mPaaS: Delivering Mobile Platforms as a Cloud Service

Arpit Gupta, Joice John, Raghav Tripathi
Department of Computer Science
North Carolina State University
{agupta13, jjohn@ncsu.edu, rtripat@ncsu.edu}@ncsu.edu

ABSTRACT

Ubiquity of mobile devices has increased significantly with advent of smart-phones. The pervasive usage demands multiple mobile phones or platforms for various usage scenarios. The proposed solution in this direction is in-device mobile virtualization which is limited by power and computation limited nature of these devices. In this work we propose to combine mobile virtualization with cloud resources and deliver mobile platforms as cloud services. We identified to support such a service virtualization of mobile sensors is critical and in this work we present our solution to enable this virtualization. We test the performance of our devices for latency and observe that for most of the scenarios it provides desired performance. We further discuss limitations of our solution and discuss future improvements. We believe this work should serve as the starting point to enable delivery of mobile platforms as a cloud service and expect future solutions to further refine our solution.

Categories and Subject Descriptors

D.4.7 [**Organization and Design**]: Distributed systems

General Terms

Design, Experimentation, Performance

Keywords

Sensor Devices, Sensor Virtualization, Virtual Machines, Cloud, Mobile Devices, Mobile Platforms

1. INTRODUCTION

Mobile devices in recent past have transformed from simple cellular phones to multi-purpose utility devices. It is equally relevant for personal, production and developmental environments. The overlap of usage environments is very common these days, for e.g.. a user uses the same mobile device for corporate and personal usage. For various scenarios such an overlap is not acceptable for reasons related to security and privacy issues. Currently most of the solutions force users to carry multiple physical devices for different usage scenarios. The discomfort of carrying multiple mobile devices inspires the development of virtual mobile platforms. An ideal mobile virtualization solution liberates users from platform bondages as news mobile platforms can run over traditional ones. This should enrich user experiences and catalyse growth of better mobile platforms.

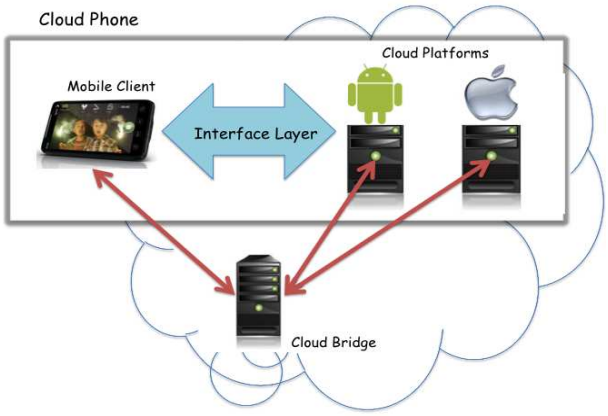

Figure 1: Conceptual representation of delivery of mobile platforms as a cloud service. Mobile Platforms run over cloud servers and devices access them as thin clients. These cloud platforms require virtualization of sensor devices present at Mobile Client

Mobile virtualization differs from various desktop or server virtualization solutions because of power, and computation limited nature of these devices. This combined with growing relevance of cloud services, motivates us to explore the option of offloading the virtualization activities. We fancy a solution in which mobile platform runs over a cloud server and mobile device connect to it as a thin client. Various solutions in this direction have been proposed [17] and is an area of active research. We observe sensor devices are intrinsic part of smart phones which differentiates them from other thin client solutions. There are two directions in which power of these sensor devices can be leveraged by the mobile platforms running over the cloud (cloud platforms). One approach is to develop sensor device

APIs which can be utilized by all the applications running over these cloud platforms. Such a solution will require mobile-apps to be 1) aware of nature of underlying mobile platform and 2) update the existing applications. This does not goes down well with application developers and mobile-apps eco-system in general. Another approach is to virtualize these sensor devices such that the operating system can use them as local devices, which enables user space agnostic solution. In this paper we focus on such *mobile sensor virtualization* which enables availability of mobile sensors for mobile platforms running over the cloud servers.

In this paper we developed a mobile sensor virtualization to enable delivery of mobile platforms as a cloud service. We developed cloudified version of commonly used mobile sensor device drivers for cloud platforms. These drivers when requested interact with the mobile device to extract sensor information. Our solution requires no changes to mobile apps running over cloud platform or mobile platform running over client's mobile device. It introduces very small computational overhead and enables concurrent usage of multiple and diverse mobile platforms, for e.g. it is possible to run 8-10 mobile platforms running Linux based Android, Firefox, Ubuntu etc. with minimal performance overhead.

