Replay Debugger for Human Interactive Multiple Threaded Android Applications

by

He Lu

A Thesis Presented in Partial Fulfillment of the Requirements for the Degree Master of Science

Approved November 2012 by the Graduate Supervisory Committee:

Yann-Hang Lee, Chair
Georgios Fainekos
Yinong Chen

ARIZONA STATE UNIVERSITY

December 2012
ABSTRACT

Debugging is a boring, tedious, time consuming but inevitable step of software development and debugging multiple threaded applications with user interactions is even more complicated. Since concurrency and synchronism are normal features in Android mobile applications, the order of thread execution may vary in every run even with the same input. To make things worse, the target erroneous cases may happen just in a few specific runs. Besides, the randomness of user interactions makes the whole debugging procedure more unpredictable. Thus, debugging a multiple threaded application is a tough and challenging task.

This thesis introduces a replay mechanism for debugging user interactive multiple threaded Android applications. The approach is based on the 'Lamport Clock' concept, 'Event Driven' implementation and 'Client-Server' architecture. The debugger tool described in this thesis provides a user controlled debugging environment where users or developers are allowed to use modified record application to generate a log file. During the record time, all the necessary events like thread creation, synchronization and user input are recorded. Therefore, based on the information contained in the generated log files, the debugger tool can replay the application off-line since log files provide the deterministic order of execution. In this case, user or developers can replay an application as many times as they need to pinpoint the errors in the applications.
To My Beloved Family
ACKNOWLEDGMENTS

I would like to express my greatest gratitude to all the people who helped, encouraged and supported me throughout my project.

I am grateful to my advisor Dr. Yann-Hang Lee for his initial idea at the first stage, the continuous support during the project implementation and the revision advice when I was writing the thesis. Without his guidance, this thesis would not have been possible. I have gained lots of practical knowledge about the field I researched, improved both design and implementation skills, and enriched Android related development experience.

I would like to sincerely thank Dr. Fainekos Georgios and Dr. Yinong Chen for their time and effort to help me to fulfill my degree requirement, as part of my thesis committee.

I would also like to thank all my friends who encouraged me when I felt frustrated. They make the whole experience a valuable and enjoyable memory for me.

I am thankful for my lab mates who helped me answer my questions and shared happiness with me when the project made progress.

Finally, I am indebted to my parents and my family. They always have the best wishes on me and support me constantly. They encouraged me when I felt upset and make me feel at home even if I am thousands miles away from them. I wish I could express my utmost and deepest gratitude to them for being supportive warmly.
TABLE OF CONTENTS

LIST OF FIGURES ........................................................................................................................................... vii

CHAPTER

1 INTRODUCTION ............................................................................................................................................... 1
  1.1 Background.............................................................................................................................................. 1
  1.2 Motivation ............................................................................................................................................... 3
  1.3 Document Outline ................................................................................................................................. 6

2 BACKGROUND .............................................................................................................................................. 8
  2.1 Multiple Threaded Applications ............................................................................................................ 8
  2.2 Non Deterministic Order of Execution ................................................................................................. 9
  2.3 Deterministic Replay ............................................................................................................................. 11
  2.4 Lamport Clock and Happen-Before Relation ....................................................................................... 12
  2.5 Android Architecture ............................................................................................................................ 14
    2.5.1 Applications ..................................................................................................................................... 15
    2.5.2 Application Framework ................................................................................................................... 15
    2.5.3 Libraries and Android Runtime .................................................................................................. 15
    2.5.4 Linux Kernel ................................................................................................................................... 16
    2.5.5 Dalvik Virtual Environment ........................................................................................................ 17
  2.6 Android Application Architecture ........................................................................................................ 18
  2.7 Android C-S (Client-Server) Architecture ........................................................................................... 19
  2.8 Android IPC Mechanism (Binder) Architecture .................................................................................. 22
  2.9 Android Socket Architecture ................................................................................................................ 25
<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.10 Android Event-Driven Architecture</td>
<td>26</td>
</tr>
<tr>
<td>2.11 Android Application’s Life-cycle</td>
<td>28</td>
</tr>
<tr>
<td>3 RELATED WORK</td>
<td>31</td>
</tr>
<tr>
<td>3.1 General Replay Techniques</td>
<td>31</td>
</tr>
<tr>
<td>3.2 JaRec</td>
<td>31</td>
</tr>
<tr>
<td>3.3 JReplay</td>
<td>32</td>
</tr>
<tr>
<td>3.4 DejaVu</td>
<td>33</td>
</tr>
<tr>
<td>3.5 Multiple Threaded Replay Debugger</td>
<td>34</td>
</tr>
<tr>
<td>4 DESIGN AND ALGORITHM</td>
<td>35</td>
</tr>
<tr>
<td>4.1 Design Overview</td>
<td>35</td>
</tr>
<tr>
<td>4.2 Algorithm Definition</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Key Event and Key Event Queue</td>
<td>38</td>
</tr>
<tr>
<td>4.4 Internal Events Record</td>
<td>41</td>
</tr>
<tr>
<td>4.4.1 Synchronization</td>
<td>41</td>
</tr>
<tr>
<td>4.4.2 Thread Creation</td>
<td>43</td>
</tr>
<tr>
<td>4.4.3 Message Passing</td>
<td>44</td>
</tr>
<tr>
<td>4.5 External Events Record</td>
<td>45</td>
</tr>
<tr>
<td>4.6 Record Overview</td>
<td>46</td>
</tr>
<tr>
<td>4.7 Internal Events Replay</td>
<td>47</td>
</tr>
<tr>
<td>4.8 External Events Replay</td>
<td>49</td>
</tr>
<tr>
<td>4.9 Replay Overview</td>
<td>51</td>
</tr>
<tr>
<td>4.10 Communication between C-Level and Java-Level</td>
<td>53</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>IMPLEMENTATION</td>
</tr>
<tr>
<td>---------</td>
<td>----------------</td>
</tr>
<tr>
<td>5</td>
<td>5.1 Record Log Structure</td>
</tr>
<tr>
<td></td>
<td>5.2 Initialization</td>
</tr>
<tr>
<td></td>
<td>5.3 Internal Events Record</td>
</tr>
<tr>
<td></td>
<td>5.4 External Events Record</td>
</tr>
<tr>
<td></td>
<td>5.5 Internal Events Replay</td>
</tr>
<tr>
<td></td>
<td>5.6 External Events Replay</td>
</tr>
<tr>
<td>6</td>
<td>COMPARISON AND CONCLUSION</td>
</tr>
<tr>
<td>7</td>
<td>FUTURE WORK</td>
</tr>
<tr>
<td></td>
<td>References</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Synchronization Data Race I</td>
<td>10</td>
</tr>
<tr>
<td>2.</td>
<td>Synchronization Data Race II</td>
<td>10</td>
</tr>
<tr>
<td>3.</td>
<td>Deterministic order execution architecture</td>
<td>12</td>
</tr>
<tr>
<td>4.</td>
<td>Android Architecture</td>
<td>14</td>
</tr>
<tr>
<td>5.</td>
<td>Android Applications and Widgets</td>
<td>15</td>
</tr>
<tr>
<td>6.</td>
<td>Android Application Framework</td>
<td>15</td>
</tr>
<tr>
<td>7.</td>
<td>Android Libraries and Android Runtime</td>
<td>16</td>
</tr>
<tr>
<td>8.</td>
<td>Android Linux Kernel</td>
<td>17</td>
</tr>
<tr>
<td>10.</td>
<td>Android Client Server Architecture</td>
<td>20</td>
</tr>
<tr>
<td>11.</td>
<td>Android System Service List</td>
<td>21</td>
</tr>
<tr>
<td>12.</td>
<td>Android Binder Structure</td>
<td>22</td>
</tr>
<tr>
<td>13.</td>
<td>Android Binder Communication Example</td>
<td>24</td>
</tr>
<tr>
<td>14.</td>
<td>Android Socket</td>
<td>25</td>
</tr>
<tr>
<td>15.</td>
<td>Android Window Manager Service Architecture</td>
<td>27</td>
</tr>
<tr>
<td>16.</td>
<td>Android Application’s Life Cycle</td>
<td>28</td>
</tr>
<tr>
<td>17.</td>
<td>Android Application Status</td>
<td>30</td>
</tr>
<tr>
<td>18.</td>
<td>Record Overview Architecture</td>
<td>35</td>
</tr>
<tr>
<td>19.</td>
<td>Replay Overview Architecture</td>
<td>36</td>
</tr>
<tr>
<td>20.</td>
<td>Immediate Happen Before Example</td>
<td>38</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>21</td>
<td>Key Event Attributes</td>
<td>39</td>
</tr>
<tr>
<td>22</td>
<td>Key Event Construction Function</td>
<td>39</td>
</tr>
<tr>
<td>23</td>
<td>Key Event Action Definition</td>
<td>40</td>
</tr>
<tr>
<td>24</td>
<td>Key Event Queue Attributes</td>
<td>41</td>
</tr>
<tr>
<td>25</td>
<td>Synchronization Block</td>
<td>42</td>
</tr>
<tr>
<td>26</td>
<td>Synchronization Method</td>
<td>43</td>
</tr>
<tr>
<td>27</td>
<td>Message Passing</td>
<td>44</td>
</tr>
<tr>
<td>28</td>
<td>External Event Record Architecture</td>
<td>45</td>
</tr>
<tr>
<td>29</td>
<td>Record Overview Structure</td>
<td>47</td>
</tr>
<tr>
<td>30</td>
<td>Events Replay Example I</td>
<td>48</td>
</tr>
<tr>
<td>31</td>
<td>Events Replay Example II</td>
<td>49</td>
</tr>
<tr>
<td>32</td>
<td>External Events Replay Architecture</td>
<td>51</td>
</tr>
<tr>
<td>33</td>
<td>Replay Overview Architecture</td>
<td>52</td>
</tr>
<tr>
<td>34</td>
<td>Socket Architecture between C layer and Java Layer</td>
<td>54</td>
</tr>
<tr>
<td>35</td>
<td>Log File Example</td>
<td>55</td>
</tr>
<tr>
<td>36</td>
<td>DVM Initialization Procedure</td>
<td>58</td>
</tr>
<tr>
<td>37</td>
<td>Window Manager Service Initialization Procedure</td>
<td>58</td>
</tr>
<tr>
<td>38</td>
<td>Synchronization Block in C (lock)</td>
<td>59</td>
</tr>
<tr>
<td>39</td>
<td>Synchronization Block in C (unlock)</td>
<td>60</td>
</tr>
<tr>
<td>40</td>
<td>Internal Event Record Implementation Architecture</td>
<td>60</td>
</tr>
<tr>
<td>41</td>
<td>External Event Synchronization Lock</td>
<td>61</td>
</tr>
<tr>
<td>42</td>
<td>Internal events Replay Implementation Architecture</td>
<td>62</td>
</tr>
</tbody>
</table>
43. External events Replay Implementation Architecture .................. 63
Chapter 1

