Enabling the Virtual Phones to remotely sense the Real Phones in real-time

~ a Sensor Emulation initiative for virtualized Android-x86 ~

Researcher ~ Raghavan Santhanam - Columbia University, NYC ~ rs3294@columbia.edu
Technical Point Of Contact ~ Songchun Fan - Duke University, NC ~ schfan@cs.duke.edu
Research Advisor ~ Prof. Jason Nieh - Columbia University, NYC ~ nieh@cs.columbia.edu
In Association With ~ Prof. Li Erran Li - Columbia University, NYC ~ lierranli@cs.columbia.edu

---

1. Introduction

Being in 2013, the smartphone industry has almost captivated us humans with its astonishing capabilities. The capabilities that were once seen only as science fiction are now realities. The capabilities include but not limited to sensing heat of a thermal body, atmospheric pressure in a submarine, change in ambience for an auto-brightness light, speed of a vehicle, spatial orientation of a stationary object, gravity's influence on an object in space, magnetic radiation in a mining area, geographical position in remote parts of our earth, and a lot more. In recent times, these capabilities have become an inseparable part of our lives due to their limitless usefulness in our day-to-day activities.

Smartphones' capabilities and the ability to control them remotely, when these two bond together as a technical possibility, the impact is huge. The impact is so huge that it adds a whole new dimension to the endless use cases of the bonding. Remote sensing is the new dimension. Evidently, remote sensing has numerous applications in the field of medicine, defense, cosmology, to name a few. Remote sensing becomes predominant when it's real-time. Real-time updates are crucial. Crucial as it gets to sense alarming situations such as life-threatening shock waves in the form of vibrations, high-intensity magnetic fields, etc.

Accomplishing such a task of remote sensing in real-time requires a considerable amount of work. Reason being, the software shipped with these smartphones will not be usually designed for the purpose of remote-sensing at all, but only for localized usage by an ordinary cellphone user. The considerable work will be in the form of including features into these smartphones so that they can be remotely monitored by a machine running software that's compatible with that of those smartphones. The included features must be efficient, generic, accurate, fast, and robust - a challenge that the current work of Sensor Emulation has accomplished.

2. Motivation and Use Cases

Sensor Emulation is a work of emulating the important sensors on a real Android device in a virtualized Android-x86 environment. The motivation has been in terms of its immense usefulness in the area of remote-sensing Android smartphones and the fact that nobody else has accomplished this in a portable, comprehensive, consistent, efficient, remote, and real-time manner. Current work addresses the portability by deploying the Sensor Emulation in the Hardware Abstraction Layer(HAL) of Android which is driver-agnostic, the comprehensiveness by emulating a total ten Android sensors, consistency by having lossless exchange of sensor readings, efficiency by having very minimal overhead in the overall emulation, remoteness by having wireless interaction over the IP network, and real-timeness by providing the sensor readings instantaneously and remotely.

Use cases of the Sensor Emulations are diverse. The sheer diversity is dominated by the real-time nature of the sensor readings provided by the emulated sensors. Below are the notable ones.

1. Monitoring patients' body temperature, etc.
2. Broadcasting alert messages when magnetic field threshold is crossed.
3. Triggering solar panels to open when there is ample solar light.
4. Tracking vehicles' speed.
5. Taking necessary actions when the tolerable proximity is surpassed.
6. Testing of sensor-based applications in a virtualized environment.
7. Validating sensor-based applications in a virtualized environment with no restriction on the Android OS version in use. Just one real Android device with any Android OS release is needed.
8. Controlling many virtual Android devices with one real Android device.

All of the above use cases are accomplishable in a remote manner in real-time. The last use case is actually an extended use case of the present work.

3. Design Comparisons

The sensor emulation is designed at system level making it a system-level Sensor Emulation. This system-level Sensor Emulation is being done at Hardware Abstraction Layer(HAL) illustrated in below **Fig. 1**(next page) and hence it's generic and portable as opposed device-driver level Sensor Emulation which many other research papers focus on, given that the device driver is device-dependent as well as its vendor with respect to its specifications, etc. The HAL based implementation logic holds good regardless of any newer releases of Android that can happen in the coming years and also it's regardless of the changes that happen below the HAL like

