Teledroid Project - The Missing Synchronization Service for android platform

Peter Burns Lihuan(Riku) Xie Xi Zhang

Department of Computer Science
University of San Francisco
2130 Fulton Street, San Francisco, CA 94117-1080
{rictic, rikutse, xeno.zhang}@gmail.com

Abstract

In this paper we present ***Teledroid***, a synchronization service for Android to enable real-time syncing and aid making use of cloud computing facilities to get more powerful computational ability and faster execution with longer battery life.

Teledroid uses an SSH connection between the server and the mobile device to communicate and determine the files to be transfered. A multi-channel, single connection implementation was included in ***Teledroid*** to communicate with the server and transfer files while minimizing connections and bandwidth used. ***Teledroid*** employs filesystem monitors as a low-overhead method to keep track of changes that need synchronization. The changes from both local and remote are then analyzed to generate a list of files needing synchronization. We use several techniques to prevent our system from unnecessary transfers, including temporary unregistering files from inotify while transferring and synchronizing the file modification time afterwards.

We conducted several experiments to find out whether `inotify` provides a significant performance advantage over a naive scan of the filesystem. While our initial results are disappointing, we have a number of ideas for more nuanced tests, and improvements of our system for future work.

Keywords: I/O, Android, filesystem-monitors, inotify

1 Introduction

Our paper is organized as follows:
Section 2 describes the motivation behind the ***Teledroid*** project. Section 3 describes the architecture and gives a high level overview of the components of the project. Section 4 goes into detail of the implementation of our system. In section 5, we present the results of our experiments to evaluate the performance of ***Teledroid***, and compare the performance differences between filesystem monitoring modes. After the discussion of related work in section 6, we conclude and present our possible future work in section 7.

Acknowledgments We would like to thank Dr. Greg Benson for providing guidance and advice and for pushing us to take on more ambitious work, and the kind folks of `irc://chat.freenode.net/#concatenative`, who helped debug an old GCC, an incomplete version of glibc, and a generally troublesome server environment.

2 Motivation

Modern mobile devices have significant synchronization requirements. With storage in the gigabytes, these devices can carry a users' music collection, photos, videos, and other data. As sensors and software progress, they're increasingly used to produce and download this material directly as well. As this trend continues, the need for more frequent synchronization of this data between a user's mobile device and computer will become more pressing.

Furthermore, by keeping data synchronized between a low-performance mobile computer and a high performance desktop or server, new capabilities become available. For example: the audio of a lecture recorded with the inexpensive microphones in a consumer mobile device will be alternately too quiet then too loud. A single pass through a post-processor can level out the volume and make a tremendous difference, but it would take hours and surely drain the battery of the device. By synchronizing with a powerful server it's not difficult to run an unmodified desktop application on the audio, and the results can then be automatically synchronized back to the device, all without the lengthy and obnoxious step of tethering to the desktop.

We performed a feasibility study of real-time, wireless synchronization. By utilizing filesystem monitors present in the Android operating system and modern desktop operating systems we aimed to keep resource usage reasonable.

3 Architecture

Regarding the design of ***Teledroid*** Application, there are three main components in the architecture, as in Figure 1. Initially, we implement a file browser activity as our main activity in ***Teledroid*** app. Users are able to view all of the files in the Android file system. In addition, the file browser can open audio, plain text, and image files. We've used this file browser for testing purpose. Next, a long running local service can be started from the menu on the file browser activity. The service will stay connected with the server even when ***Teledroid*** is no longer visible. This background service serves to monitoring, scanning, and sync files with remote server. Finally, note that two sorts of threads are invoked by the background service. There's the `FileMonitorThread`, and the `ScanFilesThread`. We prefer to implement threads rather than all the functions within one process because this enables us to decouple functionality and also make blocking calls.

