Community based testing

SAULIUS ALISAUSKAS

KTH Information and Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

Community based testing

Saulius Alisauskas

saulius@kth.se

Master thesis draft

Examiner:
Professor Gerald Q. Maguire Jr.

School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

30 of June 2013

Abstract

Currently, Android is the most popular operating system for mobile devices, but at the same time, the market for Android devices is heavily fragmented in terms of available versions, types of devices, models, form-factors and manufactures. As a result, it is virtually impossible to test applications on all existing devices. Testing on a set of the most popular devices is more realistic but can be expensive, which makes it much more difficult for individual developers to create high quality applications.

Fortunately, each Android application developer around the world typically owns at least one device which is not used all the time and could be shared with other developers. This way, a community shared pool of Android devices can be created for automated test execution.

This master thesis reviews existing testing frameworks that are used for testing Android applications and analyzes existing services that in one way or another try to solve the problem of providing affordable ways of performing testing on real devices. Main result of this thesis project is a working distributed community based testing service that enables developers to easily connect, share, and execute automated test cases on devices that use Android operating system. Moreover, it provides ways of decreasing overall test execution time by executing parts of tests in parallel on multiple devices and aggregating received results.

Sammanfattning

För närvarande är Android det mest populära operativ system för mobila enheter, men samtidigt marknaden för Android-enheter är starkt splittrat i fråga om tillgängliga versioner, typ av enhet, modeller, form-faktorer, och tillverkar. Som ett resultat, är det praktiskt taget omöjligt att testa applikationer på alla befintliga enheter. Testa på en uppsättning av de mest populära produkter är realistiskt men kan vara dyrt, vilket gör det mycket svårare för enskilda utvecklare att skapa högkvalitativa ansökningar.

Lyckligtvis äger varje Android ansökan utvecklare i världen typiskt åtminstone en enhet som inte används hela tiden och kan delas med andra utvecklare. På så sätt kan en gemenskap delad pool av Android-enheter skapas för minst automatiserade test exekvering.

Detta examensarbete går igenom gällande testning ramverk som används för att testa Android applikationer och analyser liknande tjänster som på ett eller annat sätt försöka lösa problemet med att tillhandahålla prisvärda sätt att utföra tester på riktiga enheter. Huvudsakliga resultat av detta examensarbete är en fungerande distribuerad gemenskap baserad testning tjänst som gör det möjligt för utvecklare att enkelt ansluta, dela och exekvera automatiserade testfall på enheter som använder operativsystemet Android. Dessutom ger det möjligheter att accelerera övergripande testexekvering tid genom att utfö!ra delar om tester parallellt på flera enheter och sammanställa erhållna resultat.

Acknowledgments

First of all I want to thank the love of my life Vaida for support, encouragements and for those numerous sacrificed evenings that were meant to be spent together. I wouldn't have finished this without you!

Also, this work wouldn't have been that much fun without great ideas and suggestions from Professor Gerald Q. Maguire Jr.

Table of contents

List of figures

List of tables

List of Acronyms and Abbreviations

ADB	Android debug bridge
ADT	Android development kit/plug-in for Eclipse
API	Application programming interface
APK	Android package[1]
CBT	Community based testing
CBT agent	Community based testing agent
CBT RIP	Community based testing restful interface protocol
CBT web service	Community based testing web service
CI	Continuous integration
CPU	Central processing unit
DEX	Dalvik executable
GPU	Graphics processing unit
HTTP	Hypertext transfer protocol
HTTPS	Hypertext transfer protocol secure
ID	Identifier
IDE	Integrated development environment
JAR	Java archive
MVC	Model-view-controller software architecture pattern
MWC	Model – view – controller
OS	Operating system
SD	Secure digital
Test-script	Compiled test code packaged into a known format (JAR)
Test-profile	Information about types of devices to executed tests on and execution mode selection (*normal*, *fast*)
Test-configuration	Information aggregating test-script, test-profile and test-target
Test-target	Information about Android application to be tested, may refer to actual application.
UI	User interface

1 Introduction

Android[2] is the most popular[3] operating system for mobile devices. It is based on a modified Linux[4] kernel. Applications run on a Java[5] compatible framework. Performance greedy applications, such as games, can be written in C/C++[6, 7] using NDK[8]. Android was built to support various types of devices differing in technical characteristics, screen sizes, pixel densities and screen resolutions, sensors, input devices and etc. Since possible differences in device properties have been thought through from beginning, it is rather easy[9] to write applications that can adapt to all possible variations, although that definitely makes development much more complicated compared to developing for only one device. Moreover, varying device properties complicate application testing process as well. Android OS is constantly improving, bringing a handful of features not only with every major release (such as 2.0, 3.0, 4.0), but also with minor version updates. This is great for users since they get improved software more frequently, but it makes developers life harder due to the addition development time needed to support new system features, while at the same time, keeping compatibility with older versions of the OS.

In order to develop high quality applications, proper testing of application features on various types of hardware configurations must be performed before releasing an application to the public Even after the application is released, changes and improvements to the code will be necessary and testing will have to be repeated. Depending on the application, manual testing might be sufficient. However, most of applications today are quite complex and manual testing is not suitable simply because it will take too much time to re-test the whole application after every change. Therefore, many developers use testing automation tools and continuous integration tools. Automated test scripts can be written to test applications at different levels: unit tests perform testing of single components such as classes, while functional tests make sure that certain functionality is provided by higher level components. The system level testing (in case of Android applications) encompasses testing the functionality from a user interface perspective. Considering Android's fragmentation in terms of the number of different device types and numbers of different versions of the operating system (OS), it becomes clear that testing of Android applications and making sure they function properly and that the user interface is presented as expected is a major challenge[11].

Testing needs to be performed not only before each release, but also with each code change in order to detect and fix problems as soon as possible. This means, that testing routines must be able to be performed as fast as possible and that puts heavy requirements on automation tools and testing environments. Continuous integration tools like Jenkins[12] can be used in conjunction with test scripts in order to automate the process of test execution. Tests execution can be triggered by certain events such as code from one developer being committed into the code repository. Tool will then interpret test results and inform interested parties about success or failures and will provide detailed reports which will help to identify the source of the problem. Tests may take different amounts of time, depending on many factors such as application complexity, dependencies, testing environment availability and testing scope. In some situation it might be enough to run tests on Android emulators, but tests should be performed on real physical devices and on as many as possible, in order to have high enough level of confidence in applications quality.

Developers can usually choose between few options when deciding how to perform testing on real device: first is to get a number of most popular devices and perform testing on them[13], second approach is to lease devices for the time that is needed to run the tests[14] and third is to hope that application will work the same on other devices same as it work on developer device. First option greatly increases the total cost of application development. Moreover, special software and infrastructure is required in order to manage devices and distribute testing routines. Second option decreases development and investment costs sue to the fact that costs of these services is determined by the usage. However, second option might still be too costly for individual developers and they are left with the third option.

This master thesis analyses the problem of testing Android applications of real physical devices, designs and implements a system that enables device sharing between developers for automated test execution. It utilizes existing devices owned by developers eliminating the need to purchase and maintain expensive smart devices this way dramatically decreasing cost of system deployment and maintenance. Additional possibilities open up when many devices are connected into a controlled network such as decreased test execution times using parallel execution and availability of less popular devices.

The paper starts by giving background information on Android operating system, testing frameworks available for testing Android apps, overview of what continuous integration and testing automation is, and analysis of existing services, which try to solve problems of testing Android apps on real devices. It then analyses possible ways of solving the problem of the scarcity of Android devices available for automated testing, due to the high costs of hardware and what exact approach was taken for solving this problem as well as defines the goals of this work in chapter 3. Chapter 4 gives a detailed description of the implemented system prototype that solves mentioned problem by leveraging Android devices available within developer community. Description starts from higher level components and dives deeper into specifics. Paper ends by discussing possible future work and giving conclusions in Chapter 1.

2 Background

This chapter provides general information about Android OS, design and lifecycle of Android applications, how Android applications can be tested, and what testing frameworks and tools are available to support testing process. Next, subchapter 2.3 briefly presents continuous integration [15] and what tools are available for supporting continuous integration on the Android landscape. Finally, subchapter 2.4 presents and compares existing third party services which are similar to the prototype service that was designed and developed during this master's thesis project.

2.1 Android OS

Android[2] is an open source operating system (OS) owned by Google[16] and the Open Handset Alliance[17]. At the time of this writing, the Android OS is the most popular OS[3] as it is implemented in far more handsets that other OS, such as Apple's iOS, Symbian, RIM, and Microsoft's Windows Phone. It uses a modified Linux kernel[4] and a Java based framework on top of which applications reside. The Android software development kit[18] (SDK) is freely available for application developers and provides a number of tools to facilitate development and testing. Although applications are mainly written in Java, these applications are quite different from ordinary Java applications due to concepts such as *Activity*[19] and *Intent*[20] and quite complex application lifecycle shown in Figure 2-1 Activity lifecycle in Android OS. Similar to other OSs, Android is composed of a large number of components, but, describing all of these components is outside the scope of this thesis. Therefore, only those parts of the Android OS that are more related to testing will be discussed in this section.

Activity[19] is part of application which acts as a controller and model when referencing MVC[21] (model-view-controller). All *activities* stored in a stack are managed by OS and only one can be displayed at a time. Moreover, even though applications are usually composed of a number of *activities* (in fact, there might be only one *activity*), each activity runs in a separate process making it much more difficult to pass information between them. Inter-process messaging is used for this and container objects called *bundles*[22] can be exchanged. Therefore, from a system perspective, application is a collection of *activities* stacked on top of each other and user is in control of going up and down through the stack. In some cases, user might not even be aware that he is switching to an *activity* that belongs to another application.