In this paper we also introduce the concept of cloud-bridge to offload the complexities of handling sensor device requests from multiple platforms. Apart from developing the drivers for cloud platform's sensor devices, we also developed cloud-bridge which multiplexes device requests, apply security and privacy policies, maps device instructions to format understood by client device, schedules requests and deliver corresponding responses to respective platforms. We analyze the performance of these virtual devices in terms of end-end latency under various usage scenarios. Our results show that latency performances were within the acceptable performance limit of 150ms perceivable to human eyes [8].

These days main focus for mobile system researchers is to enhance computational prowess and provide platform flexibility. Works like MUAI etc. [14, 18, 11, 12, 13] focus on enhancing computational prowess only and works like MVP, Cells etc. [9, 7] focus on platform flexibility only. We focus on combining these two disparate approaches to virtualize mobile platforms over cloud servers providing both better platform flexibility and computational prowess. Some solutions like Virtualized Screen [17] conceptualize similar ideas. We identified that presence of mobile sensors uniquely define mobile platforms and virtualization of these sensors for platforms running over cloud is essential. To best of our knowledge this is the first work which focuses on virtualization of mobile sensors for cloud based mobile platforms.

In the ensuing sections we start with discussion on various high level design choices we considered. We further discuss implementations of various components developed for mobile client, cloud platforms etc. in Section 2. We discuss our experimental set-up and experiments in Section 3. Section 4 discusses current implementation at a higher level and explores how it can handle various challenges. It also talks about future enhancements to solve some obvious problems. We conclude our findings in Section 6 after going through related works in Section 5.

2. DESIGN & IMPLEMENTATION

2.1 Design Choices

We will begin with discussion on various high level design choices we encountered. Our proposed solution is unique in the sense that we virtualize sensor devices for platforms which are running remotely over cloud compare to traditional in-device virtualization solutions.

In-device vs Cloud Based Virtualization: In spite of major advances in Mobile Computing, a lack of battery life remains an issue with mobile devices. This problem is further compounded when we have multiple virtual phones running on a device as each of those virtual phones will hog the limited resources. If we can offload some of the computation to the cloud, we can reduce power consumption at the mobile device level and have access to potentially large amount of compute and storage capability. One potential issue with this approach is that network latency might be high when the mobile device communicates over the network. However, with the advent of 4G networks and increasing bandwidth [15], network latency can be contained well within the 150 millisecond threshold of lag perception by humans [8]. Thus any compute or virtualization overhead associated with running multiple platforms can be moved over to the cloud, conserving the limited power of the mobile client.

Full vs Para Virtualization: In delivering Mobile Platform as a Service, a unique problem is that the mobile device and the virtual machines are separated. This is different from the traditional sense of virtualization where the guest VMs resides on the host machine. A major challenge here becomes emulating the presence of the mobile sensors on the platforms running on the cloud. Full Virtualization would have meant we would have to do binary translations of the device driver code. This entailed a lot of overhead. Instead we wanted to conform to the legacy implementation with minimal code changes to already existing device driver code. We opted for a middle ground between full and para virtualization. We modified already existing device drivers for LED and Accelerometer. This has the advantage of legacy implementation, because it makes it easier to open source and contribute to. Also since

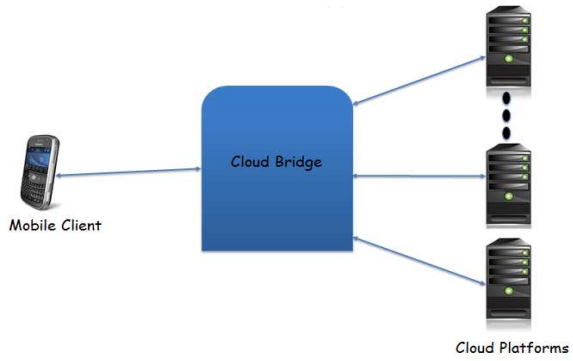

Figure 2: Our System consists of three major components. CloudPlatforms are mobile platforms running over cloud servers. The sensor device related instructions are sent from these platforms to CloudBridge which handles various such platforms. It sends the device instructions to MobileClient. Responses from the sensor devices follow the reverse direction