INTRODUCTION

1.1 Background

Quality Assurance has become one of the most important steps to ensure the quality of software products. Debugging, as an inevitable part of QA (Quality Assurance), is the most challenging part among the whole quality assurance procedures. Based on the result of a survey, in a 2002 NIST report (Tassey), the average time spent on coding and unit testing is 4.9 hours, whereas, average speaking, finding a bug in post-product release takes 15.3 hours to fix. According to the data they provided, the average time a developer spent in order to investigate and resolve a bug is 17.4 hours. The amount of time is quite costly and increases the software developer’s work load incredibly.

Therefore, debugging is expensive, while debugging in a multi-threaded application can be more cost consuming, since the thread execution order may vary between every run. If we think one step further, debugging in a multi-threaded application with user interaction is even more complicated.

One commonly used debugging technique is sequential debugging or cyclic debugging. Using this method, developers execute the application several times in order to pinpoint the erroneous section by looking into application logs and adding break-point to debug that section in detail. However, what is paramount to this schema is that the erroneous
execution should be guaranteed to be replayed with every run. Unfortunately, this is not always the case.

In basic single thread application, the workflow of application is sequential. Therefore, the assumption that the erroneous execution can be replayed might be true. However, in multi-threaded application, the workflow is concurrent. In other word, the erroneous execution might happen just in one of those workflows. This situation makes the sequential debugging a little bit awkward. One vital problem is that the error message can only be printed in a few random runs. In this case, when user wants to debug the application step by step, the error messages are not always available, which would make the whole debugging procedure incredibly tough and time consuming.

In practical situations, there is always the case that user invalid inputs become the root cause of an application crash. Thus, keeping track of human interaction is very necessary for designing a complete and formal debugging schema. However, it is unrealistic to make users or developers to remember all their actions and re-input them accordingly in order to identify which action leads to the crash.

All of the situations discussed above make the multiple threaded applications non-deterministic and hard to debug in conventional way. In cycling debugging or sequential debugging schema, developers have to look into the dumped trace data from a failed run to figure out the cause of the problem, e.g. deadlock and starvation, which is definitely time
consuming and error-prone. The replay debugging mechanism proposed in this thesis improves the debugging efficiency by replaying the recorded run according to log files.

1.2 Motivation

Mobile computing is in a rapid evolution as a new platform of tale-computation platform. Based on the statistic provided by Microsoft Tag (Aden, Hepburn), there are 1.08 billion people using smart-phones by 2011 and by 2014, mobile internet usage is slowly taking over desktop internet usage.

Android is a mobile device operating system provided by Google and it is one of the most popular mobile operating systems in current market. The importance and popularity of Android in current market is increasing dramatically day by day. Advantages of choosing Android as the experiment platform are, 1) Source code of Android operating system is completely open source, which simplifies code modification for both server-end and client-end. 2) Android has a large-scale repository of applications and most of them are multiple threaded. Therefore, client-end selection is adequate. 3) Android uses event driven mechanism which maintains a message queue to implement all the actions including user behavior. Also Android has a strict and clear service structure and each service has its own responsibility. 4) Android provides its own debug tool ADB (Android Debug Bridge), DDMS (Dalvik Debug Monitor Server) and
Trace-view / logcat (Google “Tools Overview”). These tools are helpful for the implementation of this proposed debugger for multiple threaded applications.