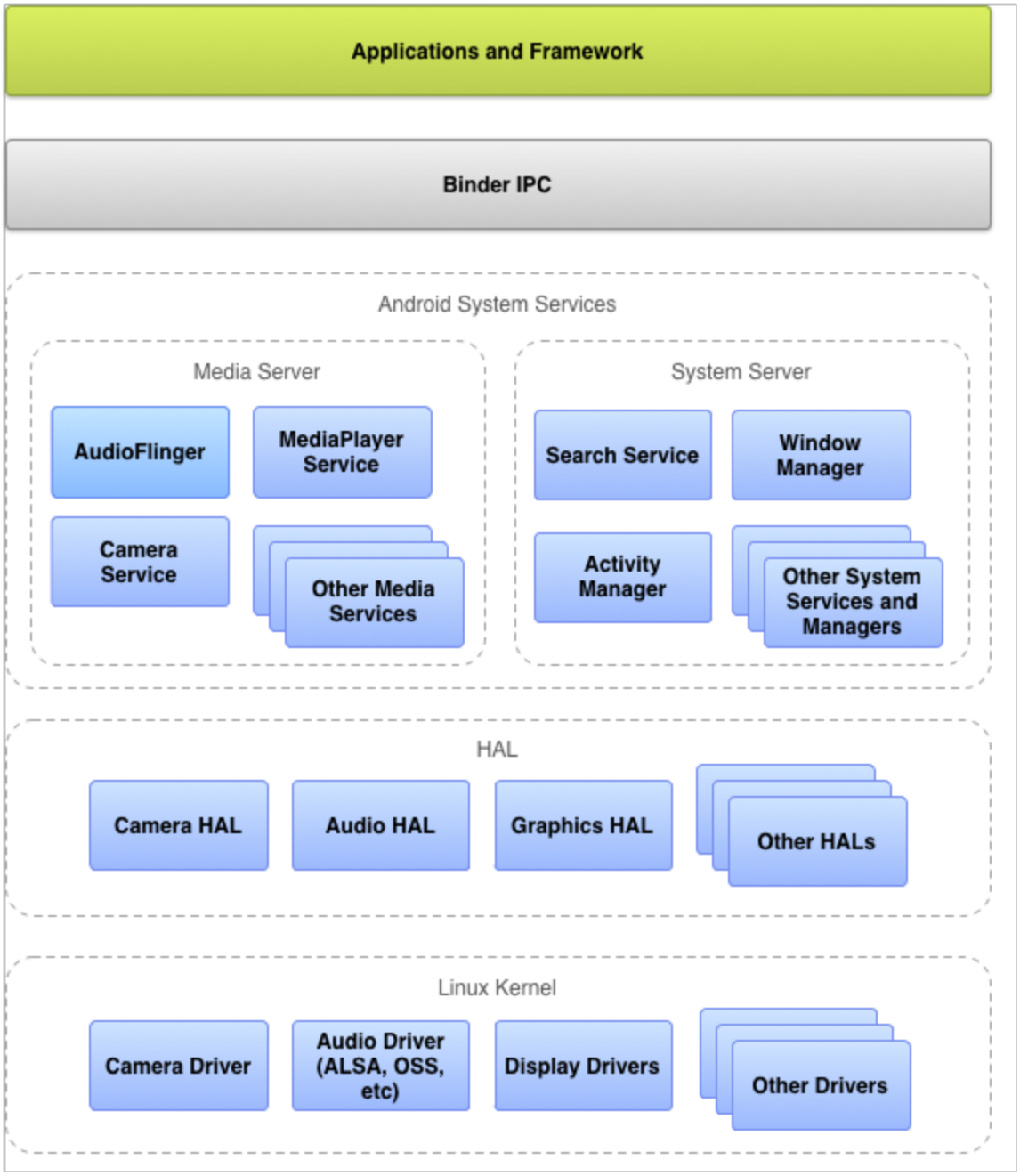


Fig. 1 Android Low-Level System Architecture
(Source: http://source.android.com/devices/images/system-architecture.png)

in the device-driver level, etc. This is because, the HAL is the abstract representation of the underlying hardware as opposed to the detailed and tight representation. Hence, the abstraction shall and will never break, and so is the Sensor Emulation work done at sensors HAL.

The system-level Sensor "Emulation" is way better than any application-level Sensor "Simulation", as the latter will always be running as a stand-alone application and the other sensor-based applications need to be modified to interact with this stand-alone application to get the "fake" sensor readings being generated artificially based on some logic such as input devices' data, etc. In addition, if the real device sensor readings are needed, then again this stand-alone application needs to be implemented such that it gets from the underlying abstraction layers(JNI, HAL etc) which affects the time taken in getting the real device sensor readings. And then the same stand-alone application needs to be modified for the emulated Android-x86, if the real device sensor readings are needed. Even then, the sensor-based applications running in the emulated Android-x86 need to be modified to interact with the stand-alone application running in the same emulated Android-x86 to get the real device sensor readings.

With the HAL based Sensor Emulation, there is no real burden on the application developers. The exact applications they develop for the real Android device equipped with real device sensors run with absolutely no problem, as far as the respective sensor's behavior is concerned.

Thus, the system-level Sensor Emulation at HAL is proven to be efficient, portable, and universal to all the Android systems and not only just the x86 based ones.

4. Design Considerations

The design considered comprises of three main components.

1. The device server on the real Android device.
2. The userspace "C" program in the host.
3. The emulator server on the virtual Android device.

Socket-communication based interactions between these three components facilitate the necessary Sensor Emulation on the virtualized Android-x86. All these socket-communication are designed to be part of the Sensor HAL module on both the real and virtual Android devices, in addition to the client-server logic implemented in the userspace "C" program targeted for the host running the virtualized Android-x86 on Qemu.

Inside the sensor HAL module, the reads and writes of the sensor readings are designed to be efficient, fast, reliable and real-time. In the device sensor HAL module, the each of the existing sensor event loop is modified to notify the respective sensor readings to the real device server running as part of the same device sensor HAL module via one of the inter-thread communication mechanisms, pipes. The real device server is designed to be as efficient as possible and robust at the same time.

The efficiency is in terms of the absence of synchronous polling for change in the device sensor readings, but asynchronous as pipes are being used for sharing sequentialized sensor data. Robustness is in terms of being fault-tolerant to the failures in socket-communication errors. Should there be any socket communication errors, the respective socket communication logic is designed so as to reset the socket-server to a sane state saved previously, which is usually the entry point of the corresponding sensor server thread.

The device server running as part of the device's sensor HAL module is designed to be dedicated to a single sensor as opposed to one server for all the sensors. This design choice ensures that each healthy sensor server remains unaffected by any other error-prone sensor servers. This also implies that the device sensor server for a sensor runs on demand only when that sensor is enabled and active. In addition, the dedicated sensor servers rule out the very need to parse every incoming network buffer/packet to detect which sensor's reading it is which would be the case when a single sensor server is sending the aggregated sensors readings compromising on the number of sensor device server threads in the device sensor HAL module with an intent of reducing the number of socket TCP/IP connections. This dedication also simplifies the entire design to a greater extent as it doesn't necessarily mandate the event loop of each of the sensor to be filling the same shared buffer in a synchronized manner with the respective sensor readings, given that each sensor is implemented as a separate hardware module. The sheer amount of synchronization overhead involved in such a scenario prevents the sensor readings to be sent and eventually available on the emulated Android-x86, in real-time at high-speed. Hence, the design choice of the dedicated servers for each of the sensors is proven to be efficient and robust as claimed.

The userspace "C" program is designed to be having a dummy server which acts as a gateway to the emulator server. There are dedicated dummy server for each of the emulator sensor servers. These dummy servers are made to accept connection with respect to a mapped port from the host to the guest(Android-x86) and then put to forever-sleep. So, when the client-to-device or client-to-remote-server wants to send the received sensor readings to the emulator server, it sends those readings to the respective dummy server for the corresponding sensor at a particular port. Due to the port-mapping/forwarding mechanism, the same reading gets forwarded from the host to guest and thus it reaches the respective emulator sensor server listening to the mapped port. This design considered over explicit send-receive between the userspace "C" program and the emulator server helps in reducing the overhead of making explicit send/recv socket-communication based calls. And also, this rules out any need for the userspace "C" program and the emulator servers to know each others' ip-address and also the port to be used for the socket-based communication. Thus making the interaction between the userspace "C" program and the emulator server generic.

To account for the high-frequency real Android sensors, Gyroscope and Accelerometer, in that order, as opposed to the relatively low-frequency ones which are the remaining three real Android sensors(magnetic, light and proximity), their emulator server code are designed to be customized as per the rate at which the gyroscope and accelerometer sensor readings arrive from the real device to the emulator. The customizations are in terms of the amount of the readings received at a time and the sleep delay for these emulator servers. However, still, the gyroscope and accelerometer real device servers and the userspace "C" programs are designed to send the respective sensor readings one at a time, as they are not the points of bottlenecks but the corresponding emulator sensor servers. The customization is necessary since the gyroscope and accelerometer sensor readings get queued up in the network buffer even when they are being sent one at time by their respective real device sensor servers when their emulator servers are slow in receiving the readings one at a time. The slowness would be due to the inherent overhead of the network-based communication between the host and the emulator, the relatively slow the sensor server loop due to the extra checks on the network-state, packet received and so on. This slowness contributes to the slow Sensor Emulation for Gyroscope and Accelerometer and hence affects the sensor-initiated responsiveness of the sensor-based Android applications running on the emulator.