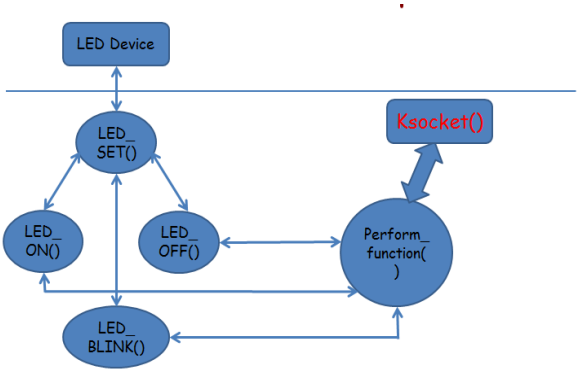

Figure 3: Structure of legacy implementations for LED device drivers. The only change we made was replacement of bios function with ksockets. Ksocket send the instructions to Cloud Bridge which in return provides response from the respective Mobile Client

the number of sensor devices are actually limited in a mobile phone, it is easier to modify the device drivers than do binary translation of instructions.

User Space Agnostic Design: We wanted to ensure that the design of our system would remain application agnostic. From a mobile application developers perspective, the task offloading to the cloud is not visible to him/her and consequently he should not be making any changes in the code he/she writes. From the mobile device side we wanted to use user codes that are already available as part of standard APIs.

Since we have multiple platforms residing in the cloud, management of different platforms becomes an issue. To avoid mobile client to manage this complexity we propose usage of *CloudBridge*. It serves as a multiplexer between these different platforms. It is responsible for instruction mapping, all policy handling mechanisms can be implemented here. In addition to this any security related policies can also be implemented here.

2.2 Implementation

After discussing our design choices we will shall discuss implementation of various components. As shown in Figure 2, the major components of our solution are 1) Cloud Platform, 2) Mobile Client, and 3) Cloud Bridge. We will now discuss each of these components in detail.

2.2.1 Cloud Platforms

These are the VMs which run the mobile platform over cloud server and the sensors are virtualized for these platforms itself. We chose to virtualize 2 devices LED and Accelerometer as part of the project. The rationale being, these two devices are representative of the 2 major types of sensors in mobile phones. LED just receives instructions whereas Accelerometer both sends data and receives instructions. As we discussed previously, we desire to leverage existing legacy implementations of device drivers so that it is easier for open-source community to contribute to our device drivers.

LED: Basic functionalities of LED includes ON/ OFF/ BLINK. The existing implementations used BIOS arguments to turn on and off the LED device. We modified the feature to use kernel sockets. Using kernel sockets the LED device can communicate with the CloudBridge. As shown in Figure 3, we decided to use kernel sockets instead of user space sockets to avoid the overhead of world switching between kernel and user space. For every LED device registered, a *sysfs* entry in created in */sys/class*. *sysfs* is a virtual file system provided by Linux. *sysfs* exports information about devices and drivers from kernel space to user space and is also used for configuration [2]. It is similar to the *sysctl* mechanism found in BSD systems.

Accelerometer: An accelerometer is a sensor device that measures acceleration in the three dimensions. Accelerometers are increasingly being incorporated into personal electronic devices to detect the orientation of the device, for example, a display screen. The basic functionalities include receiving ON/OFF instructions, updating delay and sending sensor data. The existing implementation of Accelerometer was intrinsically linked with bus drivers. We took inspiration from the Virtual Mouse Driver Code. As shown in Figure 4, we integrated this with kernel socket library so that it can send and receive data from the Cloud Bridge. For toggling, ON/OFF we create a *sysfs* entry called Enable and to toggle the rate at which data comes in another *sysfs* entry called Rate is also created. Whenever data is received, the three coordinates are wrapped in an

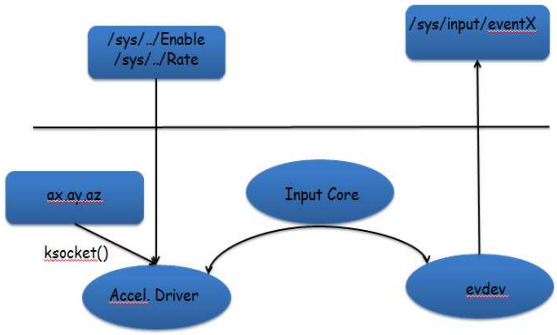

Figure 4: Structure of our accelerometer device driver. The *sysfs* parameter control state of device and rate at which sensor data needs to be reported. Accelerometer values received from Cloud Bridge are reported back to respective *evdev* events