The classical approach of single step debugging cannot be used with multiple threaded Android applications as it would affect the temporal component of the system, while interacting with the outside world. The cause is that breaking the execution will only break the internal execution while the external process will continue. Also, it won't allow extra user behavior re-input. Thus, even if the execution of target application is stopped, the outside world with all its events is still running. Therefore, that is not a strict deterministic order replay.

The inability of classical approach to deal with multi-threaded application has encouraged several researchers to look into this problem. Numerous approaches have been developed and published in order to deal with the complexity of multiple threaded Android applications. One of the most well-known and famous approach is deterministic replay or trace based replay. In that approach, sufficient information is collected during one execution of a multiple threaded application, in order to guarantee that the current execution (order of events) can be reproduced deterministically. After the trace file is ready, developers use classical sequential debugging or cyclic debugging skills to look into the erroneous runs in steps.
The proposed replay mechanism is obtained by instrumenting both (DVM) Dalvik Virtual Machine and Android C-S structure (Client-Server structure). This mechanism has two modes. 1) Record Mode, in this mode the tool records every critical event including thread-creation, synchronization and external user input etc. and put them into a log file. 2) Replay Mode, replay engine restarts the previous recorded run by using the existing log file.

During the record phase, the main purpose is to mark every critical event with deterministic order and record when these external user inputs should happen during the whole running time. The log-file maintains all necessary information like LC (Lamport Clock) number, happen-before sequence number (will explain later) and external events related action code (will explain later) etc.

During the replay phase, the main goal is to use the log-file to verify is this run identical (thread event order) with the target one. Also it has to create the emulated external events at the right time and push it to the system message queue, in order to let the message processed immediately.

LC (Lamport Clock) is being used in order to provide a standard which can be used as the ordering of events while recording. It is a logical clock that describes an order that gives the partial ordering between different events of the same thread and events in other threads.
Therefore, the partial order can make sure that synchronization events will be reproduced successfully off-line.

This approach is implemented in Android and can be applied in any platform, as long as it follows the C-S (client-server) structure and it is an event driven system. Also, non-critical events do not affect the critical events' replay. In another word, during both the record and replay phase, non-critical events won't affect the order of critical events' execution. They are going to follow the system scheduler as they always do.

1.3 Thesis Outline

The rest of document is organized as follows.

Chapter1 Introduction

This chapter gives an overview of the proposed replay method and motivations why we need to do that.

Chapter2 Background

This chapter provides the necessary background knowledge to help explain the whole replay procedure.

Chapter3 Related Work

This chapter enumerates some related works which are implemented by other researchers.
Chapter 4 Design and Algorithm

This chapter provides the blueprint of the proposed design and has a clear explanation of implemented algorithm.

Chapter 5 Implementation

This chapter shows how this proposed design is implemented in practical Android platform.

Chapter 6 Comparison and Conclusion

This chapter gives tables and figures in order to demonstrate the result of implemented replay method.

Chapter 7 Future Work

This chapter presents some features and enhancements that can be added to the existing implementation.
Chapter 2

BACKGROUND

2.1 Multiple Threaded Applications

Applications can be divided into two types. One called single threaded applications and the other is multiple threaded applications.

Single threaded applications are easy to understand and debug. They only have one main work-flow and the execution is sequential. Therefore, it is not difficult to replay the erroneous run for this kind of application, as long as developers give the same input every run.

However, as for multiple threaded applications, the applications have more than one thread running concurrently, in which case they may have more than one task running at the same time (A.J. Bernstein). On a single CPU system, threads are scheduled by interleaving or based on priority. While on a multiple CPU system, threads could run concurrently depending on the number of CPUs. This concurrency leads to some difficulties in application debugging. The most important issue that the thesis is concerned with is the non-deterministic order of thread execution. Due to this reason, same input cannot guarantee the same erroneous run happen every time.

An Android application can be multiple threaded due to the Java’s mature concurrency mechanism. Also at the VM (Virtual Machine) level, which is DVM (Dalvik Virtual Machine) in Android, POSIX (Portable
Operating System Interface) threads provide the support of current execution at the low level.

2.2 Non Deterministic Order of Execution

Non-deterministic order of execution can be found commonly in multiple threaded applications. In most cases, the order of thread execution depends on the system’s scheduler which schedules threads based on their priority level at most cases. Also, due to the implementation of concurrency, there are many inter-thread communications happen all the time. Whenever system switches the current running thread, there is a context switch, which will result the change of real execution order. Therefore, the order of execution may vary each time.

In other cases, multiple threads may get access to the same shared memory. If two threads access the same shared data with an unsynchronized manner, this case may become a data race. An example is shown in the following fig.
Fig. 1 Synchronization Data Race I

Fig. 2 Synchronization Data Race II
The two figures above demonstrate a simple example situation of synchronization data race in multiple threaded applications. Thread one has two operations “i=i+4” and “j=j*2” and thread two has two operations also “i=i+2”, “j=i*3”. These four operations will be operated sequentially on their own thread. However, what may vary is the thread execution order. In figure one, thread two interrupts the thread one before it finishes its job, therefore the output is “i=7, j=42”. However, on the other case, thread two waits until thread one finishes its own job. So the output in this case is “i=7, j=21”.

This is a simple example of data race to illustrate that the order of thread execution may cause the different output of an application.

In this thesis it is assumed that the application is free of this kind of data races, so the proposed application doesn’t need to check whether there is a data race when the recording engine is running, which may incredibly heavy the burden of recording to check the data race condition. A good reason is that data race can be detected by other tools.

2.3 Deterministic Replay

Deterministic execution replay scheduler is to ensure the replay of application’s thread execution remains the same as the recorded one. Whenever a new event occurs, scheduler stops the running thread first and checks if the event should be allowed to execute. If so, the scheduler thread allows this event execute. Otherwise, the scheduler maintains a
mutex lock on that specific thread until all the prerequisites are satisfied and then releases the corresponding lock and let the thread continue to run. Under this mechanism the replay application can guarantee the deterministic order of thread execution.

Fig.3 Deterministic order execution architecture

2.4 Lamport Clock and Happen-before Relation

Lamport clock (Leslie Lamport) is a simple method to define a partial order between events in a distributed system. It is a logical clock rather than a physical clock. Therefore, it has the following rules when counting.
1. Each thread increases their own counter, whenever a synchronization event happens on one thread, the counter (Lamport Clock counter) on that specific thread plus one.

2. Whenever there is an execution order change between two threads. The previous thread is going to include its own counter value (Lamport Clock value) with the sending message and send it to the new process.

3. The new thread received this coming message and gets the counter value (Lamport Clock value). Then the receiver thread set his own counter value to be one more than, the maximum of its own counter value and the received counter value.

4. Each thread keeps increase their own counter and whenever another order of thread execution change happen, follow the step 2 and step 3.

After the update of Lamport Clock count has finished. The logical execution sequence has been recorded. Therefore, the “happen-before” relation comes out. This relation is noted by “->” and it can be observed by following rules.

1. Assume event 'a' and event 'b' are events in the same process and event ‘a’ occurs before ‘b’, then the notation “a->b” is true.

2. Assume event 'a' is the event of a message being sent by one process and event 'b' is the event of the message being received by another process, then the notation “a->b” is true.
3. Also, if “a->b” is true and “b->c” is true, then the notation “a->c” is true.

