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.

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 is able to view all the files in Android file system. Besides, the file browser can open the files of at lease audio, plain text, and image format. We use this file browser for testing purpose. Next, a long-time running and local service can be started from the menu on the file browser activity. The service will stay connecting with server even when ***Teledroid*** is no longer visible. This background service serves with the functionalities of monitoring, scanning, and syncing files with remote server. At last, note that two sorts of threads will be invoked by the background service. There are `FileMonitorThread`, and `ScanFilesThread`. We prefer to implement threads rather than all the functions within one process because we could like to decouple each functionality so as to relieve the synchronization issue.

4 Implementation

4.1 Client Server Communication

Our implementations of the the communication and file transfer are based on JSch. JSch is a pure java open source SSH library. We will establishing internet connection with remote server when application starts, and keep the connection open. As our background service will communicate with remote server from time to time, and transfer back and forth.

JSch library provides the ability to open different channels simultaneously using only one session, so that we could avoid wasting time and resource establishing the connection. We implements a class

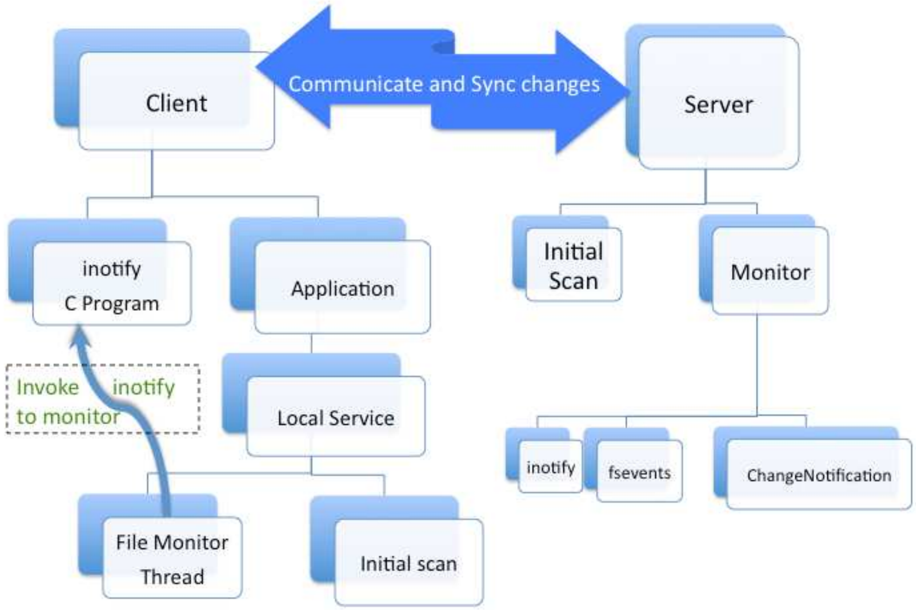


Figure 1: ***Teledroid*** Architecture

`Connection` and put in all the connection related functions, including open shell channel and transfer file over SCP. We implement function to open a shell for executing command on server, scan remote file information and comparing the changes lists from local and server using JSON. We also implement two functions `SCPTo()` and `SCPFrom()` follows the SCP protocol for transferring files with time difference to keep both side synchronized. As we could have several concurrent ongoing channels connected with server at the same time, we could have shell channels to persistently communicate with remote server, and meanwhile still be able to transfer files. By getting output stream from channel, we could receive output data from the shell of remote server. Similarly, ***Teledroid*** could execute command on server by writing a input stream to the Shell channel.

We use FIFO algorithm during the scan of the filesystem, either looking for changes or registering files. A root directory is pushed into a stack at the initial time. Then, with popping out the root directory, we scan all the files in the root directory. If the files are also directory, we push them back to stack. If not, the file name with absolute path and the modified time of that file will be stored in a list. Recursively, we will retrieve the modified time of all the files in root directory in a list or register all the files in the root directory into notify.

By executing command on server, we can get the same kind of list from server or register server into inotify and retrieve file change list. Those two lists will be converted to JSon object, which is a collection of name/value pairs with more powerful supports. ***Teledroid*** will then compare these two JSon objects. If the difference of the modified time of the same file is bigger than one second, the younger copy of file will be synced and replaced the old one on the other side with changing the modified time as the same as the younger one. By this method, we are able to avoid the synchronization issue that syncing back and forth the file because of the modified time of the new files after replacement. More details are on Synchronization section of this paper.