Intents are used to launch new *activities* and contain information such as which *activity,* or to be more precise, which action an application intends to launch and might contain a *bundle* with primitives and objects to be passed on to an *activity*. Depending on *intent,* the system might present a number of *activities*, for example, if the *intent* indicates that it would like to share a message, then a list of *activities*/applications such as email clients or social network clients, will pop up for a user to choose from. The list is constructed based on which *activities* have been registered to handle specific media types. This is a powerful feature of Android and it enables re-use and sharing of functionality between applications. Additionally, this design methodology de-couples and completely separates parts of the same application, therefore, most sophisticated code analysis tools[23] for Java cannot be directly used. Moreover, Android application framework uses system services and libraries that are available *only* in Android environment, therefore, applications *cannot* be run natively on development machines, hence they can only be run on simulators or real devices. Due to this fact, testing Android applications is quite complex and requires sophisticated supplementary tools. It is no surprise that a lot of these tools are provided within the Android SDK.

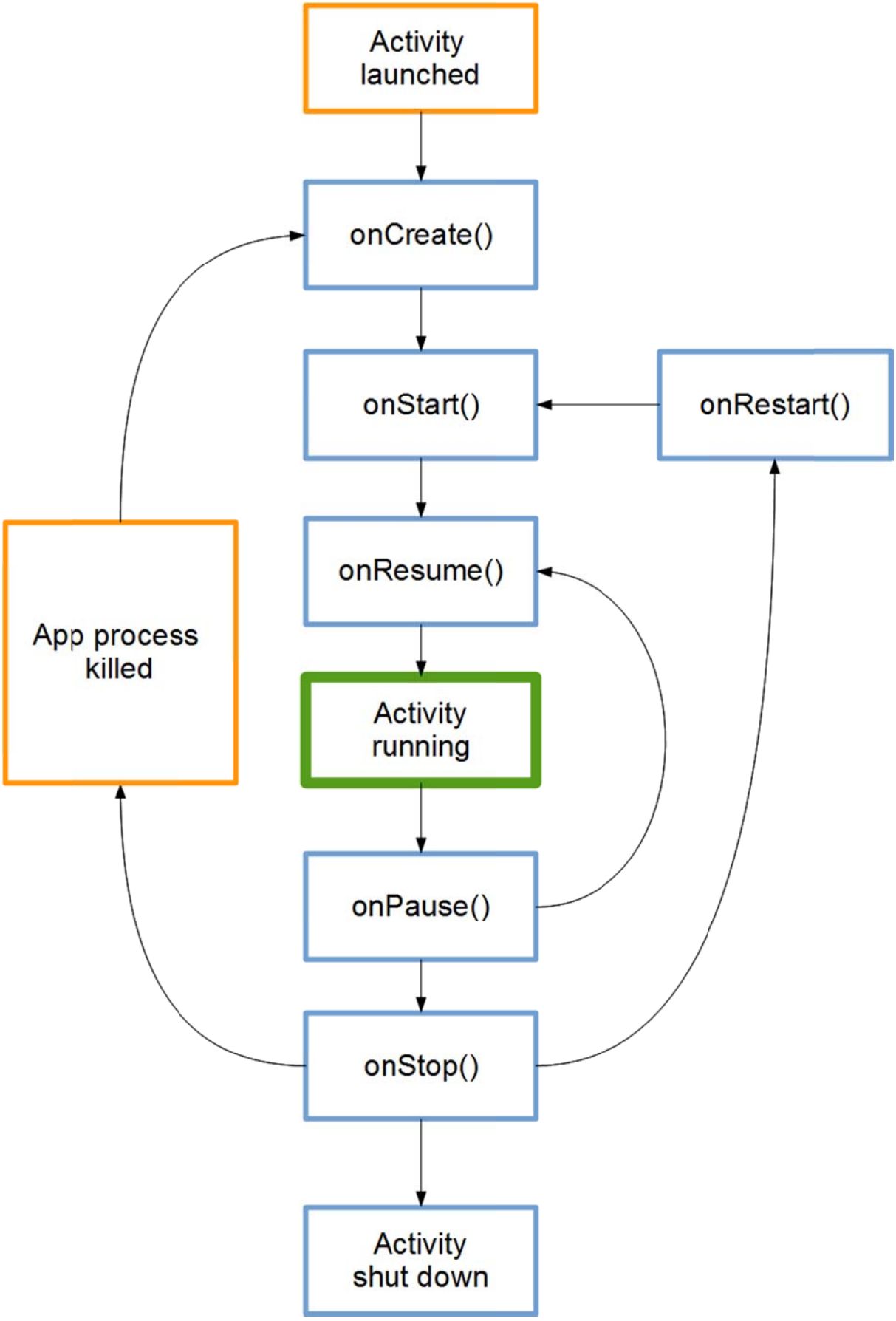

Figure 2-1 Activity lifecycle in Android OS

2.2 Testing Android applications

There are a number of testing frameworks and tools provided by the Android SDK, along with other community managed open source projects. The following section discusses some of best known frameworks and tools and explains their usage and specialization. The process of packaging and executing tests based on different frameworks is discussed in order to present a complete picture of what actions are involved in testing lifecycle and to show how complicated it is to perform these actions.

2.2.1 Unit and function testing

The Android SDK provides powerful testing tools and test design frameworks for testing various aspects of an Android application. One of possible options is to use *Instrumentation*[24] which provides special hooks into the system and application process which are used to give control over application lifecycle to the test script. This framework is based on the well-known jUnit[25], therefore, a test follows the same principles as ordinary jUnit tests. Similarly to developing an application, there is a special template for a test project when using Eclipse with Android development kit/plug-in for Eclipse called ADT[26]. Test classes must extend one of the provided super classes depending on the scope of testing.

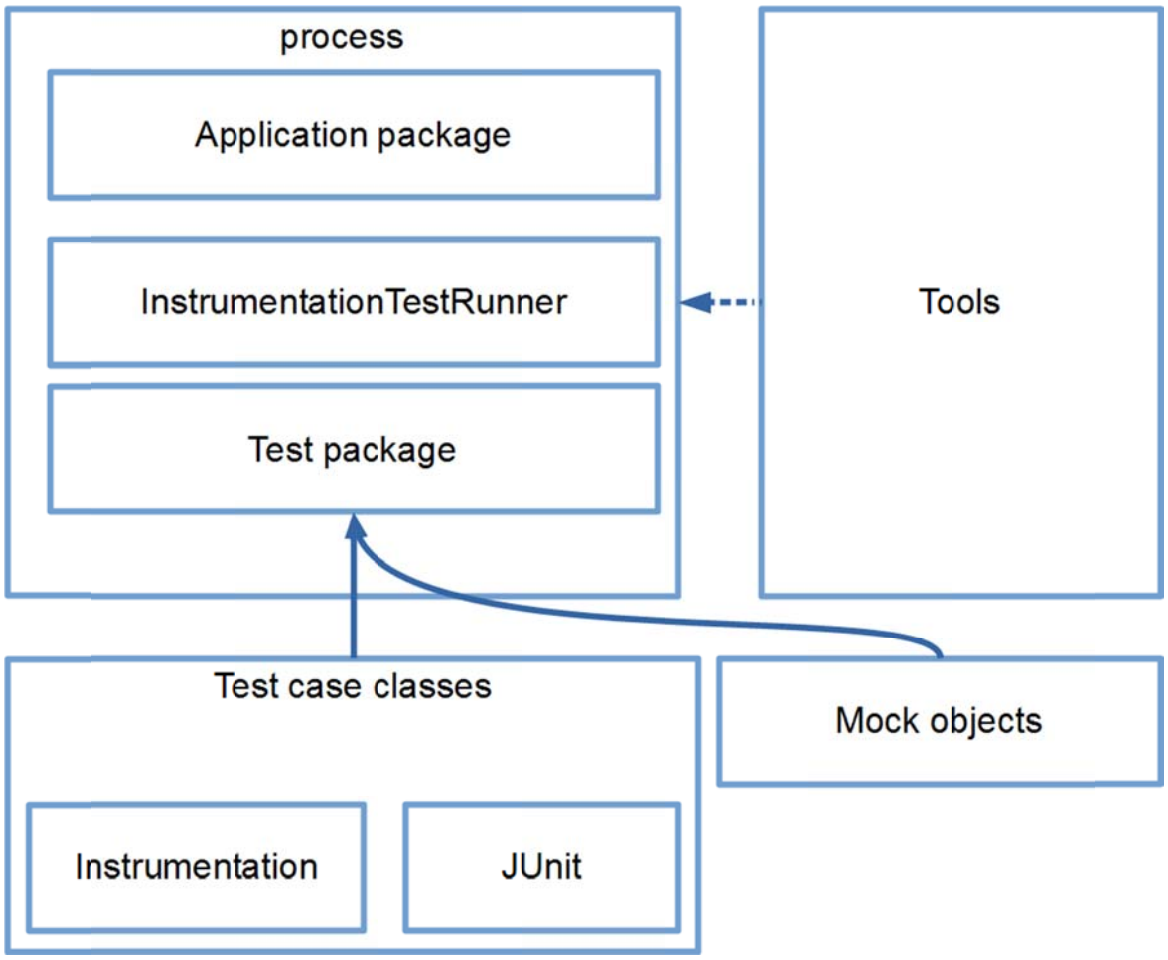

Figure 2-2 Android testing framework hooks diagram

In order to perform *activity* unit testing, a test-script must extend the *ActivityUnitTestCase*[27] super class. In this case, a*ctivity* is isolated and thus a simple test can be performed much faster. However, there are many limitations and this approach is useful only for testing methods that do not interact with a real Android system. *Activity* function testing can be performed by extending the *ActivityInstrumentationTestCase2*[28]. *Activity* will be run in a real Android environment and much

more sophisticated testing can be performed, while still being able to inject mocked objects. For example, mocked Intent can be injected to cause an Activity start. Subclasses of *ActivityInstrumentationTestCase2* class provide helper methods and mock objects to deal with different component's lifecycles and calls to system services. However, in order to write a test script, the inner structure of the application must be known and even specific IDs of certain objects must be referenced from within the test case, hence this kind of testing represents *white-box*[10] testing.