As a consequence, the gyroscope emulator server is designed to be receiving a bunch of pre-defined number of readings and so is the accelerometer emulator server. But, there is a difference in the number of readings received by these servers and also the interval at which these readings are received. The interval is short for gyroscope when compared to accelerometer, since gyroscope is too sensitive and faster than accelerometer on the real Android device. Apart from this design customization for these two sensors, the way in which the bunch of readings received is processed is also customized and tuned for only these two sensors as opposed to other three real sensors. Since, the periodically polled sensor event data loop function is designed to process only one reading at a time for speed and responsiveness, there is an intermediate buffer in terms of pipe(inter-thread communication mechanism) dedicated to these two sensors separately. One for the accelerometer and another for gyroscope. As a result, these bunch of readings are written by the respective sensor servers onto the specific sensor pipes(accelerometer-pipe and gyroscope-pipe) in a one go and the periodically polled sensor event loop function reads off one reading at a time from the respective pipe and returns it as the corresponding sensor's data to the needed sensor subsystem. This receive-many-process-one design consideration has a huge speed benefit over receive-one-process-one which became apparent during the evaluation.

The entire sensor emulation code is designed to be comprehensively and thoroughly logged. So, any developer will find the code to be friendly while debugging, if and

when the need be. These logs are macro enabled and hence can be enabled or disabled as per the need. Each sensor has separate logs to ease the debugging. All the logs are designed to be file-based and to be in separate files to let the developers follow each sensor behavior easily. The standard built-in Android logging technique is not used since the logcat output is not persistent across the booting of the system unless redirected to a file or something similar, especially when there is a crash that brings down the entire Android system triggering a reboot. So, again this ends up indirectly in a file-based logging mechanism which is what the current file-based logging does in the first place.

The design decision to do the Sensor Emulation work at HAL level provides an edge due to the fact that it's closer to the hardware on the real Android device as opposed to the same scenario with application-level Sensor Emulation. The direct and most important consequence of this is the entire Sensor Emulation at HAL needing to be done in C/C++ which again has an obvious edge over any application-level Sensor Emulation in terms of performance, which will be written in Java and hence the overhead of reaching down all the way from the application layer to the HAL through other intermediate layers to get the real device readings. This overhead can be negligible in general, but not when even nanoseconds of difference is important. This definitely is important when claiming the Sensor Emulation to be real-time, which is indeed one of the main highlights of the current system-level Sensor Emulation.

5. Software Architecture

5.1 Brief description of Android Sensors(Source: online documentation)

AndroidSensors give applications access to a mobile device's underlying base sensor(s): accelerometer, gyroscope, and magnetometer. Manufacturers develop the drivers that define additional composite sensor types from those base sensors. For instance, Android offers both calibrated and uncalibrated gyroscopes, a geomagnetic rotation vector and a game rotation vector. This variety gives developers some flexibility in tuning applications for battery life optimization and accuracy. The SensorsHardwareAbstractionLayer(HAL)API is the interface between the hardware drivers and the Android framework; the SensorsSoftwareDevelopmentKit(SDK)API is the interface between the Android framework and the Java applications. Audio recorders, Global Positioning System (GPS) devices, and accessory (pluggable) sensors are not supported by the Android Sensors HAL API described here. This API covers sensors that are physically part of the device only. ***Application framework -*** At the application framework level is the app code, which utilizes the android.hardware APIs to interact with the sensors hardware. Internally, this code calls corresponding JNI glue classes to access the native code that interacts with the sensors hardware.

JNI **-** The JNI code associated with android.hardware is located in the frameworks/base/core/jni/ directory. This code calls the lower level native code to obtain access to the sensor hardware. ***Native framework*** **-** The native framework is defined in frameworks/native/ and provides a native equivalent to the android.hardware package. The native framework calls the Binder IPC proxies to obtain access to sensor-specific services. ***Binder IPC*** **-** The Binder IPC proxies facilitate communication over process boundaries. ***HAL*** **-** The Hardware Abstraction Layer (HAL) defines the standard interface that sensor services call into and that you must implement to have your sensor hardware function correctly. The sensor HAL interfaces are located in hardware/libhardware/include/hardware. See sensors.h for additional details. ***Kernel Driver*** **-** The sensors driver interacts with the hardware and your implementation of the HAL. The HAL is driver-agnostic.

Every piece of hardware for an Android device needs to have a standard Android hardware module interface to be defined and implemented with the default methods(functions) for controlling it. A typical HAL module has the below interface as it's taken from

<android_src_path>/hardware/libhardware/include/hardware/sensors.h

```
/**
* Every hardware module must have a data structure named
* HAL_MODULE_INFO_SYM and the fields of this data structure must
* begin with hw_module_t followed by module specific information.
*/
typedef struct hw_module_t {
     /** tag must be initialized to HARDWARE_MODULE_TAG */
     uint32_t tag;

     /**
     * The API version of the implemented module. The module owner is
     * responsible for updating the version when a module interface
     * has changed.
     *
     * The derived modules such as gralloc and audio own and manage
     * this field. The module user must interpret the version field
     * to decide whether or not to inter-operate with the supplied
     * module implementation. For example, SurfaceFlinger is
     * responsible for making sure that it knows how to manage
     * different versions of the gralloc-module API, and AudioFlinger
     * must know how to do the same for audio-module API.
     *
     * The module API version should include a major and a minor
     * component. For example, version 1.0 could be represented as
```

```
     * 0x0100. This format implies that versions 0x0100-0x01ff are
     * all API-compatible.
     *
     * In the future, libhardware will expose a
     * hw_get_module_version() (or equivalent) function that will
     * take minimum/maximum supported versions as arguments and would
     * be able to reject modules with versions outside of the
     * supplied range.
     */
     uint16_t module_api_version;
#define version_major module_api_version
     /**
     * version_major/version_minor defines are supplied here for
     * temporary source code compatibility. They will be removed in
     * the next version.
     * ALL clients must convert to the new version format.
     */

     /**
     * The API version of the HAL module interface. This is meant to
     * version the hw_module_t, hw_module_methods_t, and hw_device_t
     * structures and definitions.
     *
     * The HAL interface owns this field. Module
     * users/implementations must NOT rely on this value for version
     * information.
     *
     * Presently, 0 is the only valid value.
     */
     uint16_t hal_api_version;
#define version_minor hal_api_version

     /** Identifier of module */
     const char *id;

     /** Name of this module */
     const char *name;

     /** Author/owner/implementor of the module */
     const char *author;

     /** Modules methods */
     struct hw_module_methods_t* methods;
```

```
        /** module's dso */
        void* dso;

        /** padding to 128 bytes, reserved for future use */
        uint32_t reserved[32-7];

} hw_module_t;
```

