Reducing Fate Sharing in Software Systems via Fine-Grained Checkpoint and Restore

by

Pu Dong

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Dr. Lin Zhong, Chair
Professor,
Electrical and Computer Engineering and
Computer Science

Dr. Behnaam Aazhang
Professor,
Electrical and Computer Engineering

Dr. Dan S. Wallach
Professor,
Computer Science and Electrical and
Computer Engineering

Houston, Texas
December, 2016
ABSTRACT

Reducing Fate Sharing in Software Systems via Fine-Grained Checkpoint and Restore

by

Pu Dong

The client-server architecture is used widely throughout complex software systems, but is susceptible to the problem of fate sharing between its various internal components. Fate sharing is bad because the malfunction of the server can lead to incorrect behavior of the client. We have narrowed down the primary cause of such fate sharing to state spill, a phenomenon in which a server entity holds client-specific state after serving the client’s request, tightly binding the two entities’ fates. The problem is exacerbated when multiple clients utilized the same server, because it effectively binds all clients’ fates together as well.

In this work, we propose Drill, a solution that mitigates the effects of fate sharing in server-like entities by using fine-grained checkpoint and restore (C/R) techniques to reduce state spill. We describe the design of Drill within the context of Android system services — entities that control most system resources and act as middlemen between the kernel and applications — because they are a representative example of server entities that suffer from fate sharing due to state spill. Drill attaches its C/R module to the system service to checkpoint and restore internal clients’ states in a per-object fashion. This module is service-agnostic and non-intrusive, which is generic and can be used for many system services without much modification. A
special bookkeeping service preserves each service’s checkpointed states in an external storage area so that it can resend the states to a new service instance post-crash.

To demonstrate the effectiveness of our approach, we implement Drill on a Google Nexus 5 phone running Android 6.0.1 and apply our C/R technique to two different classes of Android system services. We show that Drill can successfully restore a failed system service to its pre-crash state, keeping the application blissfully unaware of any service crashes because its fate is decoupled from that of the service. Our results indicate that the performance overhead and service downtime of our approach are affordable and that the limitations of the Drill design do not restrict it from being applied to other systems beyond Android.
Acknowledgment

I sincerely show my gratitude to my advisor, Dr. Lin Zhong. His enthusiasm and persistent pursuit of high quality research has inspired me to form research thinking.

I am deeply thankful to Kevin Boos, for his consistent guidance with high level critical thinking as well as the low level help with small technical problems. He will always be the one ready to help while letting me learn the knowledge by myself. This thesis will barely be possible without him.

I am also grateful to Min Hong Yun, for his patience with my questions about Android, Java and C++ in a daily, or even hourly basis. I may not survive without your help. I also thank Dr. Robert LiKamWa who showed me the greatness of being a researcher. I thank Jie Liao, Abeer Javed, Yunhui Hou, for our discussions and time spent together talking about life.

Last but not least, I would like to thank my parents and my friends, especially Eric Liu. I will probably not make it without your help.
## Contents

Abstract ii  
List of Illustrations vii  
List of Tables ix  

1 Introduction 1  

2 Fate Sharing Causes Real Problems 6  
2.1 Fate sharing in complex software systems 6  
2.2 Fate sharing in Android due to state spill 8  

3 Drill System Design 11  
3.1 C/R module encapsulated in a class 11  
3.2 C/R module (de)serializes major types of states 12  
3.3 Central state storage server for state preservation 13  

4 Drill on Android 15  
4.1 C/R module class hierarchy 15  
4.2 C/R module: (de)serialization front-end 16  
4.3 (De)serialization back-end: Parcel and Parcelable interface 18  
4.4 Function-state and data-state: what to/not to C/R 22  
4.5 system_server refactoring and self-healing ability 23  
4.6 Source-to-source code transformation automation tool 24  

5 Evaluation 26
Illustrations

1.1 Overview of client-server architectures. (a) is the traditional client-server architecture. (b) is the microkernel OS’s service model with the state storage server. (c) is Drill’s architecture, adding a C/R module to the server. 

2.1 Chromium Issue Tracker id=487872. (A), (B) and (C) show user’s interaction with the browser and state change in the Chromium’s process architecture. (D) is the patch to fix the bug. We adopt the diagrams from [1].

2.2 Two user applications spill their states to NotificationManagerService.

2.3 Android system service issues. (A), (B) show issue 212780 and (C), (D) show 210210.

4.1 Overview of the Drill implemented on Android operating system. Only related system components are shown here. Components we implement are highlighted in grey.

4.2 Class hierarchy. System service classes have two major inheritance paths but they all eventually inherit from the C/R module – ChkptRstr AddsOn class.
4.3 A detailed look of C/R module. States are serialized to/deserialized from a Parcel object. We implement the front-end state iterator and augment the front and back-end Bundle and Parcel classes to support more types.

4.4 Object (de)serialization sequence using the default Parcel (top) and augmented Parcel (bottom) classes respectively. Assume the object is defined in a tree like structure and the serialization method uses pre-order traversal.

5.1 Performance overhead – time to checkpoint the states for a system service. Both services’ states are checkpointed in a freshly booted system, no user interaction with the phone.

5.2 Down time – time to restart the service process w/o restoring the most recent states.
Tables

5.1 Behavior of CS and AMS before and after restoring the states when restarted from a crash.
Chapter 1

Introduction

Software systems have continuously grown in their complexity with more functionality since the first appearance of computers. For complex systems like operating systems (OS) or middleware, one important aspect of such system’s functionality is to manage the internal resources such as software buffer pools and hardware clocks. It is often a natural design choice for developers to model the system after the client-server architecture.