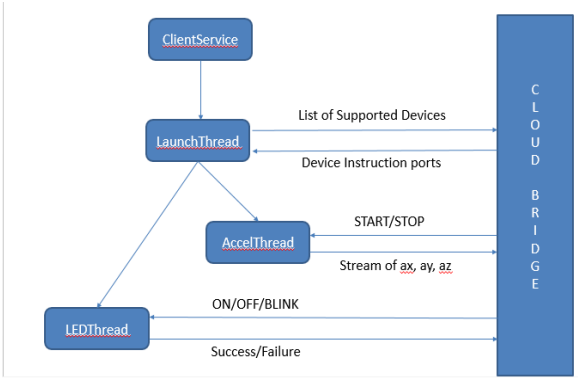

Figure 5: Structure of the Client Service interacting with Cloud Bridge to enable virtualization of sensor devices

evdev packet and sent to *evdev*. *evdev* is a generic Linux framework for all input devices like keyboard, mouse. It interfaces to user space through /sys/class/input interface. Currently we are not using *evdev*, but rather a *sysfs* entry for this purpose.

2.2.2 Mobile Client

The MobileClient is the device at user end which physically hosts the sensor devices which need to be virtualized for Cloud Platforms. It provides an application layer interface to these sensor devices avoiding need to make any changes in the Mobile Client itself. We used a device running Android 4.2.2 as the client. It essentially performs 2 functions

- Integrate with Android sensor framework to fetch accelerometer data at desired rate
- Integrate with Android notification framework to perform operations of phone LED like ON/ OFF/ BLINK
- Interface with CloudBridge

Our goal was to minimize complexities at the MobileClient due to the limited processing and power resources of a phone. As shown in Figure 5, it starts a long running background service on users instruction. The background service spawns the LaunchThread which on load creates a list of supported devices on the phone such as LED and accelerometer. Then, the LaunchThread is responsible for making a one time connection to CloudBridge over its management port and send it the following information:

- Mobile phone MAC address
- Number of platforms required
- Type of platform required, currently, only Android supported
- List of supported device ID's

The response from CloudBridge determines the next actions taken at the mobile client. For each of the device that is supported by the platform being served from the cloud, the bridge returns a device instruction port. In order to scale the communication when more and more devices are supported we spawn new device threads for each supported device.

The goal of the device threads is to accept instructions over their specific instruction port from the Cloud Bridge then execute the instruction and send back *success* or *failure* or *data* (in case of accelerometer).

2.2.3 Cloud Bridge

CloudBridge is the lynchpin of the system. Its main task is to perform multiplexing between multiple CloudPlatforms and MobileClient. Most of the complexities of the system should lie in the CloudBridge such as policy checks, instruction verification so that there is minimum complexity at the MobileClient which is our goal. CloudBridge is also essential to support scaling to multiple CloudPlatforms. We deem the CloudBridge should be in close proximity to the MobileClient to minimize network delays in their communication and hence save on precious battery power of the MobileClient.

The CloudBridge initializes and maintains a pool of CloudPlatforms currently registered with the system. It has a hash table indexed by the MAC address of the CloudPlatform which stores the CloudPlatform operating system and its current status whether it is assigned to a MobileClient or not.

When a MobileClient connects to the management port of CloudBridge, the CloudBridge iterates over its platform pool and assigns the required number of available CloudPlatforms to the MobileClient. Using this

information the CloudBridge knows how to multiplex a request from the CloudPlatform and send it to the assigned MobileClient and vice versa. For each of the sensor device supported by the MobileClient like LED and accelerometer, the CloudBridge opens a server socket to send instructions and sends its details to the MobileClient. This connection socket object is also stored as part of the hash table entry so that it can be retrieved later and be used to immediately send instructions to mobile client. This saves us the time required for socket creation during the instruction execution phase.