The test package must provide name of instrumented test runner in the application's manifest file. An instrumented test runner is responsible for setting up and tearing down test cases. The *InstrumentationTestRunner[36]* is a primary Android test runner. Tests are started using a command line Android debug bridge (ADB)[37] tool or Eclipse with ADT. In either case the system loads and starts a test package, kills any running instance of the application under test, and starts a new instance of this application passing all control to the test runner.

2.2.2 System and user interface (UI) testing (uiautomator)

Luckily, there are additional tools that help to perform *black-box*[29] testing – a type of testing when the tester does *not* know about the application's inner structure and hence must perform testing based on functionality expose by application such as buttons and text entry fields. There are a number of problems with UI (user interface) testing when the inner structure of an application is unknown. In Android, each view object is usually described in an XML file and each object has a label defined by the developer. When building a project, these labels are converted into numbers which can be used to referrer to specific UI element from the source code. When performing *black-box* testing, these identification numbers are unknown to testers, then a question arises, how does a test-script specify which view object to perform an action on? An example could be when a script needs to click on a specific button testing framework would have to simulate a touch action on a specific part of the screen. However, the same button can be located in different places on the screen depending on screen-size, resolution, orientation, or even device type. One way could be to analyze image of the rendered screen and try to find a view object based on an image prepared before, but that would be extreme resource intensive operation and would slow down automated test radically.

The *uiautomator* tools suite which is supported on devices running Android 4.1 (API level 16) or higher, solves the problems mentioned above. It provides access to a device similar to Monkey runner (described in section 2.2.4), but at the same time, sophisticated jUnit based test-scripts can be created. A tool called *uiautomatorviewer*[30] quickly builds a hierarchy of view objects and their properties. Then, values of the view object property named *contentDescription* can be used to refer to particular views when writing a test-script. It is important to note that tester does not have to examine the source code of the application nor does he actually need to have the source code, therefore, a test-script has no dependency on the source code, but at the same time the test script can avoid central processing unit (CPU) intensive operation such as analyzing images to find a specific spot on the screen. The testing framework can quickly find and identify a referred view and allow the script to perform specific actions on it.

When a view hierarchy is known, a test-script can be written extending the *UiAutomatorTestCase*[31] class. This class provides access to device functionality such as getting device state properties, pressing buttons and taking screenshots. Also, a number of helper classes exists, one of which is *UiSelector* – which provides a simple way to handle a specific view object by providing the object's class name or description – a value of the property *contentDescription* which can be retrieved using the *uiautomatorviewer* tool as mentioned earlier.

This type of tests has a different packaging and deploying procedure than *Instrumentation* based tests. This procedure is presented in chapter 2.2.5.

2.2.3 Robotium

Robotium[32] is an open source testing framework for Android applications. Currently official projects web site states that it is the "world's leading Android testing automation framework". It is a collection of useful methods and classes that improve the readability of test code and provide a lot of commonly used functions, such as waiting for certain *Activity* to occur or entering text into the first "*EditText*" field. It allows loose coupling between UI components and actions performed on them by referring to buttons and other UI field by their displayed name. UI elements can also be referred to by their auto-generated identifier. However, latter approach couples the test code with the actual application. Tests written with Robotium still have to extend the *ActivityInstrumentationTestCase2* since Robotium leverages Android *Instrumentation*[24], but, the features it provides greatly improve productivity when writing this kind of tests.

Robotium can be integrated with Maven[33] or Ant[34], which are both build management tools. There are Maven plugins for Android, so Android applications can be managed by Maven and Robotium integration with Maven allows the build manager to manage the testing through Robotium. However, all Maven related tools are provided by the community and are *not* part of standard Android tools. It's worth mentioning, that Ant was used as the default Android application build tool, but with the introduction of Android build tools version 17 Ant has been replaced by Gradle[38].

Since Robotium depends on Android *Instrumentation*, it is coupled together with the application under test and cannot be completely de-coupled and developing a test case requires having source code of target application. Therefore Robotium is most useful for unit testing of the inner components of an Android application.

2.2.4 Monkeyrunner

Another tool provided by the Android SDK is "monkeyrunner" which facilitates automated testing. Monkeyrunner enables automation of functions such as launching a new emulator instance, connecting to the emulator, installing an application package, running an application package, sending keystrokes, taking screen shots and more. All of these features are provided via an application programming interface (API), therefore, monkeyrunner can be used to automate device management and package installation. Scripts are written in Python and the tool itself uses Jython[36] to interact with the Android framework and to provide access to constants, classes, and methods. However, monkeyrunner does not allow creation of detailed test-scripts where specific user interface components are referenced from the source code since monkeyrunner does not provide a way to reference specific UI elements from within the script.

2.2.5 Packaging and deploying a test project

Currently, a test-script which extends the *UiAutomatorTestCase* can be packaged and run only via the command line interface. Since Android earlier build tools used Apache's Ant[41] as a build manager, a test project can also use Ant. Packaging and deploying a test application involves:

1. Generating a test project by running the command:
   `android create uitest-project -n <name> -t <target-id> -p <path>`
   where `<name>` is a test project name, `<target-id>` - id of existing target (targets can be listed with command: android list targets),<path> - path to the test projecy directory
2. Building the project by executing: **`ant build`**
3. Sending the generated JAR package to a target device:
   `adb push <name>.jar /data/local/tmp`

The `/data/local/tmp` is be the default location and can be changed to any other location to which user has permission to write and read from. This path will later be used to launch tests from.

After performing the above steps, a JAR package containing tests is converted into a Dalvik Executable (DEX) byte code file and deployed onto devices. The target application which will be

tested must be separately uploaded and installed following any of the available ways (e.g., using **`adb install`**).

The *ActivityUnitTestCase* and *ActivityInstrumentationTestCase2* subclass based test-script are packaged into an Android package [1]. APK is used for all Android applications, therefore, installation to a target device can be done in various ways, one of which is using the ADB tool:

```
adb install -r <target-app-name>.apk
```

This command can be used to install both test applications (but only *Instrumentation* based) and target applications.

2.2.6 Executing test

Execution differs depending on the type of test or it's packaging. Test-scripts that extend *UiAutomatorTestCase* can only be executed using console commands, whereas *Activity* unit and function tests (extending *ActivityUnitTestCase and ActivityInstrumentationTestCase2*) can be executed directly from the Eclipse IDE with ADT. However, command line execution can be easier to control with a dedicated application or automated with scripts, therefore, only the process of executing test-script using the console will be described below.

Test cases extending *UiAutomatorTestCase* can be executed with the following command:

```
adb shell uiautomator runtest <name>.jar -c <package.class>
```

Where `<name>` is the name of a package that contains the test-scripts which should be present in `/data/local/tmp` path.

Manipulating command options, user can run all test cases in a package tree or specific class or even methods providing flexibility to be able to choose exactly what is to be executed.

The following are the steps need to be done in order to execute *ActivityUnitTestCase* and *ActivityInstrumentationTestCase2* subclassed tests:

```
adb shell am instrument -w <package>/<InstrumentationTestRunner>
```

Where `<package>` is the package name of the test project as specified in the AndroidManifest.xml file and `<InstrumentationTestRunner>` is the instrumentation test runner to be used.

2.2.7 Android emulator

Android SDK provides an emulator that can be extensively customized and used to mimic real Android devices. Customization includes modification of properties such as screen resolution, pixel density, OS version, presence of a hardware keyboard, GPU, accelerometer, SD card size, etc. The emulator can perform ARM[37] instruction emulation (since most of Android device up to the date of writing still use ARM CPU's) or utilize special images provided for performing emulation on an x86 architecture. The later, runs much faster on Intel's x86 CPUs (which are usually used by developers for developing applications and tests) since it can use Intel supported virtualization technologies. However, there is one major limitation – only one emulator can be running at a time using accelerated mode on Windows OS (there is no such limitation on Linux). So, if a developer, needs to run his or her application in emulators with different screen sizes, they either have to run them simultaneously on a slow ARM[37] image or x86 images have to be emulated sequentially one by one. In any case, running multiple Android emulators on development machine slows the system down marginally which hurts development process. Therefore, developers cannot have multiple emulators running on the development machine and would need to boot different configurations in sequence for compatibility testing. Moreover, it is worth noting that successfully running an application on the emulator does *not* necessarily mean that the application will run successfully on a real device. There also might be performance differences – which can lead to undetected timing errors, therefore, most developers strive to test their applications on as many different real devices as possible.

2.3 Continuous integration, testing automation

Continuous integration[15] plays a very important role in software design these days, especially in bigger projects built by multiple developers. It is so important since it is a prerequisite for Agile software development as well as continuous deliver which enables delivery of new versions of the product in a matter of hours or days. According to this practice once a small change is introduced in software, tests should be performed in order to verify that the change does not introduce any faults in depended software components. Only after a change is successfully verified, should the code be integrated into main development line (this could be a central branch in code revision control system, such as Git[12]). Verification is usually done by running a suite of tests of various levels. However, somebody has to write the tests and in the end, the quality of these tests determines the level of quality assurance provided by tests. Increasing code coverage[38] increases this assurance level, while at the same time, increasing the time required to execute all of the tests. Therefore, there is always a tradeoff between the level of testing and the time required to get feedback for a change. The time required for integration testing not only depends on the test code coverage, but it also depends on the application and/or application component dependencies on other components, applications, and services, time needed to setup the testing environment and possibly other factors. As a result the verification process can consume a lot of time. One of the requirements of CI is rapid feedback, since the faster and at earlier stage a problem is detected, the easier and therefore, less expensive it is to fix the problem. In order to speed up verification or integration, tests can be broken into groups and performed on a number of identical environments specifically prepared for testing that application. However, this approach depends on a number of environments being available and having a lot of integration testing environments is costly no matter what kind of application is being tested.