5.2 Current Work

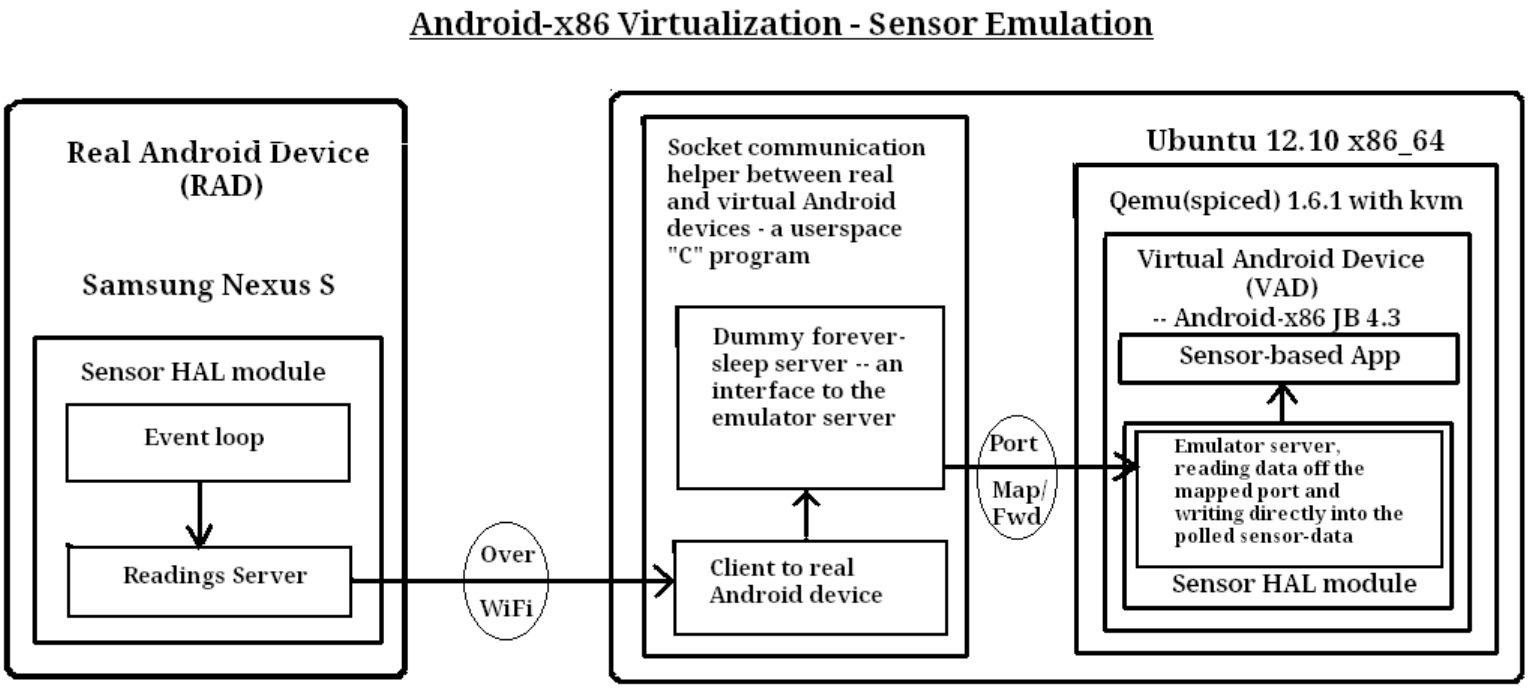


Fig. 2 Sensor Emulation Software Architecture

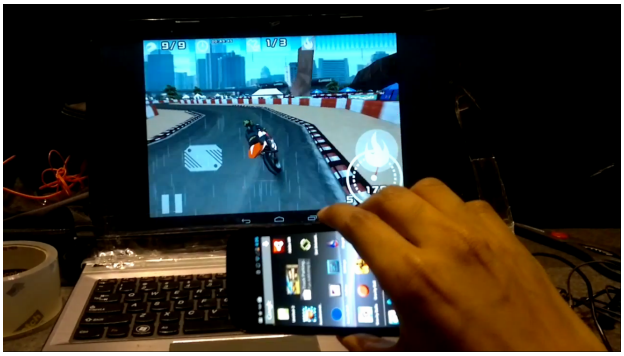
Note: The above architecture remains same in the case of a remote-server in place of RAD except that the remote-server generates the artificial sensor readings for each of the sensor being queried for the emulation. The method used to generate artificial readings is a simple one based on random number generation, as of now.

As seen in the below Fig. 2, the work is accomplished by including socket-communication servers and clients in the both real and virtual Android devices wherein the real Android device sends the readings and the virtual Android device receives them, over the network. This socket-communication happens at the Hardware Abstraction Layer(HAL) module for Sensors in both the real and virtual Android devices. The initialization method of the sensors in sensors HAL module are modified to spawn the socket-communication servers on the real and virtual Android devices and will be ready to send/receive the sensor readings once the respective sensor is enabled and is made active by loading the sensor HAL module.

To facilitate the send/receive of sensor readings in a seamless manner between the real and virtual Android devices, there is a userspace "C" program which gets the readings from the real device and sends it to the virtual device. This program has a client-to-real-device to receive the real device sensor readings and a dummy server listen to port that's mapped from the host to guest. Upon receiving the real device readings the client-to-real-device sends it to the dummy server which will do nothing as it will be sleeping all the time after accepting an incoming connection. However, since the port under used is mapped from the host to guest, these readings reach the emulator server running as part of the sensor HAL module of the emulated Android-x86. Following which, these readings are written into the sensor event data that is polled at regular intervals in the same emulated Android-x86's sensor HAL module. Then, these real device readings are available to any sensor-based app running in the same emulated Android-x86 environment via the layers which are on top of HAL including the topmost one which is the application layer.

6. Demo

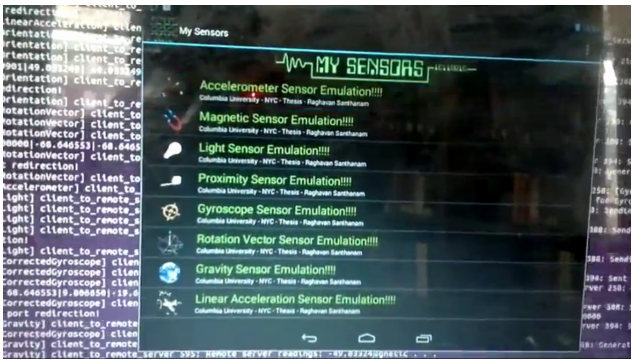
Paired-real-device-scenario - Sensor Emulation using the real device sensor readings received over the network(WiFi).



Snapshot 1 - Emulated Accelerometer in action

Snapshot 2 - Emulated Gyroscope in action

Remote-server-scenario - Sensor Emulation using artificially generated sensor readings fetched from a remote server, supposedly running anywhere as long as it's ip-address is known and public.



Snapshot 1 - All 10 sensors in action - remote server scenario

7. Implementation

The HAL module-specific implementation is split based on the real Android sensors and the virtual Android sensors resulting from the Android Sensor Fusion technique. All of the sensors that are present in the real Android device, Samsung Nexus S are being emulated. They are as under.