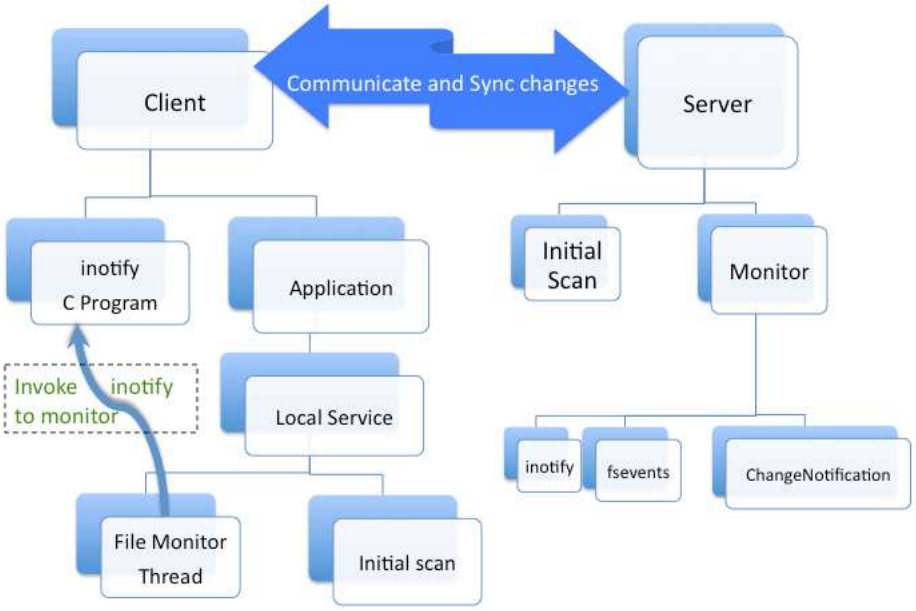


Figure 1: ***Teledroid*** Architecture

4 Implementation

4.1 Client Server Communication

Our implementation of client-server communication and file transfer are based on JSch. JSch is a pure Java open source SSH library. We establish a connection with the remote server when the ***Teledroid*** application starts, and we keep the connection open, as our background service will communicate with remote server from time to time, and transfer files back and forth.

The JSch library provides the ability to open different channels simultaneously using only a single connection, so that we avoid wasting time and resources establishing new connections. We implement a class `Connection` and use it as an abstraction over this interface, enabling us to open shell channels and transfer files over SCP. We implemented functions to open a shell for executing commands on the server side. We also implemented two functions: `SCPTo()` and `SCPFrom()` which follow the SCP protocol for transferring files, with some modifications to keep the modified times synchronized between client and server.

We use simple depth-first walk of the given directory either when looking for changes in scan mode or registering files in monitor mode. When scanning we produce a mapping from filenames, relative to the initial directory that we're watching, to ModificationInfo objects, which store the modification time and other information necessary for determining how to synchronize.

By executing commands on the server, we can get a corresponding map from the server or register a directory to be monitored and retrieve file change mappings. ***Teledroid*** will then compare the two mappings. If the difference of the modified time of the same file is bigger than a tolerance value (one second worked for our testing), then the younger copy of the file will be synced and replace the older one on the other side, while changing the modified time to be the same as the younger one. By this method, we are able to avoid the issue of syncing a file back and forth because the modified time of the file after replacement is newer than its source. More details on this can be seen below in the Synchronization section.

4.2 File Change Notification

Our application design requires a file change notification system, which should allow applications to request the monitoring of a set of files against a list of events. As the nature of running on mobile device, it should require as little system resources as possible and be easy to invoke in user space.

Our implementation of filesystem monitors on the client is based on inotify. inotify is an inode-based file notification system that does not require a file ever be opened in order to watch it. It is designed with an interface that user space application could easily accessed through system calls. Also, inotify communicates with applications via a single file descriptor instead of signals providing simple and fast invocation.

For Linux, inotify was included in the mainline kernel from release 2.6.13 (June 18, 2005)[1]. We checked the default compilation options for Android and support for inotify was not removed. So we wrote a C program to make sure inotify worked in the emulator. The result was very encouraging. And we also found that in Android, the toolbox implements a simple notify program using inofity system calls. However, attempting to use the system provided notify program was unsuccessful. When the notify program was running, our program could inexplicably get nothing back from its stdout. After excluding the possibility of permission issues, we located the problem. The Android notify program will not flush its output stream after writing the result, so our application will block perpetually when reading from notify output stream.

In our first release, we implement our own notify program, cross-compiled and pushed into the Android environment. This program simply implements the functionality of the original google notify program but flushes its output stream every time it finishes writing. The problem with this implementation is obvious. We have to create an extra process and stream with it to communicate. Also, in this system we cannot register and unregister files freely. We could only monitor a directory at a time, or we have to create multiple processes of the notify program, which will surely affect performance.

