private cloud, and public cloud. As stated in this thesis, all of these resources have their limitations. For example, the mobile device might have the storage capacity to store hundreds of fingerprints (most wanted criminals in the area), the private cloud might holds thousands (criminals in the city), and finally the public cloud can hold millions (criminals in the state). Imagine a scenario where an officer pulls over a suspect. As the suspect's fingerprint is recognized, actors can be sent to the mobile device, private cloud, and public cloud, all training over a different dataset. This creates a comprehensive search over a large database of fingerprints in an optimal way.

LITERATURE CITED

[1] "Smartphone Users Worldwide Will Total 1.75 Billion in 2014," eMarketer New York, NY, USA 2014. [Online]. Available: http://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536. Accessed on: Oct. 26, 2014.

[2] S. Abolfazli, Z. Sanaei, E. Ahmed, A. Gani and R. Buyya, "Cloud-Based Augmentation for Mobile Devices: Motivation, Taxonomies, and Open Challenges," *Commun., Surveys & Tutorials, IEEE*, vol. 16, no. 1, pp. 337-368, First Quarter 2014.

[3] C. Varela and G. Agha, "Programming Dynamically Reconfigurable Open Systems with SALSA," *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technol. Track Proc.*, vol. 36, no. 12, pp. 20-34, Dec. 2001.

[4] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. Accessed on: Oct. 20, 2014.

[5] "Samsung Galaxy S5," GSMArena, 2014. [Online]. Available: http://www.gsmarena.com/samsung_galaxy_s5-6033.php. Accessed on: Oct. 26, 2014.

[6] "Samsung Galaxy Tab Pro 12.2 32GB (Wi-Fi)," Samsung, Seoul, South Korea, 2014. [Online]. Available: http://www.samsung.com/us/mobile/galaxy-tab/SM-T9000ZWAXAR. Accessed on: Oct. 26, 2014.

[7] "Amazon EC2 Instances," Amazon, Seattle, WA, USA, 2014. [Online]. Available: http://aws.amazon.com/ec2/instance-types. Accessed on: Oct. 26, 2014.

[8] "Amazon EC2," Amazon, Seattle, WA, USA, 2014. [Online]. Available: http://aws.amazon.com/ec2. Accessed on: Oct. 26, 2014.

[9] G. P. Perrucci, F. H. P. Fitzek, and J. Widmer, "Survey on energy consumption entities on the smartphone platform," in *Proc. IEEE 73rd Veh. Technol. Conf. (VTC Spring)*, Budapest, Hungary, May 2011, pp. 1–6.

[10] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy," *Comput.*, vol. 43, no. 4, pp. 51- 56, April 2010.

[11] K. Mun, "Mobile Cloud Computing Challenges," Alcatel Lucent, 2010. [Online]. Available: http://www2.alcatel-lucent.com/techzine/mobile-cloud-computing-challenges/. Accessed on: Oct. 26, 2014.

[12] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “CloneCloud: Elastic execution between mobile device and cloud,” in *Proc. ACM The European Professional Soc. on Comput. Syst. (EuroSys’11),* Salzburg, Austria, Apr. 2011, pp. 301–314.

[13] T. Soyata, R. Muraleedharan, C. Funai, M. Kwon, and W. Heinzelman, “Cloud-Vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture,” in *Proc. IEEE ISCC ’12,* Cappadocia, Turkey, Jul. 2012, pp. 59–66.

[14] E. E. Marinelli, “Hyrax: cloud computing on mobile devices using MapReduce,” M.S. thesis, Comput. Sci. Dept., Carnegie Mellon U., Pittsburgh, PA, 2009.

[15] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. of the 6th Symp. on Operating Syst. Design and Implementation,* San Francisco, CA, Dec. 2004.

[16] A. Silberschatz, P. B. Galvin and G. Gagne, "Threads," in *Operating System Concepts*, 9th ed. Hoboken, NJ, USA: Wiley, 2012, ch.4, sec.1, pp. 163-202.

[17] "More from Android," Google, Mountain View, CA, USA. [Online]. Available: http://www.android.com. Accessed on: Oct. 26, 2014.

[18] "Dare: Dalvik Retargeting," Penn State U., State College, PA, USA. [Online]. Available: http://siis.cse.psu.edu/appanalysis.html. Accessed on: Oct. 15, 2014.

[19] R. Llamas, . R. Reith and K. Nagamine, "Smartphone OS Market Share, Q2 2014," Int. Data Corporation, Framingham, MA, USA. [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp. Accessed on: Nov. 1, 2014.

[20] S. Imai, T. Chestna and C. Varela, "Elastic Scalable Cloud Computing Using Application-Level Migration," in *Proc. of the 2012 IEEE/ACM 5th Int. Conf. on Utility and Cloud Computing.*, 2012, pp. 91-98.

[21] "Open Source Computer Vision," Willow Garage, Menlo Park, CA, USA. [Online]. Available: http://opencv.org/. Accessed on: June. 14, 2014.

[22] S. Imai, T. Chestna and C. A. Varela, "Accurate Resource Prediction for Hybrid IaaS Clouds Using Workload-Tailored Elastic Compute Units," in *Proc. of the 2013 IEEE/ACM 6th Int. Conf. on Utility and Cloud Computing.*, 2013, pp. 171-178.

[23] Vision and Modeling Group, "Eigenfaces for Recognition," *J. of Cognitive Neuroscience,* vol. 3, no. 1, pp. 71-86, 1992.

[24] "Georgia Tech face database," Georgia Inst. of Technol., Atlanta, GA, USA. [Online]. Available: http://www.anefian.com/research/face_reco.htm. Accessed on: Oct. 20, 2014.

[25] G. B. Huang, M. Ramesh, T. Berg and E. Learned-Miller, "Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments," in *Int. Conf. on Comput. Vision*, Rio de Janeiro, 2007.

[26] N. Fernando, S. Loke, and W. Rahayu, “Mobile cloud computing: A survey,” *Future Generation Comput. Syst.*, vol. 29, no. 1, pp. 84–106, 2012.

[27] H. Flores, S. N. Srirama, and R. Buyya, “Computational offloading or data binding? bridging the cloud infrastructure to the proximity of the mobile user,” in *Proc. of the 2nd IEEE Int. Conf. on Mobile Cloud Computing, Services, and Eng.*, 2014, pp. 10-18.

[28] C. Varela, "Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination," Ph.D. dissertation, Comput. Sci. Dept., U. of Illinois at Urbana-Champaign, Champaign, IL, 2001.

[29] S. Imai, P. Patel and C. A. Varela, "Developing Elastic Software for the Cloud," in *Encyclopedia on Cloud Computing*, Hoboken, NJ, USA: Wiley, 2014.