The client-server architecture draws a line to split the responsibility of resource access and resource management, but it is still susceptible to the problem of fate sharing. In such an architecture, the client accesses system resources via service requests to the server. And the server is in charge of the resources. It fulfills the service request on behalf of the client and return the results to the client if there are any. Although this architecture seems to form a cooperative relationship between the two parties, the client is actually dependent on the server. It is not able to fulfill the service request on its own if the server fails because of malfunction. The dependency of one software entity on another one in terms of functionality is called fate sharing. It essentially binds the fate of the client to that of the server. Fate sharing will become a problem if the server entity can not successfully serve the service request of the client entity.

We identify state spill as the primary cause of fate sharing. State spill will endanger other client entities and also bind their fates together. A state is defined as a
software object that carries certain information. State spill is a phenomenon in which
the server entity creates and holds client-specific objects or the server modifies its
own objects while serving the client’s request. For example, a timer server is able
to receive a timer service request from a client. It will create and hold a state which
contains information including when to set off, what to do and which entity to notify
after expire. Most of the time, the server is designed so that more than one client
can utilize it. State spill will worsen the fate sharing problem in this case because the
server now holds states of multiple clients. A server’s malfunction, such as a crash
when serving one client’s service request, will leads to a loss of all other clients’ states.
As a consequence, their service requests will not be fulfilled, which indicates that all
clients’ fates are bound as well. The state spill in traditional client-server architecture
is illustrated in Figure 1.1 (a).

The fate sharing problem can be mitigated if the state information in the server,
especially the client-specific states, can be preserved over a server crash and recovery.
This is because the lost states can be restored so that the server is able to continue
serving clients’ requests, which makes the server’s malfunction transparent to the
clients. The fates between client and server, as well as among clients, are effectively
decoupled. Although it is not stated explicitly in prior work to solve the fate sharing
problem, this approach has been studied in some microkernel OSes such as Minix
3 [2] where servers can send their state information to a special state storage server
in the memory.

In this work, we focus on existing complex systems which suffer from fate sharing
problems and by design do not have the ability to preserve the states. In the literature
and real systems, preserving the states in general requires checkpoint and restore
(C/R). Normal C/R techniques [3–6] usually checkpoint and restore the entire process
or a group of processes. They can certainly preserve the server’s states but they are too coarse grained. In some cases where multiple servers share one process and we only want one server’s states to be restored, the coarse C/R technique is not flexible enough to mitigate fate sharing.

Therefore, we must use the checkpoint and restore (C/R) technique in a much finer granularity. In this work, we present Drill, a structural modification to the client-server architecture in existing systems via fine-grained C/R to reduce fate sharing. The specific C/R method we choose is serialization and de-serialization.

Figure 1.1: Overview of client-server architectures. (a) is the traditional client-server architecture. (b) is the microkernel OS’s service model with the state storage server. (c) is Drill’s architecture, adding a C/R module to the server.
because it can checkpoint and restore the states in a per-object fashion. In Drill, we augment the server with a C/R module which is able to examine, extract and put back the states, especially the client-specific ones. We also have a state storage server for remote state preservation. In Figure 1.1 (c), we show the new client-server architecture we propose with the C/R module. We also show in (a) and (b) the client-server architecture in normal systems and microkernel OSes to distinguish our contribution. We are interested in how to transform such a system with as small amount of effort as we can to reduce the fate sharing problem.

The C/R module is the central part of Drill. Based on our knowledge, we have identified several challenges in designing the C/R module, which we will answer in this paper.

- **States access:** the C/R module has full access to all the states in the server and can identify the states to be preserved.

- **Non-intrusive:** we work in the source-code level. It is preferred that C/R module is non-intrusive and it affects no existing code.

- **Generic:** as one system may contain many servers of different types and with different states, the C/R module needs to be generic and work with the majority of them.

- **Special states handling:** special states such as objects for inter process communication (IPC) are different across platforms and they need to be taken special care of because (de)serialization methods often only work with generic objects.

To test the feasibility of our approach in real systems, we implement it on Android with moderate structural modification to system services, framework libraries and
very little in Linux kernel to support Drill. We show that it is feasible to attach an appropriate C/R module to an Android system service using (de)serialization method in Android.

We evaluate the effectiveness of Drill by conducting stateful recovery experiments on two system services. Results show that Drill can successfully recover the states of a failed system service and keep the crash transparent to the application. With unnoticeable C/R time and bearable down time, we demonstrate the usability of (de)serialization in the real system. We also identify the limitations along with our development, which we show do not essentially prevent Drill from being used to other systems.

The rest of the paper is organized as follows. We show that the fate sharing problems do exist in the real systems with some concrete examples in Section 2. The subsequent sections are the design of Drill and the actual implementation. In Section 3, we show components that Drill needs to transform a general client-server system to reduce fate sharing. Section 4 describes the implementation with details where we make Drill work with Android 6.0.1 on a real device. This section also gives a general idea how much more needs to be done for a real system. We then show two specific examples as our case study for evaluation in Section 5 and talk about the limitations of the approach along with the development. We then discuss related work in Section 6 and finally draw the conclusion in Section 7.
Chapter 2

Fate Sharing Causes Real Problems

In this section, we will show some evidence of fate sharing first in complex software systems in general and then in Android OS in particular. We also study the first 1000 crash reports on AOSP Issue Tracker and gather evidence for fate sharing.