In our release presented here, we use the Android

JNI Library to invoke the inotify system calls. JNI is not officially supported by google, so there's no documentation for it. We implement a static class Notify as the interface to our notify library. In `libnotify`, we use `initNotify()` to initialize inotify and get the file descriptor. We implement two methods: `registerFile()` and `unregisterFile()` for registering and releasing files to be monitored. We also implement `hasNext()`, `nextEvent()` and `eventMask()` to get event information. And if the event is with a subfile of directory, we use `newFile()` to get sub file information. We cross-compiled the library and put the generated `libnotify.so` file into the lib directory for our application: `data/data/net.solarsail.lib/lib/`, so that in the java static class Notify constructor, we could use `System.loadLibrary(''notify'')` to load the library.

We met with a problem when compile the C library however, the JavaVM passed into `JNI_Onload()` cannot be correctly recognized as a struct. So we ignore the JNI_VERSION check and implement a native method `registerNativeMethod()` to register the JNI native methods. We also include `<utils/Log.h>` so that we can log using the standard Android tools from within our JNI native code.

4.3 Server Side Script

The server-side portion of the project was broken into two parts, corresponding to the two phases of interaction with the server. Initially we construct a mapping of filename to modified time for each file in the portion of the filesystem we're synchronizing by doing a simple walk. This is inefficient in both time and space, but unavoidable, as monitoring for changes on both server and client side is only useful if they're initially synchronized.

After the client receives and processes the mapping of the entire directory, it launches a process that watches for changes. Two approaches were implemented here, pull and push. The pull-sync program gathers changes silently, sending them all in a batch on demand, while push-sync sends each change immediately, as it happens.

This work was implemented in the Factor programming language. Factor is a high level, stack based language with an efficient optimizing compiler. While a very interesting language in its own right, it is still a young language. It was chosen for ***Teledroid*** purely for pragmatic reasons, however. Due in part to its excellent foreign function interface, it is the only known environment with a cross-platform API for filesystem monitoring, with bindings for Linux's inotify, Mac OSX's fsevents, and a family of functions in the win32 API.

JSON was used as a serialization format as its impedance mismatch to our data mostly maps from strings to long integers was minimal. JSON also had the advantage of actually being easy for humans to read while debugging.

The source of the server-side software is available separately at http://github.com/rictic/sync-monitor/tree/master

4.4 Synchronization

Synchronization is one of trickier elements in a distributed system. The issue in our ***Teledroid*** application is that the file after replacement will generate a new modified time, which is the current time and later than the file on the other side. Thus, the file on the other side will be replaced and also generate a newer modified time. Again and again, that file will be transferred back and forth even without any content modification. This is the result of applying the modified time as a key to the file comparison. We could employ checksums in our system, and this might relieve the issue, but with large files the performance impact could be undesirable. Our solution so far is that the new generated file will not be given the current time as its the modified time. Instead, it will be assigned with the modified time of that file on the other side. In this way, ***Teledroid*** is prevented from transferring files back and forth.

4.5 Extra Parts

(Riku)

5 Performance Evaluation

We conducted our tests in two different mode: scan mode, and monitor mode. In scan mode, the relevant

portion of the filesystem is scanned and a tree is built of modification times, both locally and on the server. The server then sends this information down to the local device, where they are compared to determine what needs to be synchronized. In monitor mode, we use filesystem monitors to watch for changed files on both server and client side, so the only files that are considered for synchronization are those which changed.

5.1 Testing Plan

In our experiments, we ran the ***Teledroid***application on the background. In turn, we updated multiple files, totaling 1MB in size, on both the local device and the server side three times. In the experiment, we captured CPU and memory usage every 10 seconds. After ***Teledroid***completed one round of file transfers, we recorded the time spent. The experiments were done in both scan mode and monitor mode.

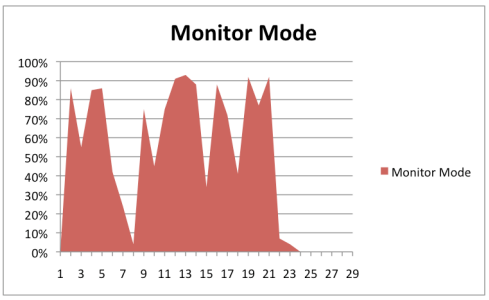
5.2 Hardware Configuration

Our test device is an Android Dev Phone 1. It has a 528MHZ Qualcomm 7210 processor and 192 MB RAM. It features a touch screen and a trackball for navigation, and provides QWERTY slider keyboard for input. Wi-Fi, GPS, and Bluetooth v2.0 are also present. It also supports 3G WCDMA in 1700/2100 MHz and Quad-band GSM in 850/900/1800/1900 MHz. Note that the Android Dev Phone 1 includes 1GB MicroSC card as an external storage device. It can be replaced with a card of up to 16GB. Concerning the network environment, our tests utilized the University of San Francisco's internet connection, accessed through 802.11g.