2.5 Android Architecture

Android for mobile device contains an operating system, middleware layer and some necessary applications.

Fig. 4 Android Architecture
(http://www.satyamsoft.in/Android/androidarchitecture.html)
2.5.1 Applications

This level contains the user programs or applications such as email, SMS, calendar, web browser, etc. For applications, programs cover the full screen and interact with the user.

![Applications and Widgets](image)

Fig. 5 Android Applications and Widgets

2.5.2 Application Framework

This layer provides high level building blocks, which can be used by developers to create new applications. Some of the most important parts of this framework are listing below.

1) Content Providers can enable processes to share their data.

2) Window Manager is used to take charge of user interactions.

3) Activity Manager is used to control the life cycle of applications.

![Application Framework](image)

Fig. 6 Android Applications Framework

2.5.3 Libraries and Android Runtime

There are shared libraries written by C/C++ which can be used by different programs. Some of the native libraries are list as following.
1) C library—a standard C library (libc) inherited from BSD, and it is customized for embedded Linux system.

2) Surface Manager—a compositing window manager which provides seamless integration for 2D and 3D image layers.

3) SQLite Database—a database engine which can be used for sorting persistent data in all applications.

Dalvik Virtual Machine is used to run the applications on Android. Each application has a Dalvik virtual machine instance. Dalvik VM runs .dex files which are optimized for small memory.

![Android Libraries and Android Runtime](image)

2.5.4 Linux Kernel

The most important operating system services of Android system rely on Linux Kernel, such as security, memory management, process management and networking. In the meantime, Linux kernel can be considered as a hardware abstraction layer between hardware and software stack. However, programs will not directly make Linux calls. They always use the Dalvik VM.
The DVM uses different kinds of libraries for the Android system. Some of the libraries include a variety of C/C++ libraries that are used by various components inside the Android system. Each Android Application runs its own process with the use of its own instance of the DVM (Tim Lindholm, Frank Yellin).

The Dalvik executable (.dex) format is written in a way that it can run multiple VMs more efficiently which is optimized for minimal memory footprint. The VM runs classes which are compiled by a Java language compiler which transforms it into a .dex format. The tool is provided and
the VM itself is register-based. Threading and low-level memory
management functionality is depended on the Linux kernel. The kernel is
also being able to use as an abstraction layer between the hardware and
the remaining software stack.

2.6 Android Application Architecture

Generally speaking, Android applications are constructed with four
components, 1) Activity, 2) Broadcast Receiver. 3) Service. 4) Content
Provider. Broadcast Receiver takes charge of responding to any
unpredictable events like incoming call or incoming web data. Service is
one of the most important components of Android application which the
thesis will talk about it in 2.7. Content Provider is the data base or data
sharing related component. Activity is the core component of every
application and will be explained in detail in below.

An activity is a single, forced thing that the user can do. Almost all
activities interact with the user, so the Activity class takes care of creating
a window for you in which you can place your UI with “View”. While
activities are often presented to the user as full-screen windows, they can
be used in other ways: as following windows or embedded inside of
another activity. There are two methods almost all subclasses of Activity
will implement

onCreate: is where you initialize your activity.

onPause: is where you deal with the user leaving your activity.
2.7 Android Client-Server Architecture

Android’s Client-Server structure has a general communication schema between client end and server end.

At the initial procedure of Android system, Android will launch some significant system processes. In this section ‘service manager’ and ‘zygote’ are two system processes which play an inevitable role in the whole Android C-S (Client-Server) architecture.

Service manager is the daemon process of Binder service (Android IPC mechanism) that is to say, all the system services will register themselves in the service manager and the service manager provides necessary interfaces to the clients (applications) and manager the ‘binder’ mechanism as well.

Zygote, as the role of the father processes of all system services and applications, takes charge of creating all the system demand and user demand services/applications. In other word, zygote ‘fork’ itself every time, when system or user needs a new service or application. One other point which is worthy to be pointed out here is, during the ‘fork’ procedure, the DVM (Dalvik Virtual Machine) will be copied as well. Therefore, this is a more efficient way to create a new DVM for a new application rather than create a new DVM from a scratch.

The whole procedure can be explained as the figure below.
As the figure illustrated above, whenever an application wants to interact with a specific system service, where the application actually invokes is the service manager. The service manager provides an easy API (application programming interface) for application to use and it will manage the IPC (inter processes communication) mechanism itself, i.e. ‘Binder’ in Android.

Fig.10 Android Client Server Architecture
The next figure shows part of the system services in Android system,

<table>
<thead>
<tr>
<th>Service</th>
<th>Service</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entropy Service</td>
<td>Device Policy</td>
<td>Audio Service</td>
</tr>
<tr>
<td>Power Manager</td>
<td>Status Bar</td>
<td>Headset Observer</td>
</tr>
<tr>
<td>Activity Manager</td>
<td>Clipboard Service</td>
<td>Dock Observer</td>
</tr>
<tr>
<td>Telephone Registry</td>
<td>Input Method Service</td>
<td>UI Mode Manager Service</td>
</tr>
<tr>
<td>Package Manager</td>
<td>NetStat Service</td>
<td>Backup Service</td>
</tr>
<tr>
<td>Account Manager</td>
<td>NetworkManagement Service</td>
<td>AppWidget Service</td>
</tr>
<tr>
<td>Content Manager</td>
<td>Connectivity Service</td>
<td>Recognition Service</td>
</tr>
<tr>
<td>System Content Providers</td>
<td>Throttle Service</td>
<td>Status Bar Icons</td>
</tr>
<tr>
<td>Battery Service</td>
<td>Accessibility Manager</td>
<td>DiskStats Service</td>
</tr>
<tr>
<td>Lights Service</td>
<td>Mount Service</td>
<td>ADB Settings Observer</td>
</tr>
<tr>
<td>Vibrator Service</td>
<td>Notification Manager</td>
<td></td>
</tr>
<tr>
<td>Alarm Manager</td>
<td>Device Storage Monitor</td>
<td></td>
</tr>
<tr>
<td>Init Watchdog</td>
<td>Location Manager</td>
<td></td>
</tr>
<tr>
<td>Sensor Service</td>
<td>Search Service</td>
<td></td>
</tr>
<tr>
<td>Window Manager</td>
<td>DropBox Service</td>
<td></td>
</tr>
<tr>
<td>Bluetooth Service</td>
<td>Wallpaper Service</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 11 Android System Service List

(http://www.slideshare.net/opersys/understanding-the-android-system-server#btnNext)

Each service has its own responsibility (Karim Yaghmour). For example, “Power Manager Service” takes charge of the brightness of screen and the timing of when system will go locked/sleep. “Network Manager Service” is responsible for DNS setting and internet interface configuration “Window Manager Service” which is the key service to interact with human interaction and it is the master of input event management and focus window management etc.
2.8 Android IPC Mechanism (Binder) Architecture

In order to get rid of the license issue of 'OpenBinder’, Google develops a new inter processes communication structure based on the “OpenBinder” architecture and named it “Binder”. The speed of invoking a function through “Binder” is not that fast as the native invocation but “Binder” makes the system more decoupled and structured (Hackborn, Dianne).

“Binder” has a five level structure.