Each request from the CloudPlatform is accompanied by its unique MAC address. We made use of instances of the python ThreadedTCPServer class to accept instructions from the CloudPlatform and then create an instruction object. The instruction object is then inserted into a global queue. The InstructionQueueScheduler thread then retrieves an instruction from the queue and executes it by sending the instruction like ON, BLINK etc. over the instruction connection object stored. It then creates a data socket to accept Success or Failure in case of LED or a stream of acceleration values in case of accelerometer. We faced a bottleneck during the scaling platform test case where in we were getting the Socket in Use exception even though we randomly generate port numbers from 10,000 to 50,000 range. This is owing to the fact when 8 platforms were sending request to the bridge, the data sockets were getting created very frequently resulting in some ports being used. To fix this issue we had to set the socket option to reuse address in case of conflict.

3. EXPERIMENTAL EVALUATION

The performance of the system is measured in terms of the end-to-end latencies for requests originating from the CloudPlatforms and the response being returned after execution at the MobileClient. Our experimental set-up consists of

- Galaxy Nexus as MobileClient running on Android Jellybean
- Servers used for CloudBridge with global IPs were *tabak.csc.ncsu.edu* and *arpit.csc.ncsu.edu*. Each having 4 CPU cores clocked at 2.40 GHz, 8 GB memory and running 64-bit Ubuntu
- Amazon EC2 t-1 micro instances as CloudPlatform. Each instance has a single core CPU clocked at 2.4 GHz, 613 MB memory running 64-bit Ubuntu

3.1 Locality

We tested the system to determine changes in performance as a single CloudPlatform is moved from a local host in the network to the cloud. The CloudPlatform sends 500 consecutive LED ON and LED BLINK requests with a sleep of half a second in between.

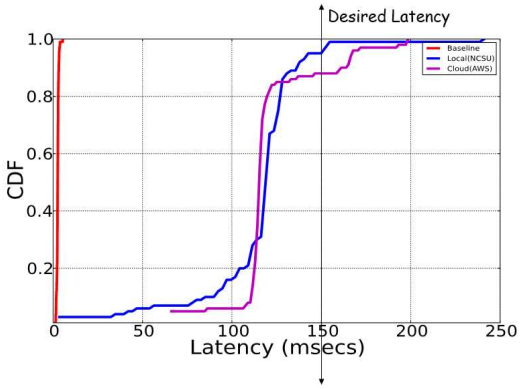

Figure 6: End-to-end latency distributions for various scenarios. Baseline is the case where both CloudPlatform and CloudBridge are hosted over a single machine. Local(NCSU) is the case when we host CloudPlatform over another server in NCSU netowork itself and Cloud is the case when it is hosted over Amazon EC-2 instance. All these latencies fall well-within the desired latency limit of 150ms

The baseline case is where all three, MobileClient (a python emulator), CloudBridge and CloudPlatform are running on the same host. Since, there is no network latency we expected the round trip time for each request to be in the order of tens of milliseconds. As can be seen from the Figure 6, the latency for baseline case is indeed very less. Almost 100% of the requests are below 10ms threshold.

Second case is where the MobileClient is running on the Galaxy Nexus phone in the same network as the CloudBridge and CloudPlatform which reside on the host tabak.csc.ncsu.edu. There is an expected rise in the latency as we now have a component which doesn't reside on the same host. As we can see, more than 90% of the requests are below 150ms which is the threshold for human perception for lag.

Whereas the third case is where the CloudBridge is on tabak.csc.ncsu.edu (a host in the same network as MobileClient) and the CloudPlatform is an Amazon EC2 instance. The network latency is well below 150ms threshold for almost 80% of all the requests.

3.2 Component-wise Latency

We also analysed the time spent at each component during a successfully completed request. As illustrated in the Figure 7. T1 indicates time measurement for end to end for Cloud Platform. T2 indicates the time spent on the CloudBridge per instruction object, in other words it measures the time taken for steps like creating data sockets, instruction queue processing etc. T3 indicates the amount of time it takes for the Android sensor code to execute on the MobileClient. We measured that on an average, a ping query from an Amazon EC2 in-

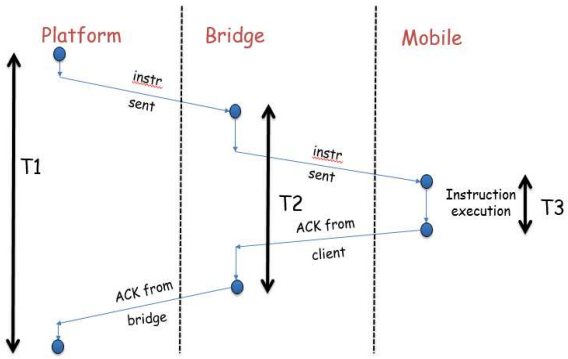

Figure 7: End-to-end timing diagram for a single instruction from Cloud Platform and its corresponding response. It shows various measurements which are observed at different components of the system

