MicKay Mouse: Writing an Android Mouse Driver

Rachel Bobbins, Molly Grossman, Philip Loh, and Jon McKay

SUMMARY

We set out to write a mouse driver to use accelerometer data from an Android phone to control the onscreen pointer. We originally chose to use Bluetooth to transfer this data but after severe difficulties switched to using wireless. The mouse has significant drift, but the pointer does respond to motion of the phone. Furthermore, the Android mouse interfaces smoothly with other mice which the system may be using concurrently, such as a laptop trackpad or USB mouse.

Using an Android phone to control a mouse pointer is not novel—the touchpad of the phone has been used as a mouse, keyboard, and other devices [1]. We therefore chose to use the accelerometer instead of the touchpad to provide a new challenge. In using the accelerometer to change position lies the danger of drift making the error much larger than the small movements which are intended to control the mouse [2]. As expected, we had significant difficulties with this issue which we have not fully resolved.

We used the guidelines in a 1999 Linux Magazine article to write the mouse driver in C [3]. Many of the functions described therein are deprecated, but our driver is structured to follow the protocols outlined in the article. The driver, written in kernel space, was intended to command the mouse directly through a set of standard functions. We discovered that we could not control the mouse in this manner, however. We therefore used the server, which we originally intended to use simply to transmit data from the Android phone to the driver, to control the mouse using `XWindows`. The Android code, which is written in Java, includes a simple user interface which includes two mouse buttons and a calibration button, which is described in more detail below. The change in position at each timestep is calculated on the phone using the accelerometer data. The Android displays the current acceleration in the x, y, and z directions as well as the position in each direction on the same screen as the buttons. We found that the Bluetooth libraries are still in their infant stages in Linux and not yet ready to be adapted easily into a project of this scope, so we switched to using wireless to transmit data from the Android client to the Linux server.

DESCRIPTION

Android

The Android application was programmed in Java and designed to allow the phone to be connected easily to any Linux computer. Initially, we adapted a Google Tutorial about Bluetooth into our application and merged it with another discovered tutorial on using the accelerometer [10]. The phone would not transmit any information until the user pressed the "menu" button and pressed a button to activate the scanning of Bluetooth computers. This worked reliably; however, we encountered many problems in getting our Linux computer to connect with the phone which were were too complicated for our team to address in the this project.

Ultimately, we decided to transmit the position data over wireless using a simple TCP socket connection [9]. The drawback to this is that the computer and the phone must have access to a wireless internet connection in order to function. However, at this point, we believe that wireless is ubiquitous enough that this requirement will not be a problem for most people. Also, Bluetooth becomes spotty after a distance of 15 feet between devices, but wireless is reliable for close to 45 feet. Since we made this design decision, the user still presses the "menu" button, but instead of having the Android scan for computers, the user must enter in the IP Address of the computer to which they would like to connect. With this in mind, another drawback becomes apparent—the user is expected to have knowledge on how to find the IP Address of their computer instead of just allowing the phone to find their computer via Bluetooth.

The change in position is obtained from acceleration by a double integration approximation using Euler's formula. The timestep and change in acceleration are used to compute velocity at each timestep, from which is computed position. To minimize drift, accelerations of sufficiently small magnitude—defined as less than 0.3 m/s^2—are considered to be zero. This helps to reduce the drift from the acceleration, which is never perfectly zero even if the surface is flat.

Once the computer's server and Android phone set up a persistent TCP connection, the user must press a "Calibrate" button in order for data to begin streaming. We discuss the reasons for this in the "Technical Difficulties" section. Each time the Android phone's hardware senses a change in acceleration, it calls a method in which we calculate the change in position from the acceleration. If that position change is between -1 and 1 we do not transmit the data because `XWindows` cannot move the cursor less than 1 increment. Once the position change is first computed, we start a thread which will send the message to the server waiting for data on the computer. We pack that message into the following format: ``X_POS,Y_POS,LEFTCLICKED,RIGHTCLICKED''.