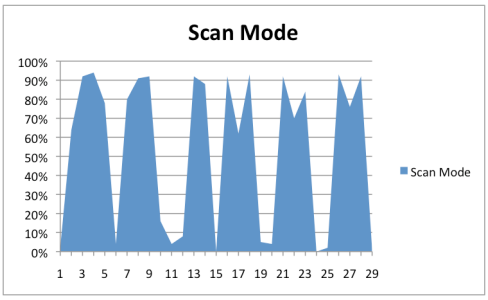
5.3 Results

The results of our CPU usage tests are shown in Figure 2. However they were not as we expected. Monitor mode didn't gain any significant advantage over scan mode here. However, synchronization finished faster than in scan mode, so overall cpu usage should be lower than our tests indicate.

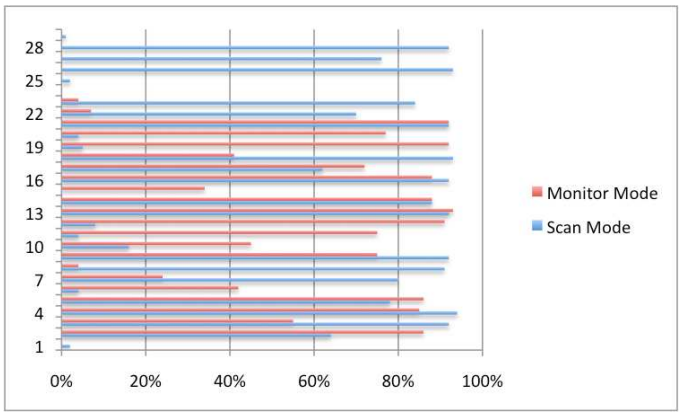
CPU Usage	1	2	3	4	5	6	7	8	9	10
Scan	2%	64%	92%	94%	78%	4%	80%	91%	92%	16%
Monitor	0%	86%	55%	85%	86%	42%	24%	4%	75%	45%
CPU Usage	11	12	13	14	15	16	17	18	19	20
Scan	4%	8%	92%	88%	0%	%	%	%	%	%
Monitor	75%	91%	93%	88%	34%	%	%	%	%	%
CPU Usage	21	22	23	24	25	26	27	28	29	30
Scan	%	%	%	%	%	%	%	%	%	%
Monitor	%	%	%	%	%	%	%	%	%	%