![Android Binder Structure](image)

Fig.12 Android Binder Structure
(1) Binder Driver

Binder Driver is the place where the real communication happens. It takes charge of the real transportation between processes.

(2) Binder Adapter

Binder Adapter interacts with Binder Driver and help implementing most of the functions.

(3) Binder Core

Binder Core contains two subclass of binder, “BBinder” and “BpBinder” these two classes represent the most basic service end and client end.

(4) Binder Framework

Binder Framework has its own implementation in Java and Native code (C++) respectively. Both of them have the same structure, the client end received the user invoke request and transfer it to the proxy end, the proxy end processes the request and return the results back to the client end.

(5) Binder Client/Proxy

Binder Client/Proxy implements the Binder Framework and registers them to the Service Manager. Therefore, application can invoke Service Manager directly and avoid the complicated Binder invocation.
The next example shows a simple IPC (inter process communication) procedure between an application and a service.

As it demonstrated above, an application (process 1) is invoking a service API which has been registered in the Service Manager. Therefore, the application (process 1) is getting the service API by invoking the Service Manager. In addition, in order to get the service code functioning, the application side is actually calling the Binder Proxy and let the proxy transfer all the data to the service end and the Binder Service will receive them. Based on the data it received, the Binder Service invokes the
function code and returns the corresponding result back to the Binder Proxy. Finally, the whole procedure can be looked like the application is invoking a native code simply and that what the Binder wants to provide.

2.9 Android Socket Architecture

Android provides a special IPC (Inter processes communication) mechanism between Native Layer (C/C++ layer) and Framework Layer (Java Layer). That is socket.

In Android system, on one hand, JNI (Java Native Interface) provides a communication method from framework layer (Java) to native layer (C) and socket, on the other hand, provides a communication method which can send message from Native layer (C) back to Framework layer (Java).

The main structure of this type of socket is demonstrated below

Fig.14 Android Socket
Based on the demand of this debugger, the framework layer is treated as the server side and it will keep waiting the client side (C layer) to send the message. Therefore, the server end will create a new socket thread at the beginning of system initialization and whenever the client need to send a message, the client will open the specific socket which is created by the socket thread running in the server side and send the necessary message.

2.10 Android Event-Driven Architecture

Android is an event-driven operating system, which means everything which need to be processed by the system will be wrapped as an event and send it to a queue and waiting for being picked up.

One of the related events is Android key event. Android Window Manager Service maintains a special queue for these external events (key events) and named it key input queue and every external event will be processed in that queue.
As the figure shows above, the input dispatcher thread of window manager service will take charge of fetching the events from input queue and dispatch them to the focused window, since user behavior will affect the current front window (focus window) only.

Also, in order to make the memory usage more efficient, the input dispatcher will be put to sleep when there is no more events need to be processed in this queue and it will be waked up whenever there is a new user input event coming.

This is only one event-driven example of Android operating system, Android’s event-driven concept can be seen everywhere, it provides the
developer a more clear image when they developing the new applications or services and it makes Android more structured and standardized.

2.11 Android Application's Life-cycle

Fig.16 Android Application’s Life Cycle (http://www.skillguru.com/blog/2011/01/13/android-activity-life-cycle/)
Applications have a lifecycle (Vinay). Lifecycle starts with the beginning when Android instantiates them through to an end when the instances are destroyed. The following table lists the lifecycle methods an application can implement. They symbolize the various states an application may go through.

Various states of Android application

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onCreate()</td>
<td>Called at start of application. Must be implemented by user. The application is in foreground.</td>
</tr>
<tr>
<td>onStart()</td>
<td>Called when activity becomes visible.</td>
</tr>
<tr>
<td>onResume()</td>
<td>Called before user interaction with the application. This is the current application with user input going to it.</td>
</tr>
<tr>
<td>onPause()</td>
<td>Intermediate state. Called when another application is going to be started.</td>
</tr>
<tr>
<td>onStop()</td>
<td>Called when application is not</td>
</tr>
</tbody>
</table>
visible to user. Application goes to background.

<table>
<thead>
<tr>
<th>onDestroy()</th>
<th>Called when application is going to be destroyed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>onRestart()</td>
<td>Called when application comes from background to foreground.</td>
</tr>
</tbody>
</table>

Fig. 17 Android Application Status

The entire lifetime happens between the first call to `onCreate()` through to a single final call to `onDestroy()`. Activity Manager Service is a Java service that manages all aspects of the application lifecycle, and sits on top of the window manager to tell it what to do with the windows coming from various applications.
Chapter 3

RELATED WORK

3.1 General Replay Techniques

Replay has become one of the most popular topics in multiple threaded application area. Many researchers have been relaying on special hardware to reproduce the program behavior. A type of noninterference architectures of monitoring has been developed to collect execution related data at the run time without bothering the executing application itself (J. J. P Tsai, et al). Therefore, if a failure occurs, the computed instruction counts are used to force the replay rollback to the failure point. Then make sure the application recreate the failure state. Also a replay mechanism has been designed to control the reproduction of the program behavior. The software instruction counter approach records the backward branches and is used to identify the exact location of the required event occurs.

Another system records the order of accesses based on version counters in Instant Replay(Slye and Elnozahy). In this system, the relative order of significant events is saved as they occur, not the data associated with such events.

3.2 JaRec

JaRec(Georges et al.) aims at portable replay for multi-threaded Java applications. It uses byte code instrumentation to capture and replay
schedules based on Lamport clocks (LC) for objects. It assumes a data-race free program and thus it only needs to instrument the synchronization operations. This technique has originally been used by the same group to implement RecPlay, a tool that combines record/replay and data race detection for Solaris. Instrumentation of the byte code is performed on the fly by the virtual machine. This enables dynamic loading and replaying of classes over a network and prevents having several versions of a class.

3.3 JReplay

JReplay (Viktor, Marcel and Armin) aim to achieve similar functionality as DejaVu does. It is a deterministic replay of multi-threaded applications which forces the application to execute according to a specified thread schedule. Solutions proposed by JReplay are independent of the underlying operating system and of the JVM implementation the program is running on. Replay of a schedule is achieved by instrumenting the original application according to a given schedule and supplying a library containing a replay engine in addition to the instrumented class files to the virtual machine at start-up. No separate thread is created to control replay. Rather, each application thread performs method calls to the replay engine at appropriate times. To achieve deterministic replay, JReplay only allows one thread to run at a time with all other threads blocked. The running thread is specified by the given schedule. To control which thread is currently running, JReplay
assigns a lock object to each thread during the replay of the instrumented program.Threads are blocked and unblocked using these locks and Java synchronization mechanism. In order to transfer control from one thread to another, JReplay unblocks the next thread scheduled to run and then blocks the current thread.

3.4 DejaVu

In the specific area of record/replay of Java programs, the first solution to achieving deterministic replay of executions of multithreaded Java programs was proposed by Choi and Srinivasan (Jong-Deok and Harini). This paper introduces a notion of a logical thread schedule, which is based on counting the number of critical events occurring between thread switch. Critical events in this context are all synchronization events performed by threads, for example monitor enter and monitor exit and shared variable accesses. The approach to capturing the logical thread schedule is based on using global clocks. This clock ticks at every execution of a critical event to uniquely identify each critical event. A local clock per thread is used to allow each thread to identify schedule intervals that belong to that thread. Their implementation is based on a modified Java virtual machine. The approach has been extended to include record/replay of networking events in distributed applications ("Deterministic Replay of Distributed Java Applications").
3.5 Multiple Threaded Replay Debugger

Multiple threaded replay debugger is Android platform based software which aims to replay multiple threaded application by using happened-before concept.