Jenkins is one of the most popular tools for CI. It is application type agnostic, meaning that almost any kind of application written in various programming languages can be used together with Jenkins. There are numerous plugins for build management tools (including Maven, Gradle, Ant etc.) that make CI configurations easier. Moreover, plugins for various testing frameworks can be added in order for a Jenkins server to be able to interpret the results of the test executions. Android is no exception and open source community has provided plugins for Maven. Android applications managed by Maven can easily be used in Jenkins and leverage all of Maven's functionality and benefits.

Nevertheless, testing Android applications requires having devices to test applications on and in the case of time consuming test suites, having access to a large number of devices could speed up the process. In order to be able to have a fully automated CI system, it is important that the tools used within the application's lifecycle support automation at every step. This same requirement is true for "device cloud" type services that provide access to a device hosted within a cloud of devices for automated test execution. These kinds of device cloud services are discussed and compared in the next section.

2.4 Existing similar services

It is important to briefly review existing services that help Android developer to perform testing on read devices in order to understand what options, problems and benefit they provide. Table 2-1 compares different services based on different criteria. In some cases, this comparison is subject to the author's opinion. For example, in this comparison it is considered positive to have support for CI build managers, such as Jenkins and to support the Android instrumentation test framework based tests, whereas, having a proprietary test framework is a negative property. Provide social features, such as: enabling other developers or simply interested people to provide feedback while doing manual testing is also considered positive property. Naturally, in this comparison providing a "free" community service is also considered a benefit.

This feature comparison table is bases on the information available at the time of writing this thesis. Considering the rapid development of the Android OS and surrounding tools, services will

most probably evolve at a similar pace in order to keep up with requirements put on them by application developers.

Table 2-1 Comparison of similar existing services

	Xamarin Test Cloud[39]	**TestDroid [14]**	**PerfectoMobile [40]**	**Keynote device anywhere [41]**	**AppLover [42]**	**Utest [43]**	**CloudMonkey** [44]
Support for Instrumentation *	yes	yes	no	no	no	no	no
Supports *uiautomator* framework	no	yes	no	no	no	no	no
Proprietary testing frameworks	no	no	yes	no	no	no	yes
Other testing frameworks, tools	Calabash (Cucumber)	Testdroid Recorder	no	no	no	no	MonkeyTalk IDE
Payed	yes	yes	yes	yes	free	yes	yes
Device cloud	yes	yes	yes	yes	no	no	yes
Emulators†	no	no	no	no	no	no	yes
Integrates with build managers	Command line tool	Cisimple[47], Jenkins REST API	no	no	no	no	Jenkins
Interactive manual testing, streaming	no	no	yes	yes	no	no	no
Manual test performed by others	no	no	no	no	yes	yes	no

Certain services (such as TestDroid and Xamarin Test Cloud) focus on automated test cases, however, a different approach is taken considering supported testing frameworks. TestDroid supports testing frameworks available out of the box in Android SDK (i.e., *uiautomator* and *instrumentation*) and their own tool which makes it easy to create a test script by recording actions issued on a device (clicking button, swiping,…). The Xamarin Test Cloud uses Calabash[46], which uses a different approach to testing. In this case the test scripts are not written in Java, which according to Calabash's creators, improves readability of large test suites. However, since there is no other way of executing Android test cases than using *Instrumentation* or *uiautomator,* Calabash generates *Instrumentation* type tests which theoretically could be executed on any platform that supports *Instrumentation* based tests. Both TestDroid and Xamarin Test Cloud services have two options: use their devices hosted in a public cloud or build your own private device cloud. A public cloud is a shared pool of devices hosted by service provider, whereas, a private cloud is a pool of devices managed by the service provider's software but hosted by a client company. In addition, both services provide tools for integrating with CI machinery. Neither of them supports sharing

* Supports tests based on Android instrumentation[34]
† Provides ability to run tests on emulators running in cloud.

devices between private clouds (at least no information about such a feature was available at the time this paragraph was written).

Private clouds solve one of the major issues with public clouds – testing queues. In order to be profitable and stay within reasonable price ranges, service providers are forced to strive to utilize their devices for as much time as possible, i.e., avoiding having idle devices. This leads to a tradeoff between device utilization and queue length (i.e., delay for test execution). Another limitation is that new devices have to be bought over and over again– to be used for testing which increases the costs. Private clouds are also useful for companies that do not want to risk exposing to the public their application at an early phase of development.

Other services focus on *manual* testing, but in different ways. For example, PerfectoMobile and "Keynote device anywhere" provide ways of remotely connecting to a device that is hosted in their public cloud. Actions performed on the device (using mouse and keyboard) are sent and emulated on device and the screen view is streamed back to the human tester. This way manual testing can be performed on remote device interactively.

Another type of manual testing is represented by services such as AppLover and Utest. These services act as mediators between professional testers or enthusiasts that want to try out new apps and developers. These services do not provide devices to run tests on, but rather, use devices provided by testers. What these services actually provide is a platform to distribute applications to multiple people and to collect their constructive feedback regarding the application's usability, bugs and user experience. Utest is more sophisticated, partly due to a fact that it is not only mobile applications oriented but it provides testing service for almost any kind of applications. However, this service provides only manual testing.

3 Method

A solution that would solve the problem of scarcity of Android devices for testing as well as avoiding high costs of hosting public device clouds or building private could be a platform that makes use of all of the devices that each Android developer and possibly even random people have. An assumption could be made, that each developer owns at least one Android based device, therefore, each new user contributes at least one new device for testing. Also, this device is typically used less than half of the day, for example during the night while the user is sleeping. While the device is not being used, another developer on another side of the globe could use the device, hence taking advantage of the difference in time zones. The platform could provide an interface similar to current social networks, where instead of sharing documents and pictures, one could share access to your device. The access to these shared devices could be provided in the form of an ability to upload test-scripts & applications, execute test-scripts, and fetch results together with screenshots from the device. Such a service would enable many developers to avoid buying expensive handsets, thus decreasing cost of testing compared to existing commercial services offering device cloud solutions (chapter 2.4). A "tit-for-tat"[50] strategy could be used to control the load on the system and could provide large number of different kinds of devices and lots of testing time, thus reducing the delay for test execution since queues would be shorter. Moreover, given a large pool of available devices tests could be run in parallel on a large set of devices hence providing wide device coverage in a short time. However, an even more interesting functionality could be provided by applying another assumption: since there are certain "popular" devices which tend to be sold more often than others, it seems right to assume, that the spread of device models by popularity in the developer community would be similar to their spread among ordinary consumers, thus such network would be mostly filled with "popular" devices and there will be large numbers of same model of these devices. As a result, tests could be split among a number of devices, hence accelerating the execution of test suites. No such service or platform exists to the author's knowledge at this time.

The main functionality of this platform would be to *manage* connected Android devices, *distribute* tests among available devices, and provide *easy to use interface* for uploading tests and applications. In order to be competitive feature-wise, it would also need to support integration with CI tools, provide useful statistics about the devices (collected in the background while tests are running) and collect screenshots, log files and possibly other files generated during test run. This platform would need to support at least Android *Instrumentation* tests and/or *uiautomator* tests.

Due to the very large set of features mentioned above and limited time & resources, the primary goals of this master thesis were narrowed down to the following core features:

- User management
- Device management
- Device and user authorization
- Manual test file uploads
- Test distribution, execution, result collection and presentation
- Support for Android's *uiautomator* based tests

As a result, the user should be able to connect their Android device to the system and share this device with other users, who can subsequently upload test files and execute them on all the devices that have been shared with them.

The following items are explicitly *out of the scope* of this master thesis:

- Security of communication between users, devices, and system components
- Detection of malicious applications or tests being uploaded to a device
- Business model(s) for how this kind of service could make a profit
- Reward system for rewarding device donors and charging device users

4 Analysis

As a result of this master thesis project a web platform was designed and implemented that enables developers from around the world to share Android devices and to execute automated test-scripts based on the *uiautomator* test framework. It provides a way of significantly reducing the cost of testing applications for compatibility with different types of Android devices by making use of devices provided voluntarily by other people. Since potentially all of the devices running Android OS can be connected to the platform, this platform may provide a nearly unlimited size pool of resources for testing. For this reason, a special mode of test execution was implemented which splits a test package into test classes and distributes these test classes over devices of the same type, therefore, radically decreasing the overall time required to execute a test suite.

This chapter describes the functionality and architecture of the prototype system starting from higher level and then getting into details of each system component.

4.1 General CBT system overview

The system prototype consists of 2 main components:

- *Community Based Testing (CBT) web service* which connects Android devices into a pool of resource that can be used for executing automated tests. This component provides a web UI which is used for file uploading and test initiation. RIP – restful interface protocol is used by CBT *agent* to communicate with CBT service.
- *CBT agent* - a Java application for managing Android devices connected to the same PC and connecting these devices to a *CBT web service.*

The overall system architecture is shown in Figure 4-1. It was build using industry standard technologies, tools, and design patterns. The CBT *web service* is a Restful[48] web service[48] which uses the HTTP protocol and is of a request-response nature.