2.1 Fate sharing in complex software systems

Commodity monolithic operating systems suffer heavily from fate sharing because there is quite little separation. Applications and kernel share the same fate because the kernel is monolithic in a way that not only the core components (e.g. scheduler, memory management) but also the extensions (e.g. buggy drivers) live in the same kernel space. It is a well known phenomenon that bugs in kernel extensions will highly likely bring down the entire kernel, which poses a huge threat to the system reliability.

In cases where an extension faults in the middle of an interrupt, evidence [7,8] shows that the Linux kernel will most likely crash as a whole. In addition to such faults in the interrupt, Windows thinks of extension faults because of syscalls also as fatal, which sadly halts the kernel as well [8]. Besides, easy error propagation [9] within the single entity of the monolithic kernel severely worsens the fate sharing problem.

Chromium web browser is another example of complex software system that embodies fate sharing caused by state spill. Its multi-process model consists of one browser process and multiple renderer processes. For renderer processes, it supports
Four specific models which are process-per-site-instance, process-per-site, process-per-tab and single-process [1]. They are presented in a descending order of renderer process isolation, which causes increasing fate sharing issues. However, a dive into the system unveils deeper fate-sharing problem by state spill. The main browser process creates internal objects called RenderHostView to represent information of individual renderer process in a centralized way. A renderer process’s change to its own RenderHostView in the browser process may affect other renderer processes’ RenderHostView’s visibility. So in some corner cases, the visibility of a process can be wrongly set or unset by another renderer process, causing behavior incorrectness. Issues with id 487872 [10] and 638375 [11] in the Chromium Issue Tracker shows that the RenderHostView is hidden when committing a navigation while the local renderer process is still trying to use it, which confirms the existence of the problem. Figure 2.1 illustrates this problem.
2.2 Fate sharing in Android due to state spill

System service layer is where fate-sharing problems occur the most in Android OS besides the Linux kernel underneath. System services lie between applications and the kernel and work as the central system-wide resource providers. Applications issue requests to the system service by an Android specific IPC system called Binder. Typical system resources managed by various system services include windows, credentials, sensors and so on. Interaction between application and system services or among system services models the client-server architecture discussed in the previous section.

Android enforces that all system services share the same life span. Except for several native services such as SurfaceFlinger, all services are implemented in Java within one single process called system_server. Major reasons for this design choice include small RAM footprint and lower communication overhead [12]. However, this design effectively binds the fates of all services together. A crash in any single service will bring down all other services at the same time (i.e. system_server). What makes it worse is that the parent of system_server, Zygote process, will also kill itself if it gets notified of the death of its child, system_server, which will kill all the user-space Java applications as well. This is a chain of massacre.

Besides the enforced fate-sharing mechanism just discussed, state spill is quite common among applications and Android system services. A good example at hand is NotificationManagerService. It maintains an internal list of all notification listeners. Whenever there is a change in the notification, it goes over the list and finds out whom the notification belongs to. The two user applications which register themselves in such services are org.cyanogenmod.themechooser and the status bar in com.android.systemui, shown in Figure 2.2. Therefore, these two ap-
public class NotificationManagerService extends SystemService {
    private NotificationListeners mListeners;
    ... 
    @Override
    public void registerListener(final INotificationListener listener, final ComponentName component, final int userid) {
        enforceSystemOrSystemUI("INotificationManager.registerListener");
        mListeners.registerService(listener, component, userid);
    }
    
    @Override
    public void setNotificationsShownFromListener(INotificationListener listener, String[] keys) {
        ... final ManagedServiceInfo info = mListeners.checkServiceTokenLocked(token);
        ... }
    }

Figure 2.2: Two user applications spill their states to NotificationManagerService.

Applications are effectively bound together. Another more obvious example is the ActivityManagerService. It works as a central mediator for various activities in the system. It keeps a list of all broadcast receivers which are either user applications or system services. It receives and dispatches the notification in a broadcasting fashion. As for state spill problems like these, we have found concrete evidence.

Android system developer community reports system crashes due to states shared among system services and applications. For the 1000 most recent crash reports, we filter them using key word to collect those related to system_server crash. Then we manually inspect every report to confirm its relevance and finally find 7 reports
Figure 2.3: Android system service issues. (A), (B) show issue 212780 and (C), (D) show 210210.

which describe system service malfunction. Two examples among them are issues with id number 212780 [13] and 210210 [14]. The former one reports the failure of the PackageManager service because of a null pointer access to a state in the ActivityManagerService. The latter issue is about an unsynchronized states between application and NetworkPolicyManagerService, which as a result leads to a null pointer dereference in NetworkPolicyManagerService. Both cases are caused by bad states which are null in the service spilled by another entity and eventually they lead to the catastrophic failure of the system_server, Zygote and other applications.
Chapter 3

Drill System Design

In this section, we describe the Drill design in detail. We present the two major components: C/R module and the state storage server, and how they are integrated in a normal system. Along the description, we also elaborate how we address the several design challenges.

3.1 C/R module encapsulated in a class

We encapsulate the entire C/R functionality of the module in a wrapper class. This approach helps us address the state access challenge and achieve the non-intrusive property of the module mentioned in the Section 1.