In order to get the click events, two buttons have been set up on the user interface in the normal position on a real mouse. Once a button is clicked, a global variable is set in the application. When a message gets formatted, the global variables for the buttons are checked and then set back to 0. We decided not to send a new message right when the button is clicked because the acceleration gets updated approximately every 100 nanoseconds. This speed is nearly instantaneous and the accelerometer data is rarely zero, so sending extra events for button clicks would not yield a noticeable improvement in the reaction speed of the mouse.

Server

Initially, we designed our system to have a server which simply received messages from the phone, using [8] as a guide to writing the server end of the TCP socket, and passed them to the driver. However, some problems with the driver (discussed in full in the "Technical Difficulties" section) forced us to increase the responsibility of the server. Our revised project consists of the server receiving messages from the phone, passing that message to the driver, and then getting back an array of integers which can be easily passed into `XWindows` function in order to move the mouse.

At the beginning of the script, a child process is forked off and the parent process is killed (making it a daemon thread). The child process then enters a loop which opens the mouse driver, binds to port 27791—randomly chosen as an unlikely port number to be used by another process—and listens for any incoming connections on all interfaces. Once it receives a connection, it enters another loop of receiving messages, sending them to the driver, and calling `XWindows` functions.

We initially had some trouble getting the server to talk to the driver because the server lives in user space and the driver lives in kernel space. Fortunately, the `ioctl` function allows user level programs to send messages to kernel level files which is exactly what we needed. Once we send the message to the driver, we call a read function which allows us to receive messages from kernel land. The read function blocks until the driver fills the provided buffer with results of parsing the message. The server then fills the `XWindows` move and click functions in order to translate that data into actions of the mouse.

We realize that it is useless and slower to receive the message from the phone, then send it to the pseudo-driver which simply parses it, then send it back to the server. It would be much simpler to just have the server parse the message itself. However, much time and energy was spent on creating the driver so we wanted to make sure that it was included in the project. Including the driver gives us the opportunity to use `ioctl` calls and the driver's `read` function on top of the usual `insmod` procedures.

Driver

The mouse driver is classified as a `miscdevice`. All devices are identified by a major number and a minor number. A driver registers its major number with the kernel and keeps track of its own minor numbers, which it uses to differentiate between devices with which the driver communicates [4]. For small device drivers which only interact with a single device, having a separate major number is wasteful. Misc devices therefore help to resolve the disadvantages associated with small device drivers by keeping all misc devices under a single major number, which is 10 for Linux systems [5].

For a Linux user desiring to write a kernel module, the process is manifold. First, the module has to be written in a manner that follows the protocol of a kernel module. This is achieved by importing `<linux/module.h>`, which forces users to comply before it will compile. Compiling a driver module is more complicated than compiling a simple C program. To compile, the `Makefile` must specify that the directory will be changed to `/lib/modules/$(uname -r)/build`, and this line of code will not proceed successfully unless the module was coded according to protocol (which is a useful debugging strategy). After compiling, the module (now a `.ko` file) would have to be inserted into the kernel with an `insmod` call. Were the module not a `miscdevice`, an additional `mknod` call would have been necessary to register the major number, device type (usually `char`), and device address in the `/dev/` directory with the kernel. Upon a successful `insmod` call, the module would be inserted, and the code in the function `module_init` would run. The file operations as specified in the module code can be accessed by calling a user space I/O call (for example, `fopen` or `fread`). At the end, an `rmmod` call is necessary to remove the module from the kernel. This will call `module_exit`.

The driver has the following five file operations: `read`, `poll`, `open`, `release`, and `unlocked_ioctl`. The `open` and `release` operations respectively initialize variables and free the interrupt request. These operations are carried out only if the mouse is, respectively, not yet used or no longer in use. The `read` function blocks until a mouse event (a change in position or mouse button state change) occurs, then sets the new `dx`, `dy`, `button1`, and `button2` states in response to the change. The `poll` function supports asynchronous input and output. The final function, `unlocked_ioctl`, supports input / output control (`ioctl`) calls, allowing the kernel-space driver to be interrupted from user-space [6].