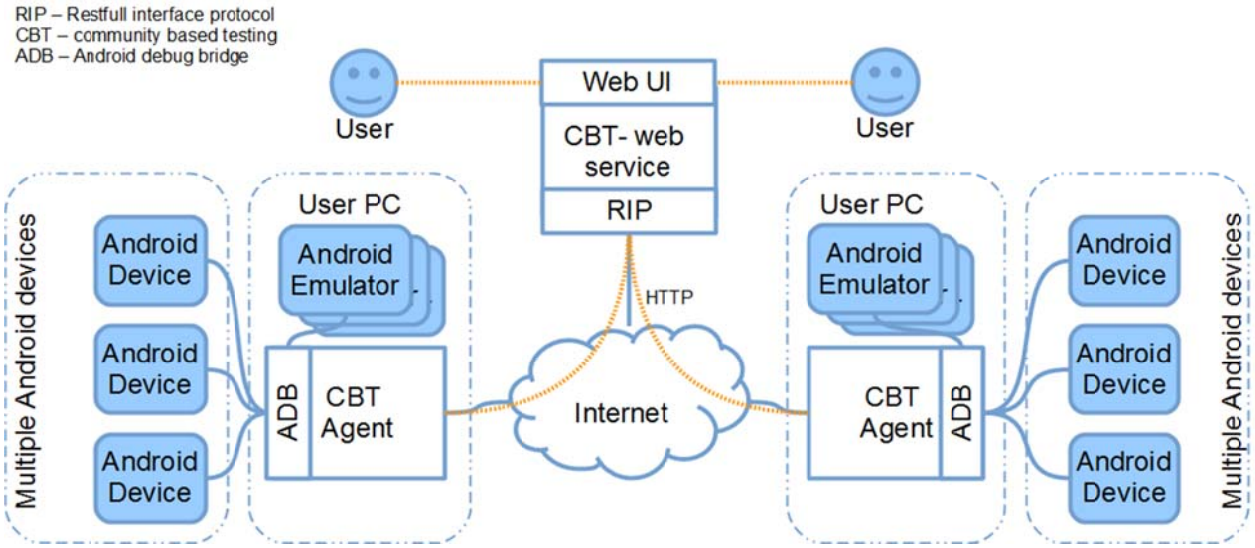

Figure 4-1 Community bases testing system components

The Java programming language was used for both, the *CBT web service* and *CBT agent* in order to re-use code, platform OS independency, and because Java is one of the most popular programming languages – hence there are numerous open-source libraries and tools. Another reason for choosing Java was that it is the main language for creating Android applications. Guice[49] was used to make the inner code structure more readable and components easier to test by employing a dependency injection technique.

4.2 CBT web service

The CBT web service exposes a CBT Restful API interface, which is used by CBT agents. It also provides a HTML based front-end for user operations, such as controlling device sharing settings, executing test-scripts and other. More importantly the CBT web service facilitates all the logic of the system: from device registration to test-script and target application uploading and managing test executions. The overall structure of the CBT *web service* is shown in Figure 4-2.

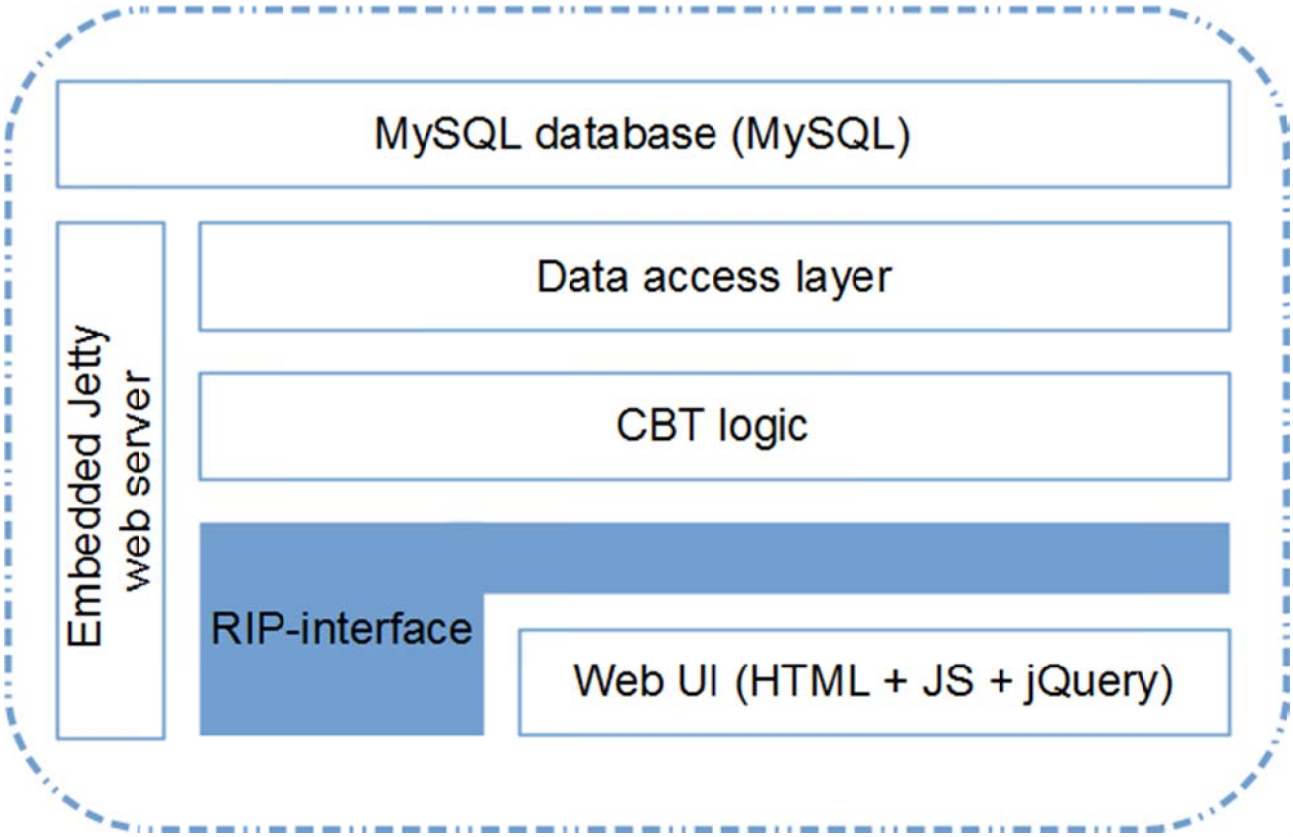

Figure 4-2 CBT web service block diagram

A database is used to store information (such as user credentials, device sharing settings, and test execution related information). None of the state information is kept in the memory of the server, thus enabling easy scalability. To promote scalability a database is used to store all real-time information, such as device status provided by agents, jobs assigned to devices, etc. This makes the database a very important system component. There are many industry standard tools and techniques for ensuring stable database operation and replication which can be used in a production system to ensure stable operation. Details of scaling the database are not in the scope of this master's theses and will not be discussed further. However, section 4.2.5 describes in detail the database's role and its structure.

The web service was developed using common web standards. It uses free/open-source tools to implement functionality such as the web server and database, so no software or license fees apply. The web server is used not only to serve web pages, but it acts as an interface for communication with the CBT *agents*. This approach was chosen because it leverages on large selection of the industry standard tools, for example web browsers and numerous HTTP and HTML related plugins.

Jetty[50] is an embedded Java server and was used to make web service deployment easier, since a packaged application is a simple Java application and does not require being uploaded to a servlet container application.

Jersey[51] framework for Restful web interfaces provided a great code structure and services for implementing Restful API's. In fact, the same API was used by the web page which is to be the

main interface to the system and by the CBT agents managing Android devices in order to re-use the code as much as possible.

The following sub-sections are arranged starting from the highest abstraction layer components to the lowest. This way, the user interface is explained first as it gives an overview of all the functionality that is available along with some insight into what is actually happening in the background.

4.2.1 CBT user front end

CBT user interface (shown in Figure 4-3) is a web interface for service users. It can be broken down into several parts based upon functionality:

- User authentication and registration
- Device access control – view which devices a user has access to as well as share devices owned with other users.
- Test file upload – packaging and uploading of file required for test executing.
- Test configuration – selecting which device types a test should run on, which mode, which test and on which target application.
- Test execution – ordering a test execution and monitoring real-time status.
- Test result analysis – analyzing received test results.
- Service usage statistics

The following sub chapter will describe each of the presented functional elements.

4.2.1.1 User authentication

Implemented authentication mechanism is very simple. Users begin by registering on the web service using a registration form. This registration form is provided as an option in a "login" dialog. Users are authenticated by providing their name and password. This information is hashed using MD5[52], a message digest algorithm, and sent as a Cookie[53] with each request to the web service. This is far from a secure implementation, but it was sufficient for prototype level functionality. This authentication is needed to provide a simple authorization scheme to determine which user can utilize which devices for test execution. Similarly authorization is used for accessing uploaded files and other content created by user.

4.2.1.2 Device access control

After logging into the system user is taken into default page called "*devices*" which displays a list of available devices. It is broken into two parts:

- Devices that are owned by the logged in user. Information is taken from database that is periodically updated by CBT agents owned by that user.
- Devices that are available for a user as other users have shared this device with him/her.

Information that is presented for each device includes:

Device ID This is a unique device ID used inside the system. This information is not particularly required or used by the user, but is presented only for the purpose of easier debugging.

Serial number The serial number is a unique name for the device. Depending on the vendor, this information might have different formats. In the case of Android emulators, the name is generated for each new emulator so that all the emulators running on the same machine have different names. This information is not really used inside the system, but servers as a human readable identifier for the user to use when referring to a device.

Status Device status depends on the time passed since the last status update received from a CBT agent. The status can be ONLINE or OFFLINE.

There is a background process running which periodically checks if device has gone offline or not. A timeout value is set in the *CBT web service* upon startup and all of the *CBT agents* are responsible for periodically updating the status of each device that is registered on the system that this agent is running on.

Last updated This shows the time of the last device status update.

Each device is owned by a certain user. This user is provided with addition control button which takes him to *device sharing* page. This page lists users which currently have access to device and provides a selection box to choose among the entire set of existing registered users within the system to add them to this device access list. Selected users will immediately get access to that device and will see it on their *devices* page.