It modified Android virtual machine (Dalvik) and record synchronization events and put them into a log file. During the replay phase, this debugger verifies each event with the log file. Whenever it found a match, the debugger let the event run, therefore it can make sure the application replays the same run with the recoded one and reproduces the same error message again.

However, it cannot handle user input events (user behavior). In other word, any events happen in other threads except the application thread will be ignored. In this case, if the failure state is caused by user misbehavior, it cannot be reproduced at the replay time.

This thesis uses this debugger as the basic formula, modifying DVM (Dalvik Virtual Machine) to record synchronization events. In addition, it modifies Android services and use event driven feature to replay unexpected user behavior. Therefore, the software designed and implemented in this thesis can replay the whole failure run.
Chapter 4

DESIGN AND ALGORITHM

4.1 Design Overview

The core part of recording and replaying phase is the ‘log file’. The main purpose of recording is to generate this log file and as for replaying phase, the main target is to use this existing log file to regenerate the running state.

Record

Recording phase aims to record any necessary information in order to fulfill the demand of replay. During this phase, only relative order and external input (user input information) will be recorded. In other word, only happen-before relationship and detailed external events will go to log file.

Fig.18 Record Overview Architecture
Replay

Replaying phase tries to use recorded log file to schedule system input and thread execution. The replay phase doesn’t generate system input (It does generate user behavior since there is no user inputs during the replay phase). The replay engine schedule them in order to make sure the same execution will be reproduced.

![Replay Overview Architecture](image)

Fig.19 Replay Overview Architecture

This thesis mentioned two types of events “internal events” and “external events”. Given the different features of these two events the main function has been divided into two parts in this proposed method. System input record/replay (internal input) and User input record/replay (external input). This design uses different method to record/replay these two types of events.
4.2 Algorithm Definition

This thesis proposed an ordering based record/replay mechanism. All events that are recorded in the log file are called thread interaction event. These events are denoted by ‘E’. ‘E’ includes synchronization events, thread creation events, message passing events and external events. All of these events will be explained later.

The object which is used to communicate between threads is denoted as ‘O_{id}’ where “id” is the identifier of each object.

Thread is denoted by symbol ‘T’. Also thread id as the identifier of each thread is denoted as ‘T_{id}’.

Also, this thesis defines event sets to group related events. A set TES_{id} (Thread Event Set) is defined to collect all the events happen ‘T_{id}’ and the set to gather all events happened in ‘O_{id}’ is defined as OES_{id} (Object Event Set).

In order to ordering them, the notation of Lamport Clock (LC) is being used.

Assume ‘a’ is an event running on one of the target recording thread. LC (a) is the notation to provide the partial timestamp of event ‘a’ based on the Lamport Clock (LC) principle.

In addition, instead of ‘happen-before relation’ (denoted as “->”), a new notation ‘|->’ is introduced as named it as ‘immediate happen-before notation’. This notation is designed for two consecutive events they happen immediate before (or after) each other.
At the figure above, event 'b' happened right after event 'a' and event 'c' is happened after event 'b'. Therefore, although both 'a->b' and 'a->c' are true, only 'a|->b' is true, 'a|->c' is a false statement in immediate happen-before relationship.

As a result above, if $E_a$ and $E_b \in TES_{id}$ or $OES_{id}$, we have the conclusion that:

For all $E_i \in TES_{id}$ or $OES_{id}$ ($i > a$ & $i \neq b$) and $E_a|-> E_b$

We have

$LC (E_a) - LC (E_i) > LC (E_a) - LC (E_b) > 0$

4.3 Key Event and Key Event Queue

Android, as an event-driven operating system, has lots of events. A key event, as a special type of event, is the event which contains all the information of user input. It structures shows below.
One of the constructor of "key event" is demonstrates at following

```java
public KeyEvent(long downTime, long eventTime, int action,
                int code, int repeat, int metaState,
                int device, int scancode, int flags) {
    mDownTime = downTime;
    mEventTime = eventTime;
    mAction = action;
    mKeyCode = code;
    mRepeatCount = repeat;
    mMetaState = metaState;
    mDeviceId = device;
    mScancode = scancode;
    mFlags = flags;
}
```

The meanings of each attributes are explained below

"downTime": The time (SystemClock.uptimeMillis()) at which this key code originally went down.

"eventTime": The time (SystemClock.uptimeMillis()) at which this event happened.
"action": Action code, either \#ACTION_DOWN, \#ACTION_UP, or \#ACTION_MULTIPLE.

"code": The key code.

"repeat": A repeat count for down events. (>0 if this is after the initial down) or event count for multiple events.

"metaState": Flags indicating which meta keys are currently pressed.

"device": The device ID that generated the key event.

"scancode": Raw device scan code of the event.

"flags": The flags for this key event.

The most important two attributes are "action" and "code". Action decides what the action is. Such as key pressed (down action) or key released (up action).

```java
/**
 * (link \#getAction) value: the key has been pressed down.
 */
public static final int ACTION_DOWN = 0;
/**
 * (link \#getAction) value: the key has been released.
 */
public static final int ACTION_UP = 1;
/**
 * (link \#getAction) value: multiple duplicate keys events have occurred in a row. The (link \#getRepeatCount) method returns the number of duplicates.
 */
public static final int ACTION_MULTIPLE = 2;
```

Fig.23 Key Event Action Definition

The attribute "code" indicates what does the key event means. Such as key code "21" means "keyboard left" and key code "55" means "keyboard comma".
As for key event queue, it is a special event queue that contains only key events.

Key event queue interact with the input dispatch thread which is the thread that takes charge of dispatching all user external events. The architecture of that queue shows below

```java
int mGlobalMetaState = 0;
boolean mHaveGlobalMetaState = false;

final QueuedEvent mFirst;
final QueuedEvent mLast;
QueuedEvent mCache;
int mCacheCount;
```

Fig.24 Key Event Queue Attributes

It has two pointers, one indicates the first event in this queue and one points to the last event in this key input queue.

4.4 Internal Events Record

As explained previously, the ordering based approach of record/replay is being implemented. The record engine generated execution related information log file which will be used for deterministic replay.

4.4.1 Synchronization

Synchronization events can affect the execution order of threads. Therefore, these events are very significant events for the replay
procedure. Android applications are written in Java and Java provides several methods to construct the synchronization block.

Java provides “synchronized” as a statement block and developer can treat everything inside it as synchronized actions. That is to say, whenever a thread access to this block this thread will go into a critical section and whenever it existing this block, the critical section ends.

![Synchronization Block Diagram](image)

**Fig.25 Synchronization Block**

In addition, Java also provides synchronized method. This type of methods use “synchronized” as the keyword to define the function. Whenever a thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object will be blocked until first thread is done with the object.
4.4.2 Thread Creation

Java uses keyword “new” to create a new thread.

```java
Thread thread = new Thread();
```

However, this piece of code will only create a thread object. It won’t actually run the thread. Java needs the code following to activate the thread.

```java
thread.start();
```

Undoubtedly these events need to be record as well, since whenever a new thread creates, if this new created thread is part of the
target recording application, it definitely is going to change the execution order of thread.