States access. Making the server’s class inherit from the wrapper class gives the C/R module the full access to the states in the server entity. In a server class, the full set of the states is formed by all the fields/member variables of that class depending on which programming language it is implemented in. State access consists of two key factors – the visibility of all the fields and the ability to inspect them. The full set of the states (public states and private states) in the a server class is only visible if the C/R module is part of the class. As the C/R module is inside the class, the ability to inspect its own fields is another programming language feature called self reflection. The C/R module also leverages this feature to know the existence of the states. Therefore, combining the two factors, we package the C/R module in the
wrapper class and force the server class to inherit from it so that the C/R module is able to access all states. For actual state access, the C/R module contains an iterator that loops over all the fields at runtime. As was just discussed, we indeed rely on the inheritance in modern object oriented programming and the reflection feature, which means it is going to be hard for Drill to work on systems written in C and C++. However, the concept to wrap the C/R module in a class is still applicable.

**Non-intrusive.** The non-intrusive property is also achieved by encapsulating the C/R module in the wrapper class. Drill works in the source code level where it is quite easy to mess things up. Injecting C/R code directly to the server’s class is error prone and prohibitively verbose. Therefore, this is the second reason why we use a wrapper class to confine the major C/R code in the class. This leads to a clean design and It is also convenient if changes have to be made to the C/R or new functionality is going to be added.

### 3.2 C/R module (de)serializes major types of states

(De)serialization is the key part that makes the module does fine-grained C/R. As we care about states, we decide that (de)serialization is a good choice because it can work in a per-object fashion. There are two metric to evaluate a serialization method: the ability to handle generic states and special states. A good (de)serialization method should be able to handle major generic types of states and supports special states which are related to IPC.

**Generic states.** Choosing the right (de)serialization method which works with major types of states is a necessary condition if we want the C/R module to be generic and service-agnostic (i.e. applicable to any server class without much modification).
Being generic is desirable as we can avoid manual inspection of each server class. However, not all the states, especially not the complex ones, can be serialized easily. Although there exist libraries [15–17] that do generic object (de)serialization, it is not always guaranteed that the recreated objects via restoring are still meaningful. An example is a singleton object in the system. If the state of our interest keeps a reference of such object, creating another such object will obviously be wrong. Therefore, Drill requires a proper (de)serialization method fit for the particular systems that maintains the meaningfulness of the states.

**Special states.** The heterogeneity of the systems has another requirement that the (de)serialization works for special states such as IPC related ones. In a system where multiple servers exist, IPC related states are quite common in a server class because servers talk to each other as well. Different IPC implementations [18, 19] have their own ways to create and recreate the IPC related objects. We need to respect the heterogeneity and maintain the meaningfulness of such objects as well. One thing worth our notice is that being generic to support the major types of states and being able to handle special states are discussed within the scope of a particular system. Drill does not aim to find a cross platform (de)serialization method.

### 3.3 Central state storage server for state preservation

Like the one in the microkernel OS shown in Figure 1.1 (b), we also take the central state storage server as a necessary part in Drill design. This server’s job is to wait and listen to other servers. It receives the serialized states from other servers and send them back to their owner if requested. However, as it contains states from different servers, it is an obvious state spill example, which seems to be a catch 22. The risk
of making state storage server a state spill source can be dramatically reduced if such a server is implemented to be simple and well tested. Similar systems like Minix 3 which also has a state storage server confirms the simplicity of such server [2]. Or one can choose a distributed design to let the server manage their own checkpointed states in a global storage like file system. We choose the state storage server approach mainly because the special states such as file descriptors IPC related objects may not be easily serialized to the disk. Backing up those states and recreating them most of the time requires special mechanism from the system. Take file descriptor as an example, blindly reusing the old integer value of the file descriptor is meaningless. Instead, the right way is to reopen the file when restoring the state, which is likely to be supported by the IPC mechanism already. Therefore, we decide to leverage such mechanism and back up the states remotely in the memory instead of to the disk. In addition, serializing to disk takes more time compared to keeping them in a remote state storage server in memory. In systems where performance is a key factor, a state storage server design is preferred as well.
Chapter 4

Drill on Android

We next present an implementation of Drill on Android system, particularly in Android system service layer. Our implementation is based on Cyanogenmod Android 6.0.1 on Nexus 5 cell phone. Figure 4.1 illustrates the implementation. Compared to the augmented client-server architecture in Figure 1.1 (c), there are new intermediate classes related to (de)serialization in Android – Parcel and Bundle classes.

In a real system like Android, the structural modification to make Drill work with Android not only lies in the C/R module wrapped around the service and the state storage server but also in customizing the (de)serialization method. We show that the modification and customization is moderate compared to such a complex system.

4.1 C/R module class hierarchy

The actual C/R wrapper we implement is ChkptRstrAddOn class. For any Android system service class, it either inherits from the SystemService class or from its own IPC wrapper class which further inherits from the Binder class. As most android services are written in Java programming language and Java only supports single inheritance, we make the ChkptRstrAddOn class as the parent class of the two. This effectively attach each system service with a C/R module. The class hierarchy among the classes is shown in the Figure 4.2. The service is left almost untouched and the C/R function becomes an extension of the service, which is the reason why we name
the class after an add-on.