All of the above information is stored in a database to avoid session state and provide fast access.

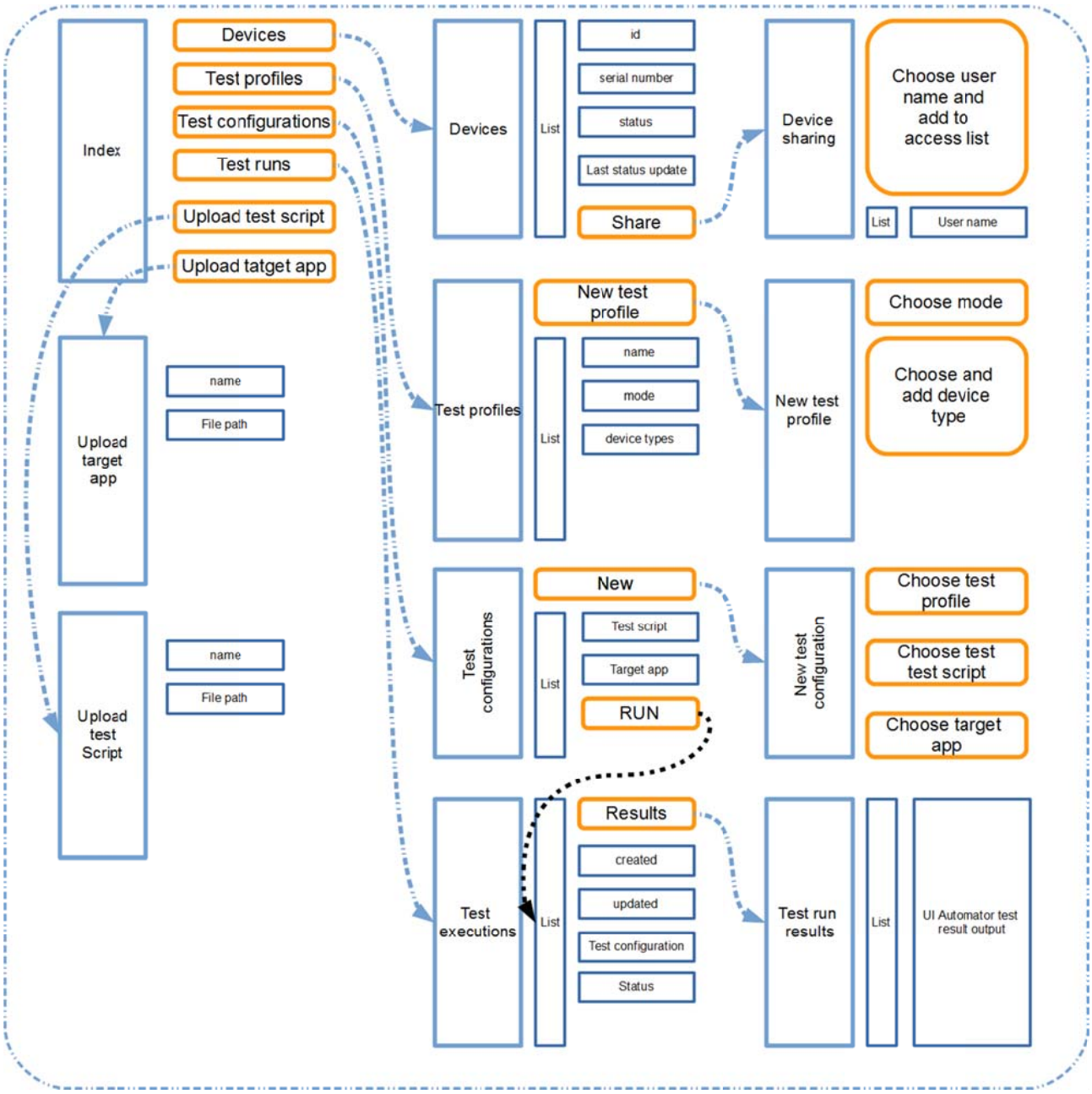

Figure 4-3 CBT web service front-end sequence diagram

4.2.1.3 Test file upload

As it was mentioned before, a test-script written for the *uiautomator* testing framework is independed (both code and package wise) of the actual target application. Therefore, in theory, one test-script can be designed to test multiple applications. For this reason, test script is treated as a separate entity throughout the system. As a result, the minimum that is required to perform a test on a device is the actual application (the target app) packaged into an APK file and *uiautomator* test-script packaged into JAR file. There are separate web pages for uploading each of these. The user interface allows a user to provide a custom name for these two files, so that they will be able to identify the files later. These files are stored in a local file system of the sever hosting the web service application. Dedicated records in the database are created for each type of file containing metadata such as name and path to actual file. Currently, prototype implementation supports only *uiautomator* type test-scripts, so no other type of files is currently needed or supported.

Figure 4-4 Test file upload flow shows how a file upload is actually performed: a HTML[54] form is used to encapsulate all the information and an HTTP request with POST[55] method is sent to certain URL belonging to a CBT RIP interface. Files are then placed according to the indicated file structure and the required records are inserted into the database.

No further file management functionality is provided (such as update or delete) since this it was not deemed essential for the prototype.

4.2.1.4 Test-configuration

The test configuration part is split into two parts in order to facilitate re-usability of content throughout the system. First, a test-profile must be created describing test execution mode and targeted devices. This test profile is accessible on a per user basis. If a user does not have any profiles created, then the user must create the first one. This test profile can be reused in other test-configurations.

The web page *test-profiles* display all of the available test-profiles. The following information is displayed about each profile:

ID	database generated unique test-profile identification number
Name	custom name given by user upon creation
Mode	test execution mode chosen upon creation; either normal or fast
Device types	list of device types chosen upon creation

This same page provides a link to a page where a new test-profile can be created. In this new test-profile creation page the user is requested to provide the information specified above. A list of available device types is displayed containing all of the different device types that are currently registered in the system. However, this list does not reflect the availability of devices which is evaluated dynamically at execution time. User selects his desired test execution mode from a drop down list. Once a test-profile is created, it will be listed on the test-profiles page and will be available for selection in other related pages, such as test-configurations: new test-configuration.

When a *test-script* and a *test-target* are uploaded and at least one *test-profile* has been created, then user can create the last piece of information required, that is new *test-configuration* which binds together these three components. This approach was chose in order to be able to quickly create a new test-configuration with, for example the same test-script and test-target but a different test-profile.

The *test-configuration* page lists all of the available *test-configurations* (to be more exact, all of the configurations created by this specific user) and provides a link to page where the user can create a new one. Each *test-configuration* is a binding between a *test-script*, a *test-target*, and a *test-profile*. This page also contains an action button next to each list element entry which invokes a new test-execution and creates a new entry on the page *test-executions*.

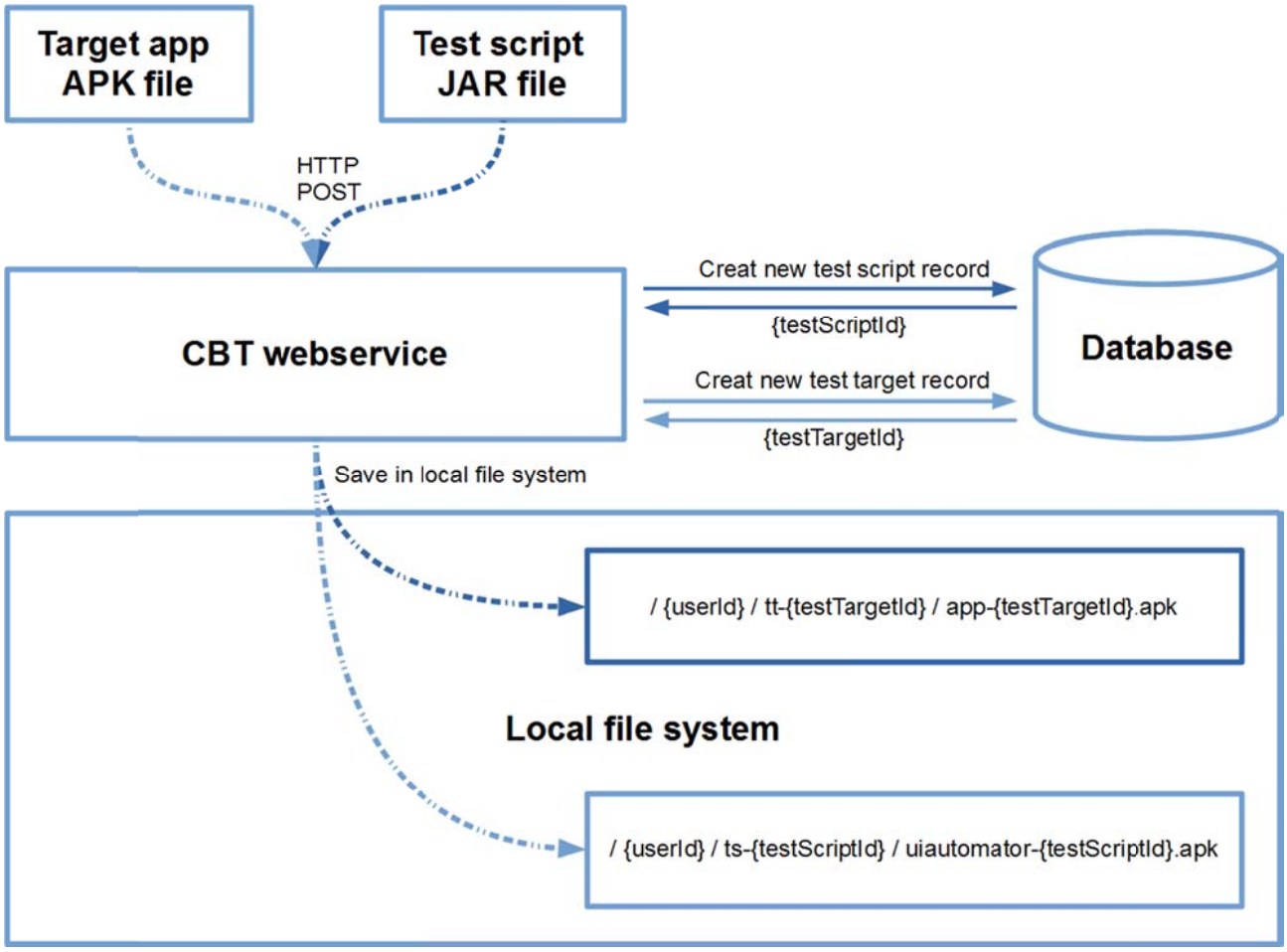

Figure 4-4 Test file upload flow

4.2.1.5 Test execution

New test execution is requested via the *test-configurations* page. Once new execution is requested, a new entry appears on *test-executions* page. Each entry contains the following information:

ID	Unique identification of test execution generated by the database. This is a key that binds together all of the information for this particular test execution.
Created	Date of test execution launch
Updated	Date when record was last updated
test-configuration	Reference to *test-configuration* information that was used for the test.
Status	Status of test execution. The value can be:

WAITING	Test has been requested, but none of the CBT agents have started executing it on actual devices.
ONGOING	At least one CBT agent has reported that this test execution has started.
FINISHED	All CBT agents have reported that they have finished the test execution.

It is important to briefly describe the internal processes of the system in order to explain how test execution is performed: in order to execute a test, first, system needs to find devices that are currently *online* and available for test execution. For each of these devices a test execution descriptor record in the database is created called *device-job*. CBT agents periodically poll for new *device-job* records through the CBT RIP interface. Once a new *device-job* record is found, CBT agent starts this test's execution on a device and reports its status when process is finished. As a result, the test execution status and results are available per *device-job.* Once all *device-jobs* have been reported as finished, then the overall test execution state is changed to FINISHED. The detailed operations are discussed in section 4.2.3.2 on page 23.

4.2.1.6 Test result analysis

As it was mentioned previously, test result is available on a per *device-job* or device basis. Therefore, *test-result* page contains a list of test execution results of each device. These results are available as soon as a CBT agent publishes them via the CBT web service. Currently, results are displayed in a raw format: unmodified output of the *uiautomator* test framework test execution. These results are parsed to extract information, such as how many tests were executed and whether their outcome was pass or fail. This is sufficient to show the possibility to retrieve and interpret the test results. Currently this test result information is limited and includes only the output produced by *uiautomator* framework and does not include access to the device logs or screenshots, as it is common in similar services. This functionality was not implemented due to limited amount of time although no obstacles were discovered for it to be implemented in future.

4.2.1.7 Security

Security in a broad sense was outside the scope of this master's thesis project. However, here we mention at least some of possible problems which could be addressed in future work.

Communication between front-end and back-end (CBT RIP) as well as between CBT agent and CBT RIP is insecure since it simply uses HTTP. There are no additional checks performed to see if the user issuing a request is allowed to make such a request. For example, it would be possible to delete other users' information by manipulating the information contained in a HTTP request.

The CBT web service and CBT agent treat test scripts and test targets as packages and therefore, do not scan their contents for potential malicious content. These limitations are discussed further in chapter 0 and possible solutions are presented in section 6.2.1 on page 31.

4.2.1.8 Service usage statistics

Service usage statistics per individual user are very important for all modern social web services and the CBT web service is no exception. Depending on the business model(s) used for this service, it could be necessary to collect the number of test executions that a user hosted on each of his or her devices that have been shared with other users and to collect the number of tests per device that a user has executed on other users' devices. Although no logic was implemented to make use of this kind of information, for example to limit the number of test executions for certain users based on their usage of the service, a web page showing some of the parameters that could be collected was implemented as an example of what information is or could be collected. This page currently calculates and presents the following information on a per user basis:

- List of user owned devices with the number of tests hosted for other users.
- Total number of hosted test executions.
- Total number of tests executed on devices owned by other users.

Information is update in real time after relevant actions are performed by user.

4.2.2 CBT RIP interface

The CBT RIP interface was build using the Jersey[51] framework as it enabled rapid development of Restful APs. This interface exposes all of the CBT web service's functionality

through a client request – server response HTTP based protocol. A single CBT RIP serves both CBT agents and the CBT front-end in order to avoid duplicated functionality. The implementation uses JSON[56, 57] for data serialization: JSON makes it easy to read and debug HTTP requests with standard tools as all of the data is in clear text and human readable.

The API's URLs are grouped by functionality. These URLs are served by their responsible classes. Some examples are given here in order to briefly explain how this works:

- In order to retrieve a list of user *test-runs* an HTTP GET request is made using a URL of the form: http://serveraddress/userid/testrun, where *serveraddress* and *userid* are replaced with appropriate values. This request would return a JSON array of *test run* objects with their respective properties. This request must carry a COOKIE containing the user's credentials encoded so that the server can authenticate the request. Generally, a user is only be able to retrieve his or her own data.
- In order to create a new *test-profile,* an HTTP PUT request is made using a URL of the form: http://serveraddress/userid/testprofile, where *serveraddress* and *userid* are replaced with appropriate values. This request carrying the data required for creating a new *test-profile* and this data must be serialized into a JSON format and sent as a HTTP payload. As in the previous example, an HTTP cookie must be sent providing the user's credentials.

All other API calls are performed in a similar way and could be easily discovered looking at the source code. There are two groups of these calls – those that must carry an authentication HTTP cookie and those that do not. One of the unauthenticated API calls is for creating a new user. This API call is used by web front-end to register new users. For this reasons, it requires no initial authentication so that it would be accessible by un-authenticated users.

4.2.3 CBT web service's inner workings

Although main function of a web service is to read and write to/from database when certain actions are triggered by a user or CBT agent, sometimes the server must perform a much more complicated task than simply adding or reading information to/from the database. One of these functions is analyzing tests JAR file contents (subchapter 4.2.3.1) to extract names of compiled test classes which serves as an input data to another function – distributing and parallelizing test execution (subchapter 4.2.3.2) on multiple Android devices. Another important function is to temporarily store (subchapter 4.2.3.3) target application and test files in servers file system. This section discusses these kinds of actions and describes the workflow that occurs after these actions are triggered.

4.2.3.1 Analyzing UI Automator JAR file

In order to be able to split, distribute and execute tests on many Android devices, CBT web service needs to be able to read and interpret the uploaded test-script package. Therefore, this section describes the structure of test package and compiled source code format of test classes written using the *uiautomator* framework and tools that were used to read and interpret this format.

Although the test-script is packaged into the well-known JAR[58] package format, it does not contain multiple classes converted into standard Java byte code[59]. Because Android applications run on a Dalvik virtual machine (VM), the applications and tools written in Java must be packaged into format executable by the Dalvik, i.e., a Dalvik executable[60]. The details of DEX are outside the scope of this thesis project, but it is important to note that since DEX differs from the well-known Java byte code[59] there are no tools embedded into the usual Java environment that can be used to read it. Although Android development environment does have tools[61] for dealing with DEX format, none of it could be used due to the fact that all of the parts of this project were developed outside the Android platform using standard Java tools. For this reason an open source library called Smali[62] was used to parse uploaded *uiautomator* test packages.

Test package JAR file is an ordinary ZIP file that contains a file called *classes.dex* which is the file that is needed to be parsed. DEX file structure is shown in Figure 4-5 as well gives some hints

into how test classes should be structured. In presented example, each test is coded into separate class and named according to the function that is being tests (naming does not matter). Fortunately, the Smali library has all that it is needed to extract the names of test classes from the *classes.dex* file and this is the only information required to split the test classes.

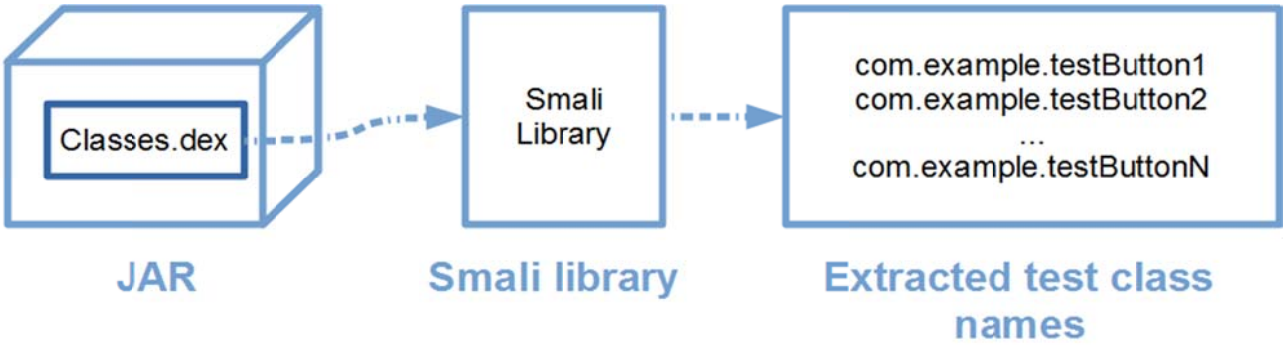

Figure 4-5 DEX file structure

When a new test package is uploaded, it is parsed in order to extract the test class names and these names are stored in the database as metadata associated with this file. The next section describes how this metadata is used to speed up the execution of tests.

4.2.3.2 Fast and normal execution modes

As it was mentioned before, there are two test execution modes for user to choose from: *fast* and *normal*.