The driver nearly follows the PS2 protocol, which specifies three bytes to be used for data transfer: the first for mouse buttons, the second for change in x position, and the third for change in y position. Ours differs in that we use one byte for each of the two buttons for simplicity; because we wrote our own server, we did not end up needing to follow any set protocol. We also specify the inputs in the following order: dx, dy, button1, button2. Were we to continue this project, we would make the driver compliant with the protocols for an input device. This might allow us to have the android mouse driver read by the computer akin to any other mouse driver, thereby circumventing some of the issues with `XWindows`.

TECHNICAL CHALLENGE

Mouse Driver and `XWindows`

`XWindows` is the utility that controls the graphical user interface in Linux. In order to create a functioning mouse driver, the driver must be registered with `XWindows`. The most common suggestion on how to do this was to edit the `xorg.conf` file with information about the driver and then restart the `XServer`. We tried this multiple times with varying information about our driver but had no success. After repeatedly trying to connect our driver to `XWindows`, we decided use `XWindows` functions from an outside function we wrote instead. However, `XWindows` functions are all in user space and our driver is in kernel space. Because of this difference, `XWindows` couldn't ask for information from the driver through registration of the driver in the `xorg.conf` file and the driver couldn't send `XWindows` information by

calling `XWindows` functions. We decided ultimately to have the driver parse the messages received by the server from the phone into a byte array so that the `ioctl` functionality of the driver was not completely unused.

Bluetooth

We also encountered difficulties connecting to our computer over Bluetooth. We found example code on Google's tutorial site for connecting two Bluetooth-capable phones over Bluetooth and believed we could adapt it to connect a phone and a computer. We had difficulties doing this from the start because those on our team who boot into Ubuntu instead of using a virtual machine in Windows were unable even to turn on Bluetooth on their computers. After scouring the web and consulting the Information Technology department, we happened to solve the problem by booting into our Windows partition, turning on the adapter, and loading back into Linux. We suspect that Windows had a mutex-like lock on the Bluetooth Adapter.

However, even with the phone built-in Bluetooth file transfer application, we were unable to connect and send data to our Linux machine. We were able to leverage our Windows partition again (whose Bluetooth capabilities are more developed) by testing the file sending application, and this test was successful. Unfortunately, we were never successful in connecting the phone to the computer in Ubuntu despite extended research into the subject, so we decided to discontinue that option and begin attempts on connecting via TCP sockets. Although users have to be on wireless in order to use our application, wireless is much faster than Bluetooth (1MB/s for Bluetooth and 11 MB/s for a TCP connection). Although these speeds are probably faster than necessary for this application, it may become necessary if the functionality were to be increased—for example, including touchpad functionality.

After investigating into Bluetooth with no luck, we were able to connect the phone to the computer successfully over a Python TCP socket with significantly less difficulty. Because TCP sockets have been used since the 1970's, the API is much more developer-friendly. Next, we translated our four line python script to a slightly more complicated server in C which then interfaced with the driver.

Noise Accelerometer Data

To obtain position from acceleration in continuous time one would integrate the acceleration twice. In discrete time approximations are necessary to get a value of position from acceleration. We used a forward Euler approximation to obtain velocity from acceleration and position from velocity. This meant that any error in acceleration was seen magnified in the velocity, and this error in the velocity was then compounded in the position.

After preparing the Android application to display current accelerometer data, it was easy to see the poor feasibility of a useful mouse. When the phone was sitting still on a flat surface, the x and y acceleration would fluctuate between 0 and 0.3 m/s^2. We decided to focus our efforts on interfacing the server, the driver, and `XWindows`. Once those were set up and accelerometer data was sent to `XWindows`, the mouse cursor shot off to the corner of the screen. This was likely caused by the compounding error in the accelerometer data each time the position was updated. With the accelerometer updating approximately every nanosecon, our position was skewed to $1E7$ meters within a few seconds.