4.2 C/R module: (de)serialization front-end

Each module contains a front-end state iterator and a back-end serialized state package. We implement the C/R front-end iterator to examine the type of each state with the help of the reflection ability and annotation feature of Java programming.
Figure 4.2: Class hierarchy. System service classes have two major inheritance paths but they all eventually inherit from the C/R module – ChkptRstrAddsOn class.

language. Based on the type of the state, it will make the corresponding method call provided by the Bundle class and further serialize the states into an back-end state package whose type is Parcel class. The state transfer flow is illustrated in Figure 4.3. Our state iterator is built on top of the Bundle class which by default supports 27 types/classes of states while the Parcel class supports 43. The majority of the classes are primitive classes (e.g. Integer) and container classes (e.g. ArrayList). User defined classes are all handled with one method in the Bundle class but they
have to implement a **Parcelable** interface class. We will talk about the back-end Parcel and **Parcelable** interface class in detail in the next paragraph. The reason for the difference between the two numbers is that **Bundle** is only a wrapper of the **Parcel** for high-level use cases, which does not need all **Parcel**'s ability. To support more types of states along with our development, we augment the existing **Bundle** and **Parcel** classes by adding 8 and 6 methods to them respectively so that the C/R module can handle **LinkedList**, **SparseBooleanArray**, **SparseLongArray**, etc.

### 4.3 (De)serialization back-end: Parcel and **Parcelable** interface

**Parcel** class and **Parcelable** interface class are a good choice of the serialization method for Android Java service layer. Other methods do exist including the default Java **Serializable** interface class and third party Java libraries such as Kryo [15] and Xstream [16]. However, there are two major reasons why we stick with the **Parcelable** interface. Firstly, an Android system service often contains special states for IPC communication using the specific IPC mechanism **Binder**. Smart objects such as file descriptors and IPC interface objects like an **IBinder** object could be passed from one process to another. The Android framework layer and the **Binder** driver in the Linux kernel will work together to regenerate the new states. The underlying **Binder** kernel driver keeps internal persistent representation of such objects. However, these smart objects can not be handled by generic (de)serialization methods without modification as we’ve discussed above. Secondly, using the **Parcelable** interface allows us to selectively checkpoint and restore objects and avoid infinitely long chain of redundant objects during serialization. This is quite problematic if the state of our
Figure 4.3: A detailed look of C/R module. States are serialized to/deserialized from a Parcel object. We implement the front-end state iterator and augment the front and back-end Bundle and Parcel classes to support more types.

interest is defined in a class within the system service class. In that case, the outer system service entity that contains the state will also be serialized when we use the generic Java Serializable interface. This will definitely result in incorrect behavior and it is not acceptable.

Parcel class together with Parcelable interface class can conceptually transform all classes to let them become serializable. Almost all data-state fall into one of the following three categories: primitive, container and other classes. Primitive types and container classes are common, such as Boolean and ArrayList. The existing Parcel
class has been heavily used by Android since it occurred and it has been able to se/de-
serialize most primitive types and many container classes in system services. For other
classes, they need to inherit from an interface class called \texttt{Parcelable} which defines
several interface functions including a private constructor to be implemented. This
interface defines the order of the fields of the class need to be put into/read from a
\texttt{Parcel} object. Good news is that some existing classes have already implemented the
interface functions and have been ready to be serialized such as the \texttt{PendingIntent}
class. However, to retrofit all classes we need requires prohibitively large amount of
efforts, which is the major drawback of using the \texttt{Parcel} and \texttt{Parcelable} interface
classes as the (de)serialization method.

Albeit being a good choice for states of Android system services, the \texttt{Parcel} class
and \texttt{Parcelable} interface class are unable to detect circular and multiple reference,
which is a common problem in serialization. This is because they are designed almost
solely for fast IPC but not for serialization. Also, objects passed across processes
via \texttt{Parcel} are often small and the developers have the responsibility to ensure there
are no circular reference when passing it. To address this problem, we augment the
\texttt{Parcel} class and come up with ways to handle the circular and multiple reference.

We add the following to the \texttt{Parcel} class. (i) one \texttt{Boolean} flag to enable the
detection feature, (ii) two \texttt{HashMap} hash tables to store fresh objects and detect
secondary reference, one for writing objects to the current Parcel object and one for
reading from, (iii) several helper functions to add special unique object ID (objectID)
and duplicate identifier (duplicateID) we design to the original serialized byte array.

In Figure 4.4, we show how the augmented \texttt{Parcel} class can handle the multiple
referenced and circular referenced objects by inserting special identifiers. We rely on
the fact that for any secondary referenced object, it must have occurred at least once
Figure 4.4: Object (de)serialization sequence using the default Parcel (top) and augmented Parcel (bottom) classes respectively. Assume the object is defined in a tree like structure and the serialization method uses pre-order traversal.

in the previous serialization sequence, which is the reason why we use a counter to record the number when an object first occurs and take it as the objectID. Future reference of the object will refer to the first one back in the sequence. In order to record the object occurrence, we use one hash table whose key is the object and the value is the objectID in serialization. Every time a Parcel encounters a new object, it updates the hash table and inserts the objectID to the serialization sequence. When it sees an object that has appeared before, it puts an duplicateID in the sequence first followed by the objectID getting from the hash table. The deserialization sequence
has the reverse order compared to the serialization and it uses another hash table with the reverse key-value pair.