Real Sensors - 1) Accelerometer 2) Magnetometer 3) Light 4) Proximity 5) Gyroscope
Virtual Sensors(constructed based on real ones using Android Sensor Fusion method) - 6) Linear Acceleration 7) Orientation 8) Gravity 9) Rotation Vector 10) Corrected Gyroscope.

For both the device and the emulated Android-x86, the source for the real sensors are present under **<android_src_path>/hardware/libsensors.** The source for the virtual sensors are under **<android_src_path>/frameworks/service/sensorservice.** The exact source paths are as under for both the device and the emulated Android-x86: **Real sensors(all under <android_src_path>/hardware/libsensors/) - AkmSensor.cpp, GyroSensor.cpp, InputEventReader.cpp, LightSensor.cpp, ProximitySensor.cpp. Virtual sensors(all under <android_src_path>/frameworks/service/sensorservice/) - LinearAccelerationSensor.cpp, RotationVectorSensor.cpp, SensorFusion.cpp, SensorService.cpp, CorrectedGyroSensor.cpp, GravitySensor.cpp, OrientationSensor.cpp**

The real sensors' code is deployed on RAD as a single library, **sensors.herring.so** under **/system/lib/hw/** while as **sensors.emu.so** on VAD under **/system/lib/hw/**. The virtual sensors' code is deployed on RAD and VAD as a single

library, **libsensorservice.so**. under **/system/lib/**. These libraries are built as according to the target platforms from more or less similar sources since only one of them will be acting as a server and the other acting as a server which receives readings instead of sending any, from the mapped port. The only obvious difference between the sources for the device and the emulated Android-x86 is that the device side of the Android sources implement servers providing the device sensor readings and the emulated Android-x86 side of Android sources also implement a server but receives the device sensor readings via the mapped port. The emulated Android-x86 side must be a server against the usual notion of server only sending data and not receiving any, because it has to **listen** to a mapped port. The userspace "C" program has two sources: **SensorEmulationClientServer.c**, **SensorEmulationRemoteServer.c**. From here on, unless and otherwise mentioned, the phrase "The userspace "C" program" refers to **SensorEmulationClientServer.c** specific to the client connecting to a real Android device. **SensorEmulationRemoteServer.c** will be referred at the end. For connecting to a RAD, **SensorEmulationClientServer.c** needs to be compiled with **DEVICE_READINGS** defined as a pre-processor macro and for connecting to a remote server implemented by **SensorEmulationRemoteServer.c**, **SensorEmulationClientServer.c** needs to be compiled with **REMOTE_SERVER_READINGS** as a pre-processor macro.

The userspace "C" program intended to be as an intermediary between the device and the emulated Android-x86, is implemented with as many clients as the real device sensors, which connect to the respective device readings server dedicated to a particular sensor(real or virtual). These clients to real device will have their server counterparts which are the dummy forever-sleep servers whose duty is to accept any one incoming connection(from its client counterpart) and go for an infinite sleep, specifically, waiting on a semaphore which never gets signalled. The dummy forever-sleep server concept is crucial in reducing the overall time consumption. The reduction is due to the lack of the need for any explicit receive-send for each of the device readings got, from the host to the guest. All that is being done is that the connected client writes to a connected socket of the dummy forever-sleep server which is in fact listening to a port that is mapped onto the same numbered port inside the emulated Android-x86. The device's ip-address is the only thing need to be known in advance for the user-space "C" program for the sake of its client-to-device. The ip-address will be read from a pre-determined file expected to be in the current directory of the userspace "C" program(process). The syntax is just plain one-liner having the ip-address in its usual layout which is xxx.xxx.xxx.xxx terminated by a new-line. The port-mapping facility employed also rules out any know-in-advance ip-addresses between the host and guest - the host and the guest neither need to be aware of each others' ip-address nor the ports being listened to. The servers on the host listen to the host ports and the servers on the guest to the guest ports - purely based on the **localhost**.

As of now, the entire concept of Sensor Emulation assumes that each sensor's servers and clients use only pre-determined ports assigned to them uniquely. This ensures that there is no intermix of the sensor readings and hence the sensor data transmission is streamlined end-to-end. The assigned ports are as under. **Accelerometer:5000, Magnetometer:5001, Light:5002, Proximity:5003, Gyroscope:5004, Orientation:5005, Corrected Gyroscope:5006, Gravity:5007, Linear Acceleration:5008, Rotation Vector:5009.** However, if one-server-many-clients needs to be implemented, then the hard-coded port numbers need to changed just by offsetting by 10(as there are 10 sensors being emulated) and letting know all the dependent client/servers/dummy-servers the exact port to be used for the socket-communication. Thus, the above idea is still scaleable to any number of clients. The delay chosen in each of the servers, clients, event loop everywhere is such that the underlying system's resource utilization is nominal and the sensor readings retain their real-time nature, end-to-end, RAD-to-VAD. All the servers and clients are implemented as **pthread**s of the respective process on the particular system(RAD, VAD, or Ubuntu). As far as the readings are concerned, they are sent from the device to the emulated Android-x86 as a buffer of a pre-determined maximum length and is a "C" string with a specific pattern for each of the sensors. For example, accelerometer will have a x-y-z triplet in its readings. So, it's reading will be transmitted in a pattern that looks like "0.000000|0.000000|0.000000", wherein the pipe(|) symbol serves as the delimiter for the individual component of the triplet. So, there is no fixed length for each of the component but the entire pattern has a maximum limit which is the pre-determined maximum length. For the maximum genericness, the userspace "C" program doesn't do anything with the sensors readings buffer of a particular pattern received from the device. Instead, it just sends it to the emulator via the dummy server based on the mapped port. The emulator server for the specific sensors will be responsible to interpret the readings correctly by parsing the buffer of a particular pattern. This very don't-touch-just-send nature of the userspace "C" program makes it very generic and portable - for any new sensor being emulated, all that needs to be done is to increase the number of sensors defined in it and update the sensors' names list. Rest of the work will be done on the device-side code as well as the emulator-side code.

For majority of the sensors, i.e., 8 out 10 sensors being emulated, the send-one-receive-one for the sensor readings are applicable. The only two sensors which are having servers that send one at a time on the real device and receive a bunch in the emulator are the gyroscope and accelerometer. The send-one-receive-one greatly simplifies the overall design for the rest of the 8 sensors as there is no separate processing criteria for each of the single sensor readings received apart from the parsing the pipe-delimited sensor readings pattern. However, gyroscope and accelerometer demanded send-one-receive-many as these

two sensors have higher sampling frequencies than the other eight sensors. Hence, the device sensor servers for these two sensors send the respective sensor readings at a higher rate than the emulator servers can process. As a consequence, **`sensors_emu.c`** implements the device sensor servers for these two sensors with receive many logic so that the sensor readings sent at a high rate even though one at a time, don't spend much time in the network buffer resulting in slow responsive emulated sensors. The implementation involves writing the bunch of readings received onto a shared pipe by the respective emulator server and the periodical polling function(**`dummy_poll()`**) reads the sensor readings off the shared pipe, one reading at a time to keep the polling function simple and quick. Each of the two sensors, gyroscope and accelerometer have their own shared pipes(**`gyro_pipe`** and **`accel_pipe`**, respectively) for these inter-thread communication of the sensor readings. To make sure that these don't affect the real-time nature of the emulation(by blocking the servers and/or the polling function when pipe is full for server to write or empty for the polling function to read), both **`gyro_pipe`** and **`accel_pipe`** are made non-blocking(O_NONBLOCK) using explicit **`fcntl()`** calls after piping them with **`pipe()`** as **`pipe2()`** isn't available in Android.