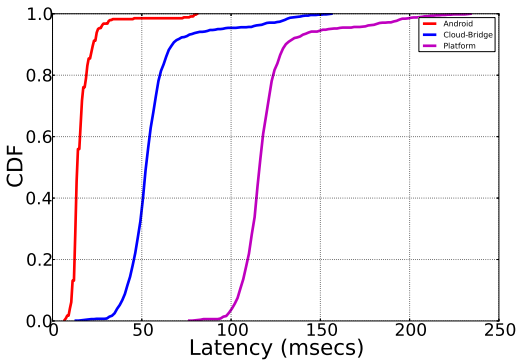

Figure 8: Latency distribution of time spent in 3 components. As evident from Figure 7, difference in T1 and T2 represents time taken to process the instruction and its response at server and network latencies between Cloud Platform and Cloud Bridge

stance to CloudBridge took around 40ms. Whereas a ping query from CloudBridge to the Mobile Client took around 10ms owing to the fact that both were running on the same network.

The experiment was conducted with CloudPlatform issuing 500 LED ON and LED BLINK requests consecutively with half a second of sleep in between. The Figure 9. demonstrates that the MobileClient on Android took an insignificant amount of time to execute. As expected, Cloud Bridge took the most amount of time when compared to other components. This can be attributed to the fact that in the CloudBridge, we are creating sockets every time we receive an instruction. This ephemeral ports creation poses an overhead in the CloudBridge. One way to remedy this is to re-use the sockets for both sending and receiving data.

3.3 Platform Scaling

We also tested the system performance while increasing the number of CloudPlatforms assigned to a MobileClient from 1, 2 and 4 . This case was done to test for existence of any performance bottlenecks when multiple CloudPlatform are sending requests to the CloudBridge.

As illustrated in Figure 10. for 1 and 2 platforms, more than 90% of the requests were within the 150ms threshold. When we scale up the number of platforms to 4, we see an increase in the latency as almost 50% of the requests are within the 150ms threshold. As the number of platforms is increased to 8 (Figure 11), we can see that close to 40% requests are within the 150ms threshold. This increase in time can be attributed to the delay occurring at CloudBridge, where all pending instructions are queued.

4. DISCUSSION

So far we have discussed how important is virtualization of sensor devices for mobile platforms running over the cloud servers. We discussed various design choices and functionalities of different components of our implementations. We analysed the performance of these sensor devices in terms of latency under various testing scenarios. We will now discuss various high level challenges for our solution and how we dealt with them.

Network Latency & Bandwidth: In-spite of fancy stories about magic of cloud resources, we observe significant variations in performance of cloud servers. It is evident from our scaling experiments in which we observed high level of inconsistencies in latency/ computational performances for cloud servers. We expect solution to such problems should result in better latency performances for our virtualized sensor devices. All our experiments focused on utilizing WiFi (802.11g) as access network and currently we believe our approach is not suited for cellular networks with higher variance in network latencies. It restricts ubiquity of usage, but we expect performance of cellular networks to improve in near future. It should be also noted that we expect better access network performance with usage of WiFox and 802.11 ac/n etc. [15].

Cloud Bridge Complexities: From our experiments exploring various factors contributing to sensor device's latencies, we observed significant time spent by an instruction at Cloud Bridge. We observed this overhead to increase as it served multiple cloud platforms. Limited computational prowess of the Cloud Bridge machine was one factor, but we need to design its functionalities to optimize for latency performance too. One area which we can focus on is to reduce the number of parallel threads created at Cloud Bridge. Currently our solution in not optimized and there is potential to reduce this thread creation overhead.