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 only require very little system resource and be able to be easily invoked in user space.

Our implementation of filesystem monitor 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)[2]. We checked the default compilation option for Android, the support for inotify was not removed. So we wrote a piece of C program to make sure the inotify in emulator. The result was very encouraging. And we also found that in Android, the toolbox have implements a simple notify program using the inofity system call. However, the attempt of using the system provided notify was unsuccessful. When the notify program was running, our program can get nothing back from its stdout. After excluding the possibility of permission issue, we located the problem. Android notify program will not flush the output stream after writing the result, so our application will block when reading from notify output stream.

In our first release, we implement our own notify program, cross-compiled and pushed into Android environment. This program simply implements the functionality of the original google version notify program but will flush the 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. And also, in this way we cannot register and unregister for single files freely. We could only monitor a directory at a time, or we have to create multiple processes of notify program, which will surely affect the performance.

In the final release, we use 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 the `libnotify`, we use `initNotify()` to initialize inotify and get the file descriptor. We implement two method `registerFile()` and `unregisterFile()` for registering file to be monitored and release the monitor. We also implements `hasNext()`, `nextEvent()` and `eventMask()` to get the event information. And if the event is with a subfile of directory, we use `newFile()` to get the sub file information. We cross-compiled the library and put generated `libnotify.so` into the lib directory for our application: `data/data/net.solarvistas.android/lib/`, so that in java static class Notify constructor, we could use `System.loadLibrary(''notify'')` to load the library.

We met a problem when compile the C library, the JavaVM passed into `JNI_Onload()` cannot be correctly recognized as struct. So we ignored the JNI_VERSION check and implements native method `registerNativeMethod()` to register the JNI native methods. We also include `<utils/Log.h>` so that we could retrieve these information from Android ddms logcat.

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.

4.4 Synchronization

Synchronization is one of the significant element 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, 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 checksum in our system, this might relieve the issue a little. However, the checksum approach has a hard time to cope with distinguishing the new and the deleted files. So now, our working version only support comparison with the modified time. Our solution so far is that the new generated file will not has the current time as the modified time. Instead, it will be assigned with the modified time of that modified file on the other side. In this way, ***Teledroid*** will prevent from transferring files back and forth and become more efficient.

4.5 Extra Parts

(Riku)

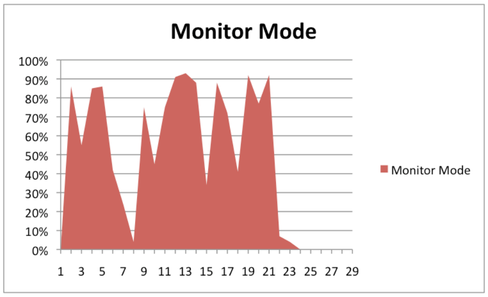
5 Performance Evaluation

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. In our experiments, ...

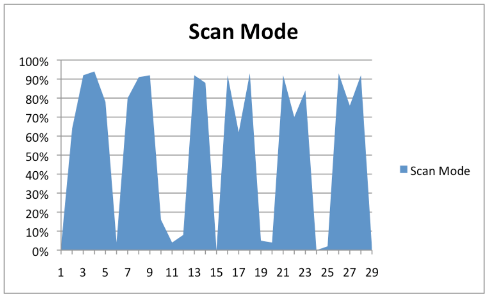
5.1 Testing Plan

(Riku) Plan:
- scan mode
- monitor mode
- lazy mode (Future work)

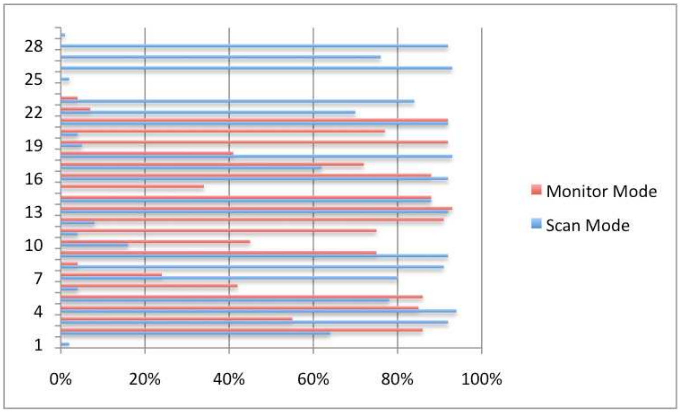
sample: - one Large-size file
- multiple small-size files