The usage of TCP over UDP for the exchange of readings end-to-end does ensure consistency and absolutely no loss of the readings as far as the network connectivity staying in tact. The only scenario wherein there can be loss of readings is that within the device itself, the pipe buffer which is written into by the respective device sensor event loop might get full causing some of the readings not being transmitted at all from the device itself over the network towards any connected client - an unfortunate scenario beyond one's control. If there are any loss of sensor readings within the device itself which prevents from transmitting the readings over the network to the emulator, then that's an unfortunate incident and the resolution for such an incident is beyond the scope of the current work of Sensor Emulation.

For the remote-server scenario, the implementation is a straightforward one. The program **`SensorEmulationRemoteServer.c`** is run on the host and generates artificial sensor readings using random number generation method, and sending it to any connected client. This remote server could be anywhere and can be replaced any other server as well, as long as the pre-determined sensors reading pattern is followed while sending the respective sensor readings to any connected client. And the client running as part of the **`SensorEmulationClientServer.c`** compiled with pre-processor macro **`REMOTE_SERVER_READINGS`** will connect to the respective remote-server instance running as part of **`SensorEmulationRemoteServer.c`** based on the specific pre-determined sensor ports. And rest of the operations once the artificial readings are received remain the same as that's in the case of RAD specific real device sensor readings explained earlier, end-to-end.

8. Evaluation

The evaluation involved timestamping at the real device end and the emulator server ends. Before finalizing the correct time measuring API, APIs were experimented with were, **`gettimeofday()`** and **`clock_gettime(CLOCK_MONOTONIC, . . .)`**. **`gettimeofday()`** was decided not to be used as it's marked deprecated. **`clock_gettime(CLOCK_MONOTONIC, . . .)`** is not used since it's based on the time elapsed since unspecified time in the past which can be different on the real device and on the emulator. So, the next reliable option was **`clock_gettime(CLOCK_REALTIME, . . .)`** and has been used for the evaluation. The obvious fluctuations in the measuring **`CLOCK_REALTIME`** is ignored at the cost of simplicity of logging the timestamps, as the **`CLOCK_REALTIME`** is the only available comparable time quantity between the real device and the emulator. To have the fluctuations compensated, a time-syncing application, ClockSync was installed on both the real device and the emulator to update the current time at any point time based on the internet based NTP server.

Based on the prolonged observation of three months of time and data collection, it has been noted that there's absolutely no loss of sensor readings being sent from the real-device to the emulator as long as the network connectivity stays in tact. And anything beyond, is out of the Sensor Emulation scope and hence can't be dealt with, in a reasonable way. This observation involved running 100+ top Android applications ranging from the basic sensor test applications that show the raw sensor readings, to the advanced 3D motion-sensor based Android games which are compatible with the emulator.

The sensor based applications used in the evaluation are : My Sensors, Championship Motorbikes 2013, Light-O-meter, Micro Metal Detector, ExionFly 3D HD, Proximity Launcher, Night Compass, Master Spirit Level, Android Sensor Box, Sensor Test and others. The number of sensor events for each of these sensors differ based on how many readings was generated/sensed by the real device sensor in a span of approximately 10 minutes of evaluation. With that, Accelerometer and Gyroscope readings were recorded for just above 1000 sensor events and all other sensors' readings having been recorded for approximately 100 sensor events.

Fig. 3 depicts the graphical comparison of the time delays applicable for the sensor readings in reaching the emulator from the real Android device. And fig. 4 depicts the numerical comparison of the time delays applicable for the sensor readings in reaching the Virtual Android Device(VAD) from the Real Android Device(RAD).

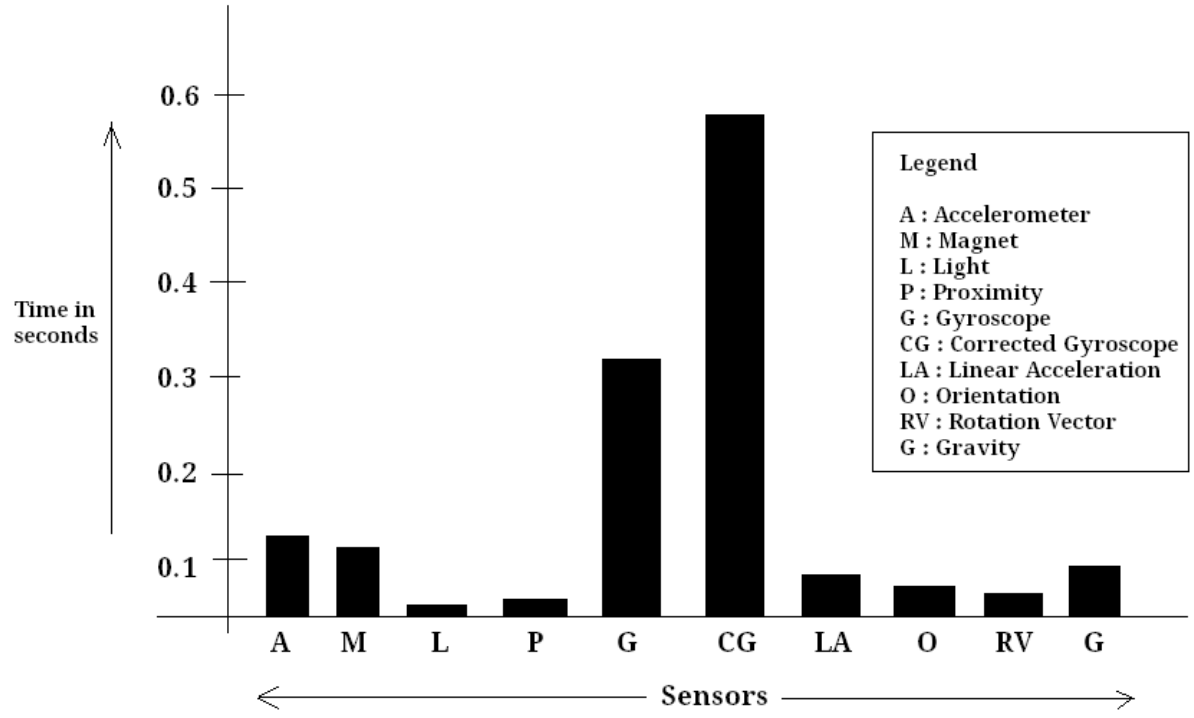


Fig. 3 Pictorial comparison of the average time delays applicable for the sensor readings in reaching the Virtual Android Device(VAD) from the Real Android Device(RAD)