4.4.3 Message Passing

Message passing in Android is implemented by a message queue. Every application has one main UI thread and this thread interacts with this message queue. The message queue is maintained inside each application, therefore whenever a thread needs to do a message passing, the thread will send the message to target application’s message queue.

Fig.27 Message Passing

This type of events definitely changes the execution order of thread. Whenever a message passing happened, the main thread has to switch to handle the incoming message. That changes the execution order of threads every time.
4.5 External Events Record

External events known as user inputs, is a special type of events that has to be considered in the whole deterministic replay procedure. Due to the particularity of external events the record schema of external events is vary.

The biggest difference between recording an internal event and an external event is the software needs to record the detailed information of external event since they are not going to be reproduced by the system during the replay phase. Therefore not only the ordering is been recorded but also the event information.

Besides, since all the user input events are handled by input dispatcher thread rather than the target application thread, recording the event information in the DVM (Dalci Virtual Machine) layer is not feasible.

![Fig.28 External Event Record Architecture](image)

Fig.28 External Event Record Architecture
All the user events are stored in the input dispatcher queue rather than the message maintained by each application and this queue is shared by all the user applications. Therefore, the software has to decide what events are needed to be recorded or what events are supposed to interact with the target recording application.

Also, the events happen time of external event is another obstacle of recording user behavior. Since the timer is based on the system time and this time is absolute time, it is not compatible with the proposed ordering based record/replay engine. Therefore, the debugger needs to remember what is the “immediate happen-before” event of these external events. During the replay phase, whenever these “immediate happen before” events happened. The debugger needs to notify the external events replay engine to re-input these user inputs in order to re-produce the user behavior.

4.6 Record Overview

Overall, the whole record phase can be summarized as follow
Internal events will be recorded at the application’s own VM layer, since application threads are easy to be recognized at VM layer. On the other hand, external events have to be recorded at window manager service’s VM since the software has to distinguish whether this input event is belonging to the target recording application in the “window manager service” class. After the classification then the software records all these events into a trace file for replay engine to use.

4.7 Internal Events Replay

During the replay phase, the system will generate those internal events and pass them to the application at the replay time. Therefore, the
replay engine doesn’t need to generate these events. It just needs to order them based on the logged file.

Fig.30 Events Replay Example I

Assume the recorded execution order is “a1 |-> b1 |-> b2 |-> a2”, which is showed as the figure above. During the replay phase, event ‘b1’ comes first then ‘a1’ than ‘a2’ and ‘b2’ is the last one. Therefore, the coming order is “b1 -> a1 -> a2 -> b2”. So the following are what the replay engine will do.

1) ‘b1’ arrived and scheduler verifies it with log file and let the thread B waiting for the pre-requisite ‘a1’.

2) ‘a1’ arrived and scheduler knows it is the first event, scheduler lets thread A continues to run.

3) Pre-requisite satisfied, scheduler makes thread B ready to run.
4) ‘a2’ arrived, based on the log file the scheduler let the thread A block here and waiting for ‘b2’.

5). Scheduler dispatches thread B and lets it run to ‘b2’.

6) Scheduler notifies thread A to run ‘a2’ since all the pre-requisites are satisfied.

![Diagram](image.png)

Fig.31 Events Replay Example II

4.8 External Events Replay

The strategy is using for external events are little bit different from internal events, since they are using different recording methods.

Rerecorded external events include the immediate happen-before event and all the necessary input event information. Therefore, the basic steps are as follow

1) Follow the internal events’ replay procedure.
2) Whenever an immediate happen before event has been replayed, before schedule next event, the scheduler notifies the window manager service.

3) Window manager service received this notification and search for log file for detailed event information.

4) A simulated input dispatcher thread re-creates the user key event and pushes it to the key event queue.

5) This 'fake' thread notifies the 'read' input dispatcher thread and since the user input event has the highest priority. The event got replayed immediately.

6) The scheduler continues to schedule next event as normal.
4.9 Replay Overview

Overall, the replay phase is combined with two parts, one is internal events replay and the other one is external events replay. The following figure explains how the whole replay engines works.
During the replay phase, whenever a critical event comes, the replay application asks scheduler thread whether it is the right event to run. If the scheduler thread say so, the thread unlock this thread and let it run. If it is not the right event which his supposed to run at next, the scheduler thread blocks the current thread and wait until all the conditions satisfied. At the same time, the scheduler thread keeps sync with the log file and remembers all the immediate happen before relation of user external event and when they come, the scheduler thread notify the receiver thread at the window manager service. The receiver thread reads the log file again to regenerate the user input event and push it into the
input event queue. Then the dispatcher thread is notified and starts to deal with the generated user input event. After that the scheduler goes back to run and schedule the next event.

4.10 Communication between C-Level and Java-Level

As the thesis mentioned above, whenever there is an immediate happens before event scheduled, the scheduler thread is going to notify the receiver thread in the window manager service. However, the scheduler thread is in the application’s VM layer which is written in C and on the other hand, the receiver is in the window manager service which is written in Java and these two threads are running at two totally different processes. Therefore, a socket between DVM layer (C-layer) and Application Framework layer (Java-layer) is proposed.
Fig. 34 Socket Architecture between C layer and Java layer

The server end of this socket is in the “window manager service” (Java layer), and the client end is in the application’s DVM (C layer). Therefore, whenever the window manager service is on, the socket server thread is on and waiting for its client to send the message. On the other hand, the scheduler thread is treated as the client end of the socket in the application’s DVM layer. Whenever the scheduler thread found an immediate happen before event of an external user input event, it will generate and send a message immediately and let the server end knows there is a user input needs to be replayed.
Chapter 5
IMPLEMENTATION

5.1 Record Log Structure

As the thesis mentioned before, during the record phase, this debugger will generate a log file which will be used as the verification file while performing the replay phase. This log file maintains all the necessary information which may be needed when the debugger doing the replay.

| 1243 | 1 | 1 | 1183 | 1136282008 | 1241 | 1241 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1244 | 1 | 1 | 1184 | 1136282008 | 1242 | 1242 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1245 | 1 | 2 | 1172 | 1136325352 | 1228 | 1230 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1246 | 1 | 2 | 1173 | 1136325352 | 1244 | 1244 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1247 | 1 | 2 | 1174 | 1136325352 | 1245 | 1245 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1248 | 1 | 2 | 1175 | 1136325352 | 1246 | 1246 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1249 | 1 | 2 | 1176 | 1136325352 | 1247 | 1247 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1250 | 1 | 2 | 1177 | 1136325352 | 1248 | 1248 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1251 | 1 | 1 | 1185 | 1136282008 | 1243 | 1243 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1252 | 1 | 1 | 1186 | 1136282008 | 1250 | 1250 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Fig.35 Log File Example

The figure above shows a simple piece of the example log file, from left to the right each section is, serial number, event type, thread ID, Lamport Clock, event index, thread happen before relation, event happen before relation, event execution status, external event action, external event key code, external event repeat time, external event meta data, external event scan code and external event flag.