Power Consumption: We observed with usage of

multiple network sockets for receiving device instructions results in higher power consumption. Such power consumption weakens the case for cloud based platform virtualization. We believe two advances in our current design to counter this problem. Traditional power saving solutions for wireless networks are specific to access protocols [16] and require network support. Firstly we should leverage on the fact that mobile client interacts with only one server, Cloud Bridge. Thus unlike traditional solutions sleep coordination is possible between the two end hosts as they can coordinate a sleep schedule at application layer taking various factors like latency, available bandwidth etc. into consideration without requiring any network support. Secondly we should focus on reducing the number of network sockets used for sending/receiving the data/instructions for various sensor devices. Currently our design uses multiple sockets in absence of defined message format or communication protocol between the two hosts. We can consider defining the communication protocol to unify sending of sensor instruction and corresponding data over the same network socket. Such a solution will also reduced the overhead due to socket and thread creation at Cloud Bridge as discussed above.

Security & Policy: We aim to provide the user with the flexibility of concurrently using multiple mobile platforms, and thus it is important to ensure protection to platforms against the unsecured ones running malicious programs. As all the cloud platforms share the same set of physical HW devices, device operations may be vulnerable to security threats and eavesdropping. We must also ensure that the proposed security model do not require complex non-scalable changes, is flexible and easily manageable. Cloud Bridge provides an ideal vantage point for implementation of such security solutions. Also various device access policies similar to the ones proposed in Cells [7] can be easily implemented at Cloud Bridge. We developed a module called *policy* to which an instruction is passed before being enqueed for execution. Modular single point implementation allows flexibility in development and implementation of multiple security rules and access policies.

5. RELATED WORK

The idea of providing unlimited computational power over light weight user devices has been well explored. In context of mobile devices, researcher have been bitten by the fancy powers of cloud services. These days focus is on identification of computationally complex instructions and offloading them to cloud servers. Apart from enhancing computational prowess of mobile devices there has been various works focussing on in-device platform virtualization. It enables usage of multiple mobile platform over a single mobile device. Remaining portion of this section will discuss these releated works in greater details.

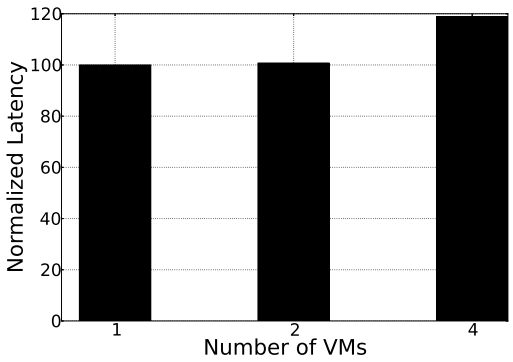

Figure 9: Normalized latency for the scaling experiment. The values are normalized wrt latency values for a single platform

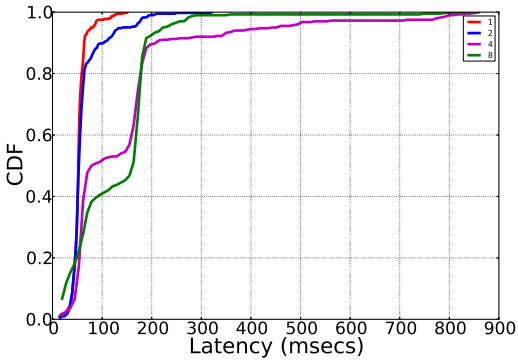

Figure 10: Distribution of latencies for the scaling experiment. We observe that for majority of latency values are beyond desired limit for scaling factor 8

Desktop Virtualization: The idea of using remote workstations and desktops is well known. Desktop virtualization, which moves computation to the data center, allows users to access their applications and data using stateless client devices. It ensures cheaper, secure and easily manageable desktop delivery at enterprise level [10]. Traditionally RDP [4] and VNC [6] are used for creating a remote desktop session, but they deliver poorly for interactive sessions. Recently services like Splashtop, LogMeIn, Google Chrome's RD [1, 5] etc. overcame the issue of interactive remote desktop sessions by deliverying remote screen as HD video streams. These desktop virtualization solutions support usage of input devices like mouse, keyboard etc., but dependence of mobile devices over various sensor devices poses various new challenges. It motivates development of dedicated thin client solutions with emphasis on availability of sensor devices.

Remote Executions: Remote execution of resource-intensive applications for resource-limited hardware is

also a well-known approach in mobile/pervasive computing. All remote execution work carefully designs and partitions applications between local and remote execution, and runs a simple visual, audio output routine at the mobile device and computation intensive jobs at a remote server [14, 18, 11, 12, 13]. Satyanarayanan [18] explored saving power via remote execution at locally available cloudlets. This remote execution of intense computations at remote servers is also called as computational cloud offloading and can be broadly classified in two categories, namely developer agnostic and vice versa. Developer agnostic offloading either involves complete process migration or a thread migration to a remote clone VM running in the cloud.