Statistics → Sensors ↓	**Lowest delay(in nanoseconds)**	**Highest delay(in nanoseconds)**	**Average delay(in nanoseconds)**	**Number of readings missed during the transmission from the real device to the emulator**	**Number of readings with delays above the delay threshold of 4 seconds**
Accelerometer	2954496	432915456	123727873	0	0
Magnetic	2304	843893248	111025584	0	0
Light	134656	91592704	9595756	0	0
Proximity	4751616	81309440	12776519	0	0

Gyroscope	23935744	1375095808	327739803	14	0
Corrected Gyroscope	5789696	1124702464	592972183	0	0
Linear Acceleration -n	20992	624702208	69012239	0	0
Orientation	8448	622296832	54785776	0	0
Rotation Vector	112384	347361792	46375939	0	0
Gravity	8704	727843328	96040132	0	0

Fig. 4 Numerical comparison of the average time delays applicable for the sensor readings in reaching the Virtual Android Device(VAD) from the Real Android Device(RAD)

Note: 1) The 14 missing readings reported for the gyroscope are not really missing but they were at the very end of the transmission of readings from the RAD to VAD and hence failed to reach the VAD and hence were not logged and thus are reported to be missing. 2) The sensor readings of each sensor are collected with all other sensors enabled at the same time, except for the gyroscope and accelerometer whose readings were collected with only two of them enabled at the same time to measure them accurately considering their high-frequencies compared to other sensors.

9. Challenges

The real challenge was in getting the RAD sensor readings onto VAD in real-time without any loss over the network. It involved experimenting with many inter-process and inter-thread communication mechanisms as under. Inter-process: 1) File synchronization done in a mutually exclusive manner between two programs(server and client on the host) to communicate sensor readings from one to another. Inter-thread(after merging client and server programs into one and running as threads): 1) Shared-memory access guarded by semaphores. 2) Shared-memory access guarded by futex. 3) Shared-memory access guarded by pipe.

With respect to the device, the semaphore based notification mechanism employed for letting know the device server that a new sensor reading(more specifically, an individual component of the respective sensor readings which could be triplet, etc), proved to be inefficient and was a hindrance to the real-time nature expected in the Sensor Emulation process. Futex were thought of to be used as they are claimed to be faster user-space mutexes than the generic kernel-based mutexes and semaphores.

But, then the pipe usage proved to be the just right choice in terms of the sequentialization of the sensor data for a classic producer(server)-consumer(client) problem. Pipes helped to retain the real-time nature of the readings end-to-end.

Before all of these, the whole Sensor Emulation was intended to be done at the device driver level. After deep investigation of all the possible alternatives, ease, and portability of the needed Sensor Emulation technique, the HAL specific implementation was finalized. The investigation was an exhaustive one involving the examination of the couple of kernel-level device driver code namely, **<android_src_path>/kernel/drivers/input/misc/adxl34x.c** and **<android_src_path>/ kernel/drivers/staging/iio/accel/lis3l02dq.c**.

Though the problem of consistency and synchronization got solved, there was a problem with the Qemu emulator that was being used. It was crashing when the readings were sent at high-speed to cater for the real-time nature. The reason became apparent after a detailed analysis of the Qemu logs and searching for the specific error messages in the internet discussion forums. There was a problem in the version of the Qemu(v1.2) in the way of handling the incoming packets in **slirp/slirp.c** of Qemu source. The bug causing the problem was fixed later, and using the latest Qemu(v1.6.1 at the time of this writing) which had **kvm** code merged from v1.3 onwards, now the readings can be sent virtually any speed - no crash has been experienced thereafter.

Initially, to keep things simpler, the whole client-server connection setup was being carried out for each and every sensor reading that's being sent and received at all points of the Sensor Emulation. Soon, this proved to be inefficient. This was replaced by on-demand and fault-tolerant connection logic which ensures there is absolutely no activity if there's no connected client in the first place and also if either of the client or the server fail or crash, only then the connection is reset at the failed point and that happens immediately. Some of the servers have a **SIGPIPE** handler installed to prevent the entire sensor HAL module from exiting when there is a socket-communication error triggering a **SIGPIPE**. Upon handling **SIGPIPE**, the state of the respective program(thread) is restored performing a **longjmp()** from the previous successful context saved using **setjmp()**.

Until the port-mapping mechanism was studied and deployed, there was no visual improvement in the sensor readings transmission end-to-end. Then came the concept of the dummy forever-sleep server bridging the gap between the host and guest by simply referring the mapped port on behalf of the respective emulator server.

Another big challenge was to find a 3D racing game for a test and demo of the Sensor Emulation, that uses accelerometer sensor readings while it's running. This

game-hunt took almost two whole weeks of tiresome experiment with 100+ games available free for testing from several websites. The real problem that was faced was due to the lack of updated OpenGL ES support on Android-x86. At the time of this writing, only OpenGL ES 1.1 was working on Android-x86 while all the top 3D games would have never been tested in an emulated Android which anyway lacks sensor support, but only real devices which come with OpenGL ES 3.0. So, it took a while to realize that most of the games are ARM based and need ARM emulation in the x86 based Android on Qemu.

libhoudini - the ARM emulation support library which was once hosted by formerly active website **buildroid.org**(now **androvm.org** and no longer hosts these libraries) helped to at least launch some of the ARM based Android games on the emulated Android-x86 which were failing solely for the lack of ARM emulation support, in the first place. However, the binaries of **libhoudini** were not available in the Android-x86 source either. So, it took 2-3 days of searching and hacking to find the binaries ultimately in an ISO image hen Android-x86 JB 4.3 ISO was booted, under **/system/lib** and **/system/lib/hw.**

The lack of the updated OpenGL ES support anyway didn't take the success of having launched the ARM based games any further. So, after few seconds of launch, these games were crashing mostly with a java exception related to mismatch in config specification in one of the graphics related Java sources under Android Framework. Though, there was a workaround announced in some of the websites to a similar error as there was a noted bug in the Android emulator source which was wrongly determining the OpenGL ES version available on the system and was always returning the old versions to be present which was causing the mismatch in the config specification problem possibly due to the differences between the different versions of OpenGL ES libraries. When the announced fix was incorporated into the corresponding Java sources of Android Framework, that specific issue got fixed but something similar kept cropping up. So, any attempt to fix these were given up as it was learned to be the limitation of Android-x86 itself in not having the latest OpenGL ES version of library capable of running the top 3D Android games which are accelerometer based.

Apart from the graphics issue, there was a problem in terms of the game data download forced over WiFi. Though the WiFi drivers were loaded and **wlan0** and **wlan1** interfaces were listed in the emulated Android-x86 environment, they were not coming alive and hence some of the games mandating the game download over WiFi were not tried. Even when the game data was manually placed, they were not considered and download over WiFi was kept on being imposed.