4.4 Function-state and data-state: what to/not to C/R

In order to simplify our work, we roughly divide the states into two big categories: function-state and data-state. A function-state is an object created in the system service that (i) mainly provides specific functionality or acts as an interface to other or underlying system components, (ii) contains few to no internal states or its internal states are free from being changed. Examples of function-states include signal handler and a random number generator. We treat most states of this type stateless. On the other hand, a data-state (i) mainly works as as data storage and contains information, (ii) is susceptible to change when the service operates. A Batch class which groups adjacent alarms in Android AlarmManagerService represents this category.

For almost all the function-states, we do not C/R them. Instead, we recreate them with the help of the existing mechanism in the service framework. Many system services inherit from the parent class SystemService in the service framework. The constructor and a series of interface functions such as onStart and onBootPhase are available to be called in different phases and events in the life cycle of the system service. For other system services which do not inherit from SystemService class, only the constructor is used. Some function-states mentioned earlier are created in these places, which we leverage to recreate those states in the crash and recovery.

Data-states including client-specific states are what the C/R module mainly works on. The data-states include both the client-specific and the service’s own states. For states that serves both as function-state and data-state, we treat them as data-state. Although grouping the states is a trick to simply our work, the (de)serialization
method still needs to be handle all the state types. We will talk about the limitations in evaluation section.

4.5 system_server refactoring and self-healing ability

As is discussed in the section 2, almost all (more than 80) Android system services cram into one single system_server process. A crash of any service within that process will effectively result in a reboot of Android Java world in a chain effect from the user’s perspective. As part of the implementation, we decide to refactor the system_server, move selected services out of it and run them in a standalone new process, drill_server. However, we do not argue that Android system services should be fully decoupled from each other and become individual processes for the sake of fate sharing. Instead, we refactor the Android Java system services only for proving the concept of reducing fate sharing in such a monolithic subsystem.

Android has one native C/C++ world and one Java world. We modified the booting process of Android Java world and leave the other world untouched. Zygote is the initial Java process and it forks itself to create the system_server. As drill_server is similar to the system_server in almost every aspect, in the source code level, we strip down the the system_server source code by removing all except the selected services and let it become our drill_server. We then add the code to start drill_server right after the system_server in the Zygote’s source code.

Android system services have dependencies when system boots, which results in synchronization problems after refactoring. Based on their priority, system services are roughly categorized into bootstrap, core and other services and they are started in such sequence. Refactoring one service out from the system_server most of the time requires we respect the starting order but it entirely depends on the service
to be refactored. One example is that if a refactored service needs to acquire a wakelock from the PowerManagerService when being started, it has to wait until system_server finishes starting the PowerManagerService.

We equip the drill_server with the self-healing ability so that when the service crashes, so it could be brought back to live automatically. To obtain this feature, we modify the signal handler in the Zygote process. When it detects the death of its child drill_server, it won’t kill itself like the way it does for system_server. Instead, we make Zygote recreate it. This feature works as a primitive version of the reincarnation server in the Minix 3 [2]. It is sought after by us initially for easy debugging but finally we find it is useful for concept proving purpose.

4.6 Source-to-source code transformation automation tool

Drill’s ChkptRstrAddsOn class works with all primitive and container types of states. For other types, they have to inherit from the Parcelable interface class, which is the major code transformation needed in the source code level. We implement a generic code transformation tool that currently only support to transform internal classes or classes within the same file of the service class and make them become Parcelable.

It should be used in the following steps. First, prepare the service source code file with states already annotated using @States. Second, feed the source code to the transformer in Eclipse IDE and run the transformer. Third, examine the new source file of the service from the output. Substitute the classes needed in the old source file by those transformed in the new one. Finally, recompile the subsystem in the Android source, debug slightly and reload this subsystem to the cellphone.

We leverage an existing Java source-to-source code transformation framework named Spoon to build our tool. Internally, the tool scans the fields (i.e. states)
of the class file and identify those with annotation. It then determines the actual class type of the field (if it is a container type of class, iteratively get the inner class type until find the actual class type). If the actual type is not primitive type and has not implemented the `Parcelable` interface, our tool locates that class’s definition. It finally reads every field of such class and uses appropriate methods of `Parcel` class to construct and populate the `writeToParcel`, `readFromParcel` methods. It checks if the field is a file descriptor and populate the `describeContents` method. It finally creates a private constructor. We made this process recursive. Whenever it encounters a new type when parsing its field, it change it first, meaning any class within the same file that needs to be transformed is transformed in a single run. Finally, the tool dumps the transformed class to a new class file.

This tool only aims for assistance purpose and the tool is quite limited due to the heterogeneity of the states. We try to but do not guarantee to offload all responsibility of the system developers to this tool.
Chapter 5

Evaluation

In Drill, we select two services, ClipboardService (CS) and AlarmManagerService (AMS), from Android system services, and we deliberately crash them. We then recover them with their prior states as our case studies. We evaluate the behavioral correctness of the stateful recovery, the performance overhead and downtime. We also show the limitations of the approach and discuss why there does not exist fundamental limit to use Drill to other systems.

CS is a pure software system service managing the copied text data for each user. It has 7 states inside and only one of them is a data-state that we mark with \texttt{@States}. The rest 6 are left untouched since we rely on the service class constructor to recreate them. CS represents the simple system services in Android.

AMS is another system service holding alarms and timers for individual activities. This service is much more complex than CS. It has 78 states in total in the class definition. We mark 36 of them with the annotation and leave the rest of them to be recreated either in the service constructor or in the service’s \texttt{onStart} function. Another reason AS is more complex is that it relies on the underlying alarm driver in the Linux kernel and the physical timer hardware to set up timer and be woken up.