(a) CPU Usage in Monitor Mode



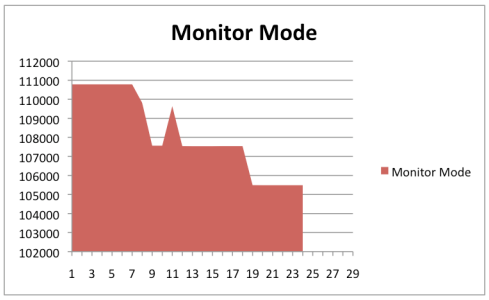
(b) CPU Usage in Scan Mode



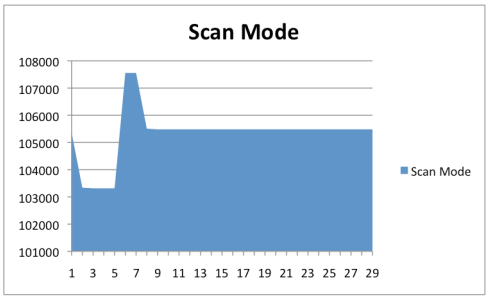
(c) Comparison of CPU Usage

Figure 2: The CPU usage comparison of Monitor and Scan mode

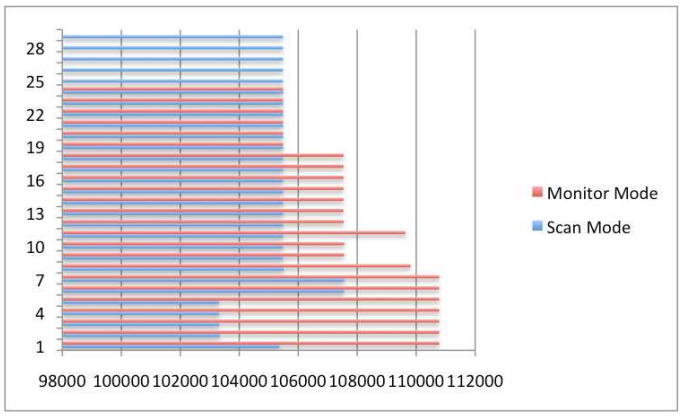
Similar to our CPU usage results, the memory usage data was also disappointing. As shown in Figure 3, the memory usage in monitor mode is even higher than scan mode initially. We think this is due in part to the fact that in monitor mode we have to perform an initial complete scan before we can switch over to the low overhead scan mode. This, combined with the fact that monitor mode makes use of several more threads and an external JNI module seems to explain the difference.



(a) Memory Usage in Monitor Mode



(b) Memory Usage in Scan Mode



(c) Comparison of Memory Usage

Figure 3: The Memory usage comparison of Monitor and Scan mode

In our tests of synchronization speed we finally got data that indicate that monitor version performs better. As we can see in Figure 4, the speed of monitor mode is faster than scan mode on the client side. However, the server didn't show significant differences. This may be due to our current server side script using a pull-sync instead of push-sync as client side.

Speed	Local ⇒ Server (Scan Mode)	Server ⇒ Local (Scan Mode)	Local ⇒ Server (Monitor Mode)	Server ⇒ Local (Monitor Mode)
Round 1	42.7	45.2	27.2	43.3
Round 2	40.9	41.4	30.7	44.2
Round 3	42.2	41	29.1	42.3
Averge	41.9	42.5	29	43.3

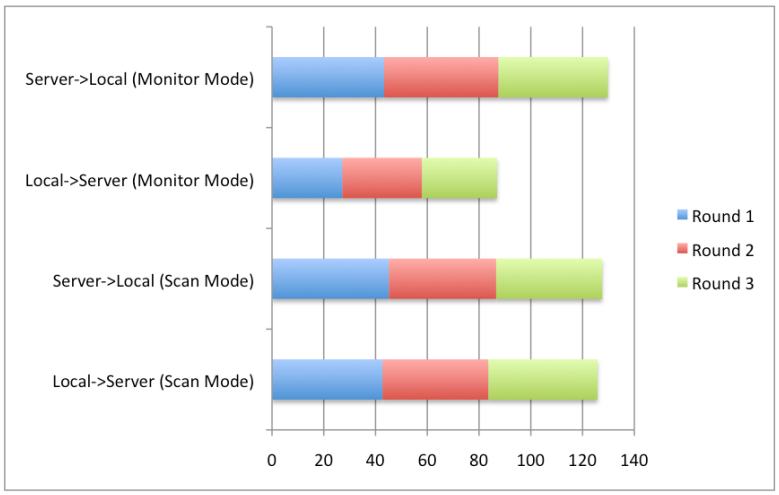


Figure 4: Comparison of Synchronization Speed

6 Related Work

Related work here.

7 Conclusions and Future Work

While our experimental results weren't what we hoped, we're nevertheless optimistic about our approach. Due to the complexity of the system and our limited time, we had comparatively few iterations of testing. With this information in hand and with careful tracing we should expect to significantly reduce memory and cpu usage for the monitoring approach. Our experiments were also more artificial than we would like. Ideally we'd like to capture several real world scenarios. Intuitively we would expect that real-world tests would feature longer times with no changes, and only short bursts where synchronization is needed. Real world use might also have larger, more complicated directory trees with more files, which we would expect would favor a monitor-based system.

There are several factors that we didn't have time to implement and experiment with as well. The period between communicating with the server could lengthen over time as no changes are detected. As this period gets longer, at some point it may make sense to no longer maintain a constant connection with the server, only reconnecting when the period is up.

On the other hand, if we keep the connection open and use a push based system from the server we wouldn't need to have an explicit waiting period at all. This would require the Wifi radio to be actively listening however, which might raise the device's idle energy usage too high.

In short, we still believe that wireless synchronization may be feasible. There are still a number of parameters that can be varied to get better performance before it's clear whether it is too inefficient. We also remain convinced that as mobile devices become more capable, this feature will only become more desirable.

References

[1] R Love. Kernel korner: Intro to inotify. *Linux Journal*, Jan 2005.