(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

(with new or modified files)

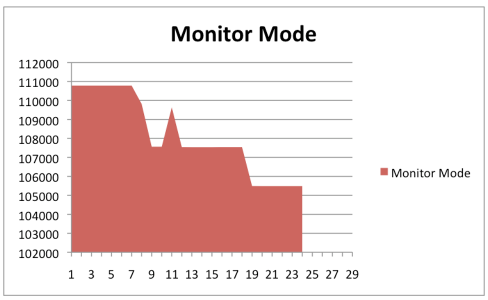
5.2 Hardware Configuration

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.

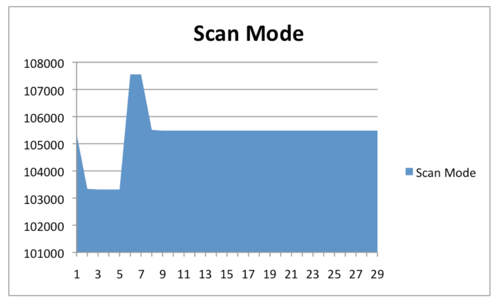
5.3 Results

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

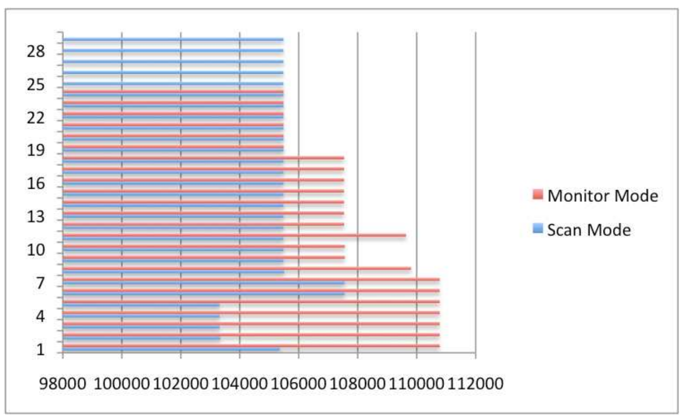
The same as CPU usage, the memory usage data is also disappointed. As shown in Figure 3, the memory usage in monitor mode is even higher than scan mode. We think this is due to our implementation use only `ScanFileThread` in scan mode while using an extra `FileMonitorThread` in monitor thread.



(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 the test of synchronization speed, we finally got the data represent the the performance of monitor version is better. As we can see in Figure 4, the speed of monitor mode is faster than scan mode on 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.

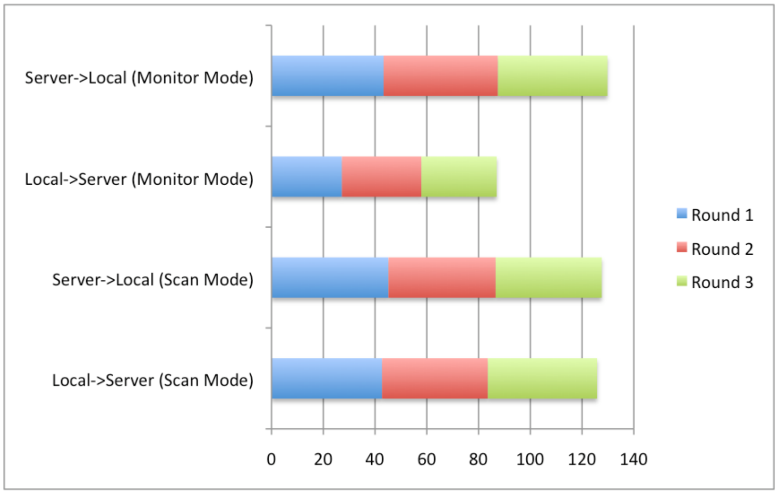


Figure 4: Comparison of Synchronization Speed

6 Related Work

Related work here.

7 Conclusions and Future Work

- deleted files
    - conflict and merge files
    - lazy mode
    - more??

References

[1] Google Android Group. *Android Online Documentation*.

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