For the actual experiment, we extract the CS and AMS one at a time from the \texttt{system_server} and put them into the \texttt{drill_server} process. From a user’s perspective, we use the two services and store user’s states in the services (i.e. texts copied in the CS and timers in the AMS). The book-keeping service is in a different process.
Table 5.1: Behavior of CS and AMS before and after restoring the states when restarted from a crash.

We then send out the command to checkpoint the states from the service and store it in the book-keeping service. Finally, we kill the drill_server process and restore the states.

### 5.1 Behavioral correctness

Similar to works in the application migration [6] and fault tolerance [20], we also use the behavioral correctness as the metric from the user perspective. Without restoring the states, the user can no longer have the option to paste the texts from the clipboard. After the restoring, we can paste as normal, and the texts are the same as we have copied in the first place.

The behavior described in the Table 5.1 are as expected. It shows our approach and implementation can successfully checkpoint and restore the states in the two services. In the with Drill case, the cell phone user can still use the clipboard and the clock app to set alarms and timers as if no crash happens, which means the crash is transparent to the user and the fate of the application is to a great extent unrelated from the service. The behaviors we described in the table can be seen from our demo.
5.2 Performance overhead

In order to quantify the performance overhead brought by the C/R module and the states transfer, we measure the latency of checkpointing and restoring the states for the two services respectively.

From the Figure 5.1, in both checkpoint and restore cases, the AMS’s time is higher than that of the CS. This is because there is a huge difference between the amount of videos [21, 22].
states (36 vs. 7) to be checkpointed, transferred and restored. This is also confirmed by comparing the size of the serialized data (~4600 bytes vs. 40 bytes) of the two services. Also, restoring in general takes more time than checkpointing. We think it is because reconstructing the states requires calling the constructors of the classes with the arguments in the serialized data, which takes more time than simply storing information to the parcel in the checkpoint case. Another point worth noticing is that the standard deviation of latency is larger in the AMS case compared to the CS. This is due to the fact that the AMS is always actively used by other system components periodically in Android while the CS is not. Internally, the AMS will batch alarms whenever a new alarm is added, which results in quite frequent change of the states in that service.

We think the latency is still affordable because system services are long-term services designed to start when the device boots and die once the phone is turned off. Also, we roughly measure the frequency of the requests issued to the services. The system usually invoke AMS every minute for periodical tasks. For CS, it relies on how fast the user uses copy-paste functions in the phone, which is much longer than tens of milliseconds.

### 5.3 Down time

We measure the service down time which consists of the time for Zygote to fork a new drill_server process, start the service within it and restore the states. The absolute down time with and without restoring the states are shown in Figure 5.2. From it we can see that the major down time is spent in recreating a process and starting the service, which is inherently determined by Android system and hardware and we have no control over it.
Figure 5.2: Down time – time to restart the service w/o restoring the most recent states.

5.4 Limitations

States in the kernel. In a service crash, the states related to the process in the Linux kernel will highly likely be lost and therefore need to be taken care of. In our implementation and experiments, we find that the AlarmManagerService at first will not ring the phone when the timer set by user expires. The reason is that the AlarmManagerService opens the alarm driver file in the Linux kernel and creates alarm states even in the kernel. When the old AlarmManagerService process is killed (we emulate a service crash), such states in the kernel will be lost due to the fact that
the alarm driver is implemented as a miscellaneous character driver whose release function will be called where residual states are cleaned. In general, services which rely on their corresponding drivers in the kernel will highly likely leave states in it. There is need to preserve the states as well in the kernel when the process dies. However, we can still benefit from the fact that there are only certain types of drivers in the Linux kernel. That will likely help us handle such states in the kernel in a general way.

(De)serialization difficult for some systems. We choose serialization and (de)serialization to serve the purpose of fine-grained C/R and uses Parcel and Parcelable interface classes in the implementation on Android. However, (de)serialization methods are not available in every such system, especially when it involves special objects for IPC. Systems implemented in C and C++ which by design lack the reflection property of the programming language, thus it will be hard to checkpoint individual object even with the help of third party (de)serialization libraries. However, there does exist third party libraries that help those systems for (de)serialization. With careful engineering implementation, it is not impossible to apply Drill to other systems.

Parcelable not easily applied for complex states. Any state object that does not inherit from the Parcelable interface class will have to be transformed before it is ready for (de)serialization. Using Parcelable interface class and manually transforming the class ask for prohibitively large amount of efforts. It is also error prone and hard to maintain the meaningfulness of the states.

One example is a state in Context class. This state is quite common in the system service framework. In some cases where we have talked about, this context can be
treated as a function-state and be created in the system service constructor. In many other cases, we are not able to use such trick. However, to transform this class using Parcelable interface is nearly impossible. The Context (actual implementation class is ContextImpl) class contains 33 fields implemented in 20 different classes. Among all these classes, 13 of them need also to be transformed to become Parcelable classes. Some of them are even more complex.