Such solutions have shown significant performance improvements for many computationally intense applications but face various limitations. Inability to migrate native states or unique native resources because of processor architecture differences makes such migrations relatively superficial. To reap maximum benefits from such remote executions resource synchronization is required which will require even tighter latency constraints. Also computations involving hardware devices are restricted for local execution. Though advances in remote execution has been promising, but it only solves the problem of enhancing computational prowess for mobile devices. These days researchers are concerned for both computational power and platform flexibility. Mobile platform virtualization solves this flexibility problems.

Mobile Platform Virtualization: VMware proposed its mobile platform virtualization solution called MVP [9], which targets to run a single enterprise virtual phone over the existing one. MVP has higher computational over resource limited mobile device and is targeted to run just one enterprise virtual phone over existing host personal phone. Cells [7] overcomes the limitation of MVP by ensuring minimal computational overhead and allowing multiple virtual phones to be used simultaneously. But it requires significant kernel modifications and expects all virtual phone applications to run over same mobile OS restricting platform flexibility. Cells [7] and MVP [9] solve the problem of mobile platform virtualization but they do not consider limited computational prowess of mobile devices. Recently Lu et al. proposed Virtualized Screen [17] which conceives the idea of delivering mobile platforms as a cloud service. This work is orthogonal to our contribution of sensor virtualization for such cloud platforms and combination of the two can provide a much better cloud based platform virtualization for mobile devices.

6. CONCLUSION

In this work we presented an argument to deliver mobile platforms as cloud services. We combine to ideas of mobile virtualization with remote execution and propose that mobile platforms can be virtualized over cloud servers. We identified that to enable such a service virtualization of multiple sensor devices is important and focussed to enable such a virtualization. We discussed various design choices for this sensor virtualization and proposed usage of CloudBridge to reduce complexity at mobile client. We discussed our implementations and related challenges. We tested our virtualized sensor devices for latency performances under various scenarios. We also analysed sources contributing to observed latencies and discussed improvements to enhance the performances. We also discussed various future works to enhance performance of our sensor devices. We believe our work is one of the first in this direction and should serve as the starting point to enable delivery of mobile platforms as a cloud service.

We have kept our source-code open and strongly believe that such a research can progress with input from open source community. The source code can be accessed from our Github repo [3].

7. REFERENCES

[1] Gaikai: www.gaikai.com.

[2] Kernel Docs. https://www.kernel.org/doc/.

[3] mPaaS git repo. https://github.com/agupta13/mPaaS.

[4] Remote Desktop Protocol (Windows): http://msdn.microsoft.com/en-us/library/windows/desktop/aa383015(v=vs.85).aspx.

[5] SplashTop: www.splashtop.com.

[6] VNC: http://www.realvnc.com/.

[7] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: a virtual mobile smartphone architecture. In *ACM SOSP'11*.

[8] G. Armitage. An experimental estimation of latency sensitivity in multiplayer quake 3. In *ICON 2003*, 2003.

[9] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The vmware mobile virtualization platform: Is that a hypervisor in your pocket? *SIGOPS Oper. Syst. Rev.*

[10] K. Beaty, A. Kochut, and H. Shaikh. Desktop to cloud transf. planning. In *IPDPS '09*.

[11] N. Bila, E. de Lara, K. Joshi, H. A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan. Jettison: efficient idle desktop consolidation with partial vm migration. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 211–224, New York, NY, USA, 2012. ACM.

[12] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *EuroSys '11*.

[13] B.-G. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 8–8, Berkeley, CA, USA, 2009. USENIX Association.

[14] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *MobiSys '10*.

[15] A. Gupta, J. Min, and I. Rhee. Wifox: scaling wifi performance for large audience environments. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 217–228, New York, NY, USA, 2012. ACM.

[16] J. Liu and L. Zhong. Micro power management of active 802.11 interfaces. MobiSys '08.

[17] Y. Lu, S. Li, and H. Shen. Virtualized screen: A third element for cloud-mobile convergence. *MultiMedia, IEEE*, 18(2), feb. 2011.

[18] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct. 2009.