In order to address this problem, we created a calibrate function. This took the form of a button on the Android phone which needed to be clicked before any data was sent to the server. The calibration function took the first fixed number of accelerometer values and made the average of these values an offset to all future accelerometer values. This compensated for any non-flat surface the phone might be on as well as some of the innate drift from the accelerometer.

We also needed to cap the data that we sent to the server between -127 and 127 so that we did not have a buffer overflow. As a result, the mouse jumps between those two values within a few seconds of starting the stream. There is nothing that we can do about this—the accelerometer is just too noisy for small movements like controlling a mouse.

CONCLUSIONS

From doing this project, we learned that Linux trends are to separate coding from the kernel as much as possible, so drivers tend to be in user space. Drivers using `XWindows` are therefore in user space, and kernel space is reserved for functionality such as graphics cards. We learned how to write a kernel space driver, including the basic functions needed to control the driver, but because we needed to interact with this driver from user space we were unable to implement the functionality of this driver.

We also learned a good deal regarding device connections. The Bluetooth library for Linux is still being built, so coding reliable devices using Bluetooth is much more reliable in Windows than in Linux. When using a means other than a hardware port to transmit data—in our case, wireless—interrupts must be sent by some means other than `irq`s. The `ioctl` function can be used to achieve similar functionality for a wireless device as would an `irq` for a USB mouse.

Finally, we learned that there is definitely a reason mice do not use accelerometer data. Old USB mice used trackballs; current USB mice use optics; and current smartphone mouse emulators usually use the touchscreen. All of these methods are significantly more reliable than double-integration of discrete-time acceleration measurements and do not suffer from drift problems. From other engineering courses we understood we would likely have a problem with drift but hoped the accelerometer would be sensitive enough and reliable enough that we could overcome this difficulty.

REFERENCES

[1] Kang, K., Ha, K., & Lee, J. (2011). Android-based SoD client for remote presentation. *13th International Conference on Advanced Communication Technology (ICACT)*. Retrieved from http://ieeeexplore.ieee.org/

[2] Borenstein, J., Everett, H.R., Feng, L., & Wehe, D. Mobile Robot Positioning—Sensors and Techniques. *Journal of Robitic Systems, 14*(4), 231 - 249.

[3] Cox, A. (1999). Writing Linux Mouse Drivers. *Linux Magazine*. Retrieved from http://www.linux-mag.com

[4] Corbet, J. & Rubini, A. (2001). *Linux Device Drivers, 2nd Edition*. Sebastopol, CA: O'Reilly Media.

[5] Rubini, A. (1998). Miscellaneous Character Drivers. *Linux Journal*. Retrieved from http://www.linuxjournal.com

[6] Corsetti, L. (2004). Controlling Hardware with ioctls. *Linux Journal*. Retrieved from http://linuxjournal.com

[7] Watson, D. (2004). Linux Daemon Writing HOWTO. Retrieved from http://www.netzmafia.de/skripten/unix/

[8] Sockets Tutorial (2010). Retrieved from http://www.linuxhowtos.org/

[9] Incorporating Socket Programming into your Applications (2010). *Think Android*. Retrieved from http://thinkandroid.wordpress.com/

[10] Android Accelerometer Sensor Tutorial (2010). *Androgames blog*. Retrieved from http://blog.androgames.net/

CODE LISTING

Reader's Guide

The files `mouse_driver/amouse.c` and `mouse_driver/Makefile` contain the kernel-level code for our driver. The driver (`amouse.c`) waits for an `ioctl` and blocks on a `read`, and upon a change in the mouse event the driver unblocks the `read` and writes 4 `int`s to a user-specified buffer. The mouse event can be triggered by a hardware interrupt or an `ioctl`.

The files `sockets/server.c` and `sockets/Makefile` contain the server code. The server on the computer listens for devices trying to connect and creates the connection. The server then sends data from the phone to the driver and receives data back from the driver, which it sends to `XWindows`.

The files in `MickayAccel/*` are Java files related to the android client application. Most of the accelerometer-related code was heavily modified from an online tutorial on the android developer website [10]. The TCP code for communicating with the computer was based loosely on the guidelines given in [8].