Serial Number - It records the serial number of the recorded event. Serial number starts from 0 and is incremented for each event the log file
recorded. Also this number is used to identify happen-before relation as well.

Event Type – It identifies the type of synchronization event. As the thesis mentioned before, recorded events type include synchronization event, thread creation or external user input event.

Thread ID – It is a unique ID for each thread. During the replay it will help the scheduler thread to decide which event should happen next on this specific thread.

Lamport Clock – Lamport Clock is not the same as Serial Number, it calculated based on the Lamport Clock calculation logic which is mentioned before. It helps to record the partial order.

Event Index – Event index can be used to identify whether this event is a synchronization lock or synchronization unlock.

Thread/Event Happen Before Relation – This field helps to record the happen-before relation of the specific event. Therefore, the scheduler thread doesn’t need to calculate it every time.

Event Execution Status – It indicates that whether this event has been scheduled or not. It helps to verify the happen before relation of each event.

External Event Related Data – Action, Key Code, Repeat Time, Meta Data, Scan Code, Flag. These data field are related to the external event. Since the this debugger has to create them during the replay phase, these data are necessary.
5.2 Initialization

The initialization phase can be separated into two parts, first part is DVM (Dalvik Virtual Machine) initialization and another part is System Services initialization.

As the thesis mentioned above, each application has its own DVM (Dalvik Virtual Machine), therefore, there are many things to do when initializing each VM.

1) Logger Thread Initialization

The logger thread takes charge of recording and dumping all the necessary information into the log file.

2) Scheduler Thread Initialization

The scheduler thread is the most significant component of replay engine, which will be initialized during the DVM initialization phase and it will be put to sleep when there is nothing to do with replay.

3) Record and Replay Related Data Structure

Record data structure will be initialized during this time also and replay related data structure will try to read replay related data from the log file into the memory which will improve the efficiency performance of the whole replay procedure.
The other part is System Services Initialization. This thesis will use Window Manager Service as an example. When the Window Manager Service is initializing, it will create the message queue, the input reader thread and the input dispatcher thread etc. Besides, in order to satisfied the communication demand from framework layer to native layer, a new socket thread will be created also and it will finish its job to open a new socket and waiting for the coming message.
5.3 Internal Events Record

Since each Android application has its own DVM (Dalvik Virtual Machine) and each synchronization call will be processed in the DVM (Dalvik Virtual Machine) eventually.

Thread creation and message passing can be combined with synchronization events, since actually all of them are using mutex lock. The detailed explanation is demonstrated at follow.

![Diagram of Synchronization Block in C (lock)]

Fig.38 Synchronization Block in C (lock)
As these figures show above, whenever at the Java side, an application calls a synchronization block, it actually calls `dvmLockObject()` / `dvmUnlockObject()` and inside these two function, it will call `pthread_mutex_lock()` / `pthread_mutex_unlock()`. Therefore, the logger thread which is running at the DVM (C layer) layer will update the LC (Lamport Clock) whenever an event calls the synchronization block and record necessary information into the log file.
5.4 External Events Record

External events’ recording is based on the concept of internal events recording. In order to be processed by Android operating system, external events have to be pushed into the key input queue. Before putting these types of events into the queue, Android will gain the lock of this queue first, in order to avoid any synchronization racing issue. Therefore, DVM (Dalvik Virtual Machine) will know whenever there is a new event coming to the queue.

As the figure shows above, whenever user gives a new input event, the DVM (Dalvik Virtual Machine) will know and the logger thread which running in the DVM layer will log these events also and update necessary information as well.
5.5 Internal Events Replay

Replay phase has the same concept of recording phase. During the replay time, whenever synchronization events (thread creation, message passing etc.) happen, they will go back to DVM (Dalvik Virtual Machine) too and the scheduler thread will verify the coming event with the log file.

Fig. 42 Internal Events Replay Implementation Architecture

If this event should happen next, the scheduler thread will let the current thread continue to run, if it shouldn’t happen next, the scheduler will use condition wait to put the current to blocked and waiting for all prerequisites satisfied.
5.6 External Events Replay

The replay schema for external events is extended from the internal events replay which is explained above. Since during the replay phase, user interaction will not be generated by the system, the proposed debugger has to create them using the log file. Therefore, whenever an immediate-happen before dependency is satisfied, the scheduler (i.e. socket client) will let the socket server knows which is running at Window Manager Service currently.

![Diagram of External Events Replay Architecture]

Fig.43 External Events Replay Architecture

After the socket server received coming message, it will create a new “fake” user input event and push it the input event queue. After that, the socket server thread will wake the input dispatcher thread up and told it to process the queued message.
Chapter 6

COMPARISON AND CONCLUSION

The most valuable comparison for this proposed debugger should be with the multi-threaded debugger which was mentioned in Chapter 3.

These two debuggers implement the idea of record / replay and use partial order concept. They have similar internal events replay schema and similar record / replay log file structure. However, the biggest difference between these two debuggers is this proposed debugger can record / replay user behavior, which is a practical and necessary feature of multi-thread debugger. Since user behaviors always interrupt the current running thread, which will affect the order of thread execution order definitely, user behaviors have higher priority in record / replay schema.

Compared with other replay tools, the proposed debugger has the smallest amount of log files, which will reduce the system memory consumption undoubtedly since it uses LC (Lamport Clock) and happen-before relation to record the partial order instead of the physical order. Furthermore, the proposed multi-thread debugger is implemented in Android platform, which means this debugger can be ported to any platform which has the same design concept with Android (written in Java, C-S architecture and event-driven).

In conclusion, this debugger tool will not replay the application by physical time, which means these two runs (recording one / replaying one) may not remain exactly the same. However, what this debugger tool can
guarantee is the execution order of thread. This proposed debugger can
replay the application based on the partial order, that is to say, if a multiple
thread synchronization race or a user behavior caused synchronization
error leads to a failed run, the run will be reproduced during the replay
phase. Since the debugger is using the LC (Lamport Clock) and happen-
before concept, the execution order of thread between these two runs will
remain the same.
Chapter 7

FUTURE WORK

The record and replay method this thesis proposed are based on the client-server architecture and event-driven concept. At this stage, the software is implemented in “Window Manager Service” and in applications. In future work, based on the proposed approach, the debugger can be extended to other services as well. For example, the debugger can modify “Internet Manager Service” and in this case, all events interacting with “internet service” can be record and replay.

Besides, this application is implemented in Android platform. It is possible that the software can be abstracted into more platforms, as long as they follow the client-server architecture and adopt the event-driven concept.

GUI is another enhancement of this debugger. This debugger is using ADB (Android Debug Bridge) as the debug tool and it’s based on Android operating system. Therefore, a GUI for the proposed debugger can show the property of multiple threads and concurrent execution, and allow user control of the order of thread execution during replay. It will improve the usage of this proposed debugger tool of multiple threaded applications.
REFERENCES


Google. "Controlling the Embedded Vm". 
<http://android.git.kernel.org/?p=platform/dalvik.git;a=blob_plain;f=docs/embedded-vm-control.html;hb=HEAD>.


Oracle. "Java Virtual Machine Profiler Interface (Jvmpi)". 
<http://download.oracle.com/javase/1.4.2/docs/guide/jvmpi/jvmpi.html>.

Wikipedia. "Virtual Machine". 