As an effort to run the advanced 3D games, Hardware(HW) Acceleration was enabled on Qemu for Android-x86 with **HWACCEL=1** in the kernel boot parameters. This at

least enabled the latest Asphalt 8 Airborne and other 3D games to be fully launched and successfully run as well. But, the UI was way too slow that could have been tolerated. It was learned from the android-x86.org discussion forums that HW Acceleration isn't fully supported on Android-x86 as the related graphics drivers in Android-x86 are not quite functional at the time of this writing.

Other options of emulated Android were tested like Genymotion, etc. As they were not open-source, nothing much helped. The Intel Atom x86 System image was also tried once for running the 3D games with the Android Standard Emulator. There was no difference but the games kept crashing. The reason wasn't investigated further as it was anyway not the problem of the game but was of the outdated graphics support. This was tried with "GPU Support".

GPU passthrough for Qemu was attempted to enable better HW Acceleration. Since, there was only one graphics card in the machine that was being used and there was no secondary machine to connect from, the idea of GPU passthrough was not tried, initially. Also, for better graphics, before finalizing SPICE for Qemu, other VGA options were tried. But, all of them were not as good as SPICE. Hence, SPICEd Qemu is the one that's being used now.

The last challenge was to emulate all the sensors of a real device, Samsung Nexus S in a week's time while whole of the initial work on Sensor Emulation took an entire set of three months in making the emulation as efficient as possible. So, in addition to Accelerometer, all the other sensors were taken up for the emulation and accomplished in a week's time as agreed upon. The obstacle in accomplishing the seemingly mechanically task of emulating the remaining sensors was due the split in the implementation for the real sensors and virtual sensors wherein the virtual sensors are based on Sensor Fusion. The split meant two different libraries - **`libsensorservice.so`** for virtual and **`sensors.herring.so`** for real. Due to the fact that, the Samsung Nexus S had an official release of Android 4.1.2 as its last update and the emulated Android-x86 and its corresponding source was from JB 4.3, for **`libsensorservice.so`**, there was a mismatch in terms of the symbols in the dependent library **`libutils.so`**. It took a span 3-4 days to stop hacking the 4.3 library with dummy methods for the missing symbols from the dependent libraries. It was decided that a custom ROM of Android 4.3 could be flashed onto the same Nexus S and hence there will be no further problems as the both the device and the emulated Android-x86 will be having Android 4.3. With Android 4.3, the missing symbol issue got fixed and the real coding for emulating remaining nine sensors was started and eventually completed.

With the completion of 10 sensors, search for sensor test applications was on. This included 40+ such applications with most of them showing only raw sensor readings which was not of interest when it is the question of intuitiveness while demoing. So,

the search was then on the individual sensor test applications such as the Light-o-meter for Light sensor and so on. Thus, Sensor Emulation of 10 sensors got accomplished overcoming all these challenges.

Among the miscellaneous but inevitable problems challenging the work, there were quite a few. 1) The very initial troublesome download of the entire Android-x86 JB 4.3 source from android-x86.org. The trouble was due to the intermittent network failures on the download servers forcing repeated downloads which almost appeared never-ending until after a week when it was complete with no syncing issues. To add, initially the attempts were made with number of syncing jobs to be one but later to save time and speed up the work, more than one was started to be used. 2) Insufficient disk space problems preventing full downloads which got fixed after some cleanup. 3) While trying to flash Android 4.3 on the real device, Samsung Nexus S, two of such devices got seemingly damaged and now they are not even switching on. Reason for the same is still not clear as they don't even go to fastboot mode. So, the third device is what being now used with utmost care. 4) The machine(laptop) being used for the entire work had its some of the keyboard keys stopped from working needing an external keyboard, all of a sudden. In addition, the laptop's screen came off needing to tape it together with the base of the laptop.

Last but not the least, the endless determination and energy to complete the entire Sensor Emulation work in a span of approximately three months from September 2013 to November 2013, while the work was targeted for a total of two semesters from Spring 2013 to Fall 2013, single-handedly.

10. Related Work

There is no other system level Sensor Emulation for Android which is crucial in making the emulation much closer to a real hardware driven sensor environment.

SensorSimulator - A userspace Android application providing artificially generated sensor readings based on input devices etc. It's hosted at. However, this has no way of pairing a real Android device with a virtual Android device, although a set of functionality of sensors are only being "simulated" and hence not "emulated". Also, as already mentioned in of the above sections, any application requiring the readings generated needs to be modified so that it contact this stand-alone sensor simulator stand-alone application to get the fake readings being generated, actually "simulated".

SensorEmulation is said to work but is a wired approach based on USB connectivity. It can't be done in a remote manner, say over the network using WiFi. Moreover, according to the lead developer of that project in the respective

GoogleOnlineDiscussionGroup, there hasn't been any work from long time, there will be a little bit of delay in the emulation and also they haven't measured anything. In addition, this sensor emulation work only emulates accelerometer and compass.

In short, there is no such Sensor Emulation work which is system level generic work and emulates the ten sensors the current work emulates as described throughout this document.

11. Conclusion and Future Work

With this work on Sensor Emulation, any sensor-based Android application can be run on the virtualized Android-x86 running on any x86 emulator like Qemu as long as the application is compatible with the emulator, especially in terms of graphics. This opens up a whole new opportunity for the Android developers to test their sensor-based applications in an emulated environment with the real device sensor readings, which was previously impossible and it demanded to have real Android devices installed with the necessary Android OS release, and so on. Due to the real-time nature of the sensor readings available on the emulated Android-x86, there is absolutely no difference in running the same applications on a real Android device and on the virtual Android device. Hence, the sensor-based applications can be accurately tested in a virtualized Android-x86 environment.

The current work described in the paper is about pairing a real Android device with virtualized Android-x86. But, the standard ARM based Google's Android emulator could also be used with no extra work by the simple logic that Android-x86 is more or less the default main Android code with the target being x86 platform. However, the current work has not been tested. So, it's one of the interesting future works.

One-server-many-clients could be implemented to take this Sensor Emulation to another level enabling the sensor-based testing experience in an emulated Android-x86 to a whole new dimension. This would enable pairing a single real Android device with many virtual Android devices. The In addition, any other sensors not already being emulated could be added to the to-do list of Sensor Emulation.

12. References

1. android-x86.org from where the Android-x86 source was obtained.
2. https://android.googlesource.com for the Samsung Crespo device specific sensor code and the virtual sensors code.
3. Online discussion groups: android-x86, adt-dev, android-contrib,

android-developers, stackoverflow.com.

4. stackoverflow.com Android sensor related questions and answers.
5. Android Open Source Code online documentation in general and on HAL and sensors HAL, in specific.
6. In depth discussions with Duke University PhD student, Songchun Fan, mentioned in the header of this document.
7. Several Android applications reviews websites to determine what are the best available sensor based Android applications for testing the current work.