Another concern is that it is hard to maintain the meaningfulness of the states, no matter whether we use the special Parcel and Parcelable interface or any generic serialization method. Take the Context as the example again. Of all 33 fields, there is a field called mMainThread whose class is ActivityThread. This field is passed in and assigned in the constructor of the Context. The (de)serialization method should reassign this field instead of creating a new ActivityThread object. Obviously, it is incorrect to have two activity threads. Although these two concerns exist, conceptually it is still possible to transform the classes and maintain the state meaningfulness. Therefore, there is no essential limitations that prevent Drill from being applied to other systems.
Chapter 6

Related Work

As the major topic in this paper is how to preserve the states so that fate sharing in among software entities is reduced in the system services, we organize this session in a way to discuss different state preservation techniques.

Checkpoint and Restore. Most of the past works in checkpoint and restore have quite different purposes from ours. Our target is all needed states within Java layer of the system service and necessary ones in the kernel, which is a subset of the complete states of the service process. However, past works mainly focus on the entire process or a group of processes, which essentially checkpoints states in a coarse-granularity. Checkpoint/Restore in Userspace (CRIU) [3] checkpoints the states of a process with the help of user space CRIU libraries and kernel hooks in individual kernel modules. As it aims to checkpoint the entire process, it has restorer to recreate the process and threads and virtualize the identifiers of the process in the kernel. As an extension of CRIU, Checkpoint/Restore in Android (CRIA) in Flux [6] is more related in that the target use case is Android system and it deals Android specific states. CRIA categorizes states into app-related states and device-related ones due to various Android devices and different Android versions. CRIA only deals with app-related states and mainly resolves issues brought by Android specific kernel drivers (i.e. Binder, Logger, ashmem, pmem and wakelocks). However, It still checkpoints states in the process level. The major use case of these two projects is application
migration, whose assumption is that the migrate-to device does not contain any states of the application. This is also different from ours where we assume that other system entities still keep states related to the service of our interest after it crashes. Another checkpoint restore work [23] proposes a checkpoint method via copying the entire memory address of the process together with the states in the kernel. Doing so effectively virtualizes the Android service entity to each individual applications. We share the same goal to reduce the state spill problem in Android system services but our approach can checkpoints much less states compared to theirs.

**Record and Replay.** From the implementation point of view, record and replay is a special version of checkpoint and restore. Not all the states of a software entity are saved for restoring purpose but only the arguments used for communication (e.g. method call and remote procedure call) among entities are saved. Record and replay hides the complexity to recreate the internal states of a software entity and those beneath by simply feeding the method call arguments, which gives itself an advantage over the checkpoint and restore method. Flux [6] shows that it can record arguments efficiently but not for behavioral correctness. It records all the RPCs of the application to the system service and heavily rely on Java annotation ability to ”adaptively” reply them. They manually analyze all the RPC interfaces of system services and spend quite amount of effort in annotating them so as to achieve behavior correctness from the application point of view. Compared to our work, on one hand we also do not checkpoint all the states to reach efficiency. On the other hand, as we checkpoint the snapshot of the states in the system, we have less need to worry about if redundant RPCs cause duplicated states, which results in behavior incorrectness.

**System-aided State Preservation.** Some research projects design their own
system in a way that the states or the information used to reproduce the states could be preserved by a special mechanism. Minix 3 [2] has a centralized data store which mainly saves primitive date types like string for future retrieval in case of a entity failure. Barrelfish/DC [24] partitions states based on each physical core so that it is easy to move the tasks in one core to another. CuriOS [25] preserves system services states in the separate persistent memory region. In case of a service failure, the service states related to the clients or the information to retrieve the client states will not be gone. They retrofit the existing system services in Choice operating system [26] individually to determine all relevant states needed to be saved. It also benefits a lot from the single space address space since the states could be reused with simple remapping. FGFT [20] leverages the existing Linux device driver’s suspend and resume routines and checkpoint method to save the device states. As for the device driver states, FGFT uses the software fault tolerance techniques aided by the compiler.
Chapter 7

Conclusion

We have shown in this work that existing systems which model the client-server architecture suffer from the fate-sharing problem. We have also described in detail how Android system services become the victims of such a problem. Applying Drill’s fine-grained C/R technique with the demonstration of our Android implementation shows that adding a C/R module to the server entity can preserve the states and and reduce the fate sharing problem because of state loss to some extent.

The limitations we have identified along with our Android development are valuable. Preserving states in a per-object fashion requires a strong se/de-serialization technique. For a complex system, the meaningfulness of the state really matters. A software state can not be simply recreated without taking the role it plays in the entire system. Blindly recreating a state using any se/de-serialization method will cause fatal problems. Therefore, we conclude that being generic and special at the same time to handle various states while maintaining their meaningfulness is very challenging.

Therefore, in order to apply Drill to a real system, it is necessary to tailor a generic se/de-serialization method for various types of states. This process may be much easier if the developers of such system get involved. Therefore, we think that the limitations we have identified do no restrict us from applying Drill to other systems.
Bibliography


[18] “The chromium projects: Inter-process communication (ipc).”
https://www.chromium.org/developers/design-documents/
inter-process-communication, 09 2016.


using device checkpoints,” in ACM SIGARCH Computer Architecture News,

https://www.youtube.com/watch?v=GCUPtHb0XM8, 12 2016.


checkpoint-based virtualization of mobile os services,” in Proceedings of the 6th

and operating systems,” in 11th USENIX Symposium on Operating Systems De-

reliability through operating system structure.,” in OSDI, vol. 8, pp. 59–72, 2008.

[26] R. Campbell, G. Johnston, and V. Russo, “Choices (class hierarchical open inter-
face for custom embedded systems),” ACM SIGOPS Operating Systems Review,