Using normal execution mode, all test classes contained within an uploaded *uiautomator* packed JAR package are executed on each device type which is chosen in the bound test-profile. This is called normal mode because it is assumed that this is how tests will usually be executed, especially when there are a limited number of devices to run the test on. However, when there are many devices, to be more exact, many devices of the same type, different parts of tests can be spread among many devices and executed in parallel. This is important since the goal of testing is to check that tests of an application succeed on as many different types of devices as possible.

Figure 4-6 presents the steps and actions taken for *fast* execution mode. As an example, a test package contains 4 test classes. Each test class is for testing different application features, in this case, 4 different button clicks. After test execution in initiated by user via the *CBT web service's front-end,* database is queried to get all of the available devices of the types defined in the relevant *test-profile*. A simplistic algorithm was developed and used to distribute test classes over all of these devices, as a result *device-job* records are created in database containing list of the specific test classes to be executed on dedicated Android device. These *device-job* records are later used by *CBT agents* to execute test classes on the *test-target* application defined in the relevant *test-configuration.*

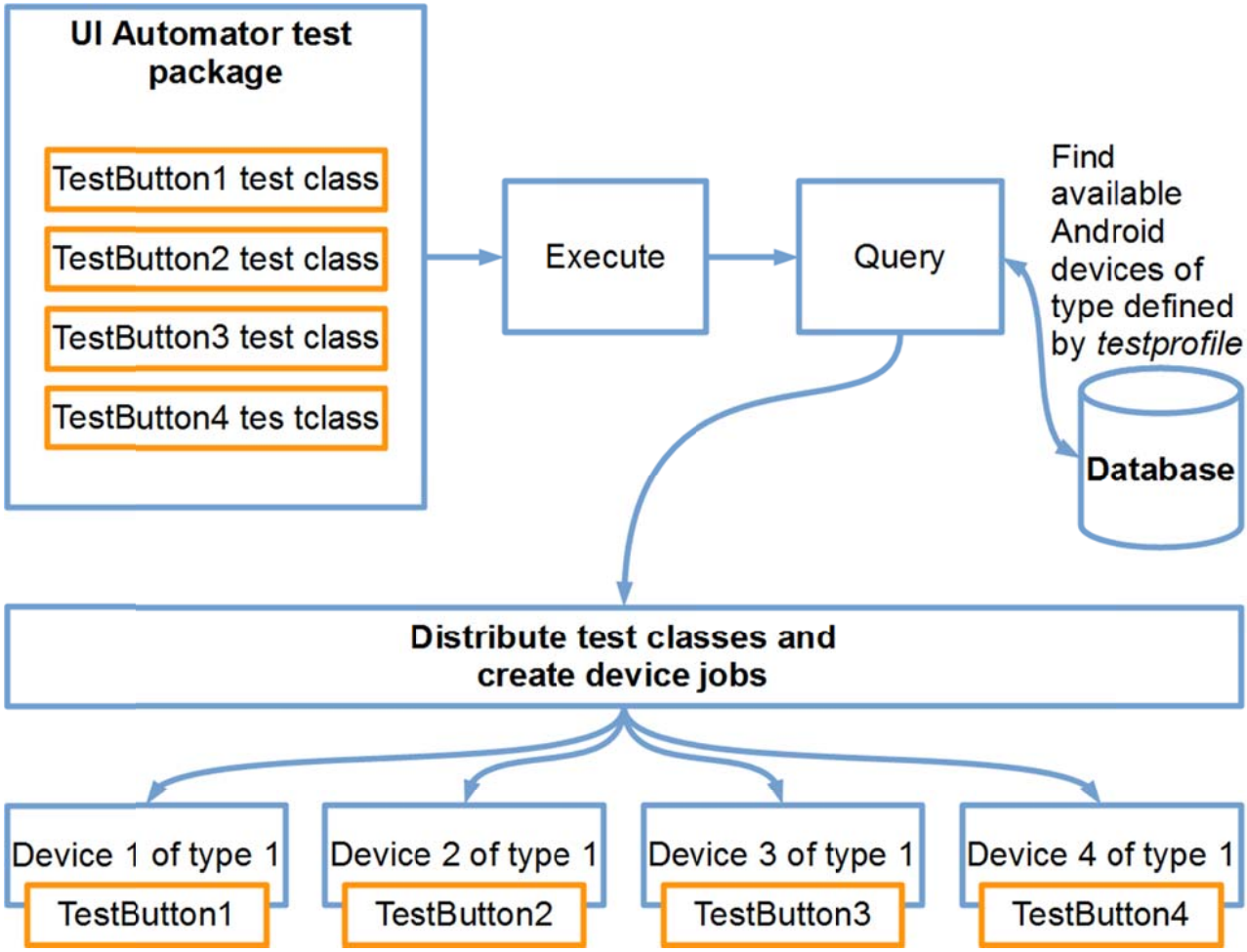

Figure 4-6 Fast execution mode internals

4.2.3.3 File storage structure

The *CBT web service* needs to store *test-scripts* and target application files so that *CBT agents* would be able to fetch them when executing tests. For this purpose, server's local file system and files are organized in a simple hierarchical folder structure which is presented in Figure 4-4. When a user uploads a new file, before storing it in a local file system, a record is created in the database. Each new record is assigned a new unique identifier by the database and this identifier is used later when creating the required folders and naming the files. If a folder does not exist, then a folder is created with a name based upon the user's ID who issued the request. Then, depending upon whether a *test-script* or target application was uploaded, folder named ts-{testScriptId} or tt-{testTargetId} is created. Finally, uploaded files are placed into the folder created in previous step and renamed to match pattern "uiautomator-{testScriptId}.jar" for *test-script* package and "app-{testTargetId}" – for target application. After placing the files into correct locations their respective database records are updated with the exact file path pointing to those files. This is necessary in order to be able to locate these files based on their metadata in database.

4.2.4 CBT data access layer

None of the *CBT web service* components access the database directly, but rather they access the database through a special data access layer[63], which is composed of data access objects[64] that provide access grouped by functionality. For example, there are data access objects for manipulating user, device, *test-script*, and target application related information in database. This approach is used for several reasons:

- Unit testing is easier since the connection to database can be easily replaced with stubs/mocks/or a in memory database that is pre-populated with known data.

- Components using these data access objects are not aware of where the data is coming from, so the data could be store in files, memory, or any other medium.
- This approach provides a better overall structure for the source code.

Most of the data access objects (DAO) deal with an underlying MySQL database. There are lots of tools and libraries to access MySQL and other databases from Java application code, such as Hibernate[65], JDBC[66], jooq[67], etc. Hibernate is a very sophisticated and feature rich persistence library and is widely used in large scale corporate applications and its API is quite different from actual MySQL query language. However, not all MySQL features are supported by Hibernate. Jooq on the other hand, is more lightweight product and provides almost direct mappings to SQL[68]. Because of this, Jooq was chosen for in *CBT web service* application and was successfully used in DAO's.

4.2.5 CBT database

MySQL[69] was used as the underlying database as it is one of the most popular[70] free databases . The database is composed of a number of tables containing information ranging from system's user names to results of test executions. Table 4-1 lists the names of the tables and briefly describes what kind of information is stored in this table.

Table 4-1: CBT MySQL database tables

Table name	Contains
device	Information about Android device: serial number, unique ID, state (online, offline), the ID of the operating system version, ID of device type, and time of last record update
device_job	Information describing the tests to be executed on specific devices, including device ID, *test-run* ID, dates of creation and update, and execution status.
device_job_result	Information about the result of a *device-job* execution: state (passed, failed), number of tests run, failed, errors, and ID of the *device-job.*
device_sharing	Information binding user's ID and the IDs of devices that have been shared with these users
device_type	Device model name, manufacture's name
testconfig	User ID, name, *test-script* ID, *test-target* ID, *test-profile* ID
testprofile	User ID, mode (*fast*, *normal*), name
testprofile_devices	Binds together *test-profile IDs* with device IDs
testrun	User ID, test-configuration ID, status (waiting, running, finished), and dates when it was created and last updated.
testscript	User ID, name, file name, path to file, and list of test class names
testtarget	User ID, file name, and path to file
user	User ID, user name, and password

4.3 CBT agent application

The *CBT agent* application was developed to run on various platforms using Java programming language. Android SDK must be installed on the machine where the client executes because of the use of Android Debug Bridge (ADB) for communication with Android devices. The agent's main tasks include:

- Detect and register new devices in *CBT web service*.
- Monitoring device status and update it in the *CBT web service*.
- Poll for new *device-jobs's* assigned for devices managed by this agent.
- Download files required for test execution.
- Install downloaded files on the device in order to prepare for test execution.
- Execute the tests
- Collect test results and publish them via the *CBT web service.*

The following subsections will describe the application in detail following inner process flow (illustrated in Figure 4-7): will start by explaining how an application is started, what configuration properties can or must be given, then an explanation of how devices are monitored is given.

4.3.1 Starting the application

The CBT agent application requires certain information to be provided by the user prior to execution. This information must be present in a file called `client.properties` and should be placed in the same directory as the application itself. Table 4-2 lists and describes the properties contained within this file.

Table 4-2: CBT agent startup properties

Property name	Description
path_adb	Path to ADB application which is provided together with Android SDK
path_workspace	Path to a directory which will be used as a workspace for temporarily storing files needed for test execution. Files include *test-target* and t*est-script*
uri_server	URL to *CBT web service* in the form: `http://address:port`
username	User name for authentication
password	Password for authentication
debug_rest	Flag indicated whether debugging information for each HTTP request should be output. The possible values are `True` or `False`.

Here is an example of such a file's contents:

```
path_adb=/home/dev/adt-bundle-linux-x86_64/sdk/platform-tools/adb
path_workspace=/home/saulius/Documents/cbt/client_workspace/
uri_server=http://127.0.0.1:8081
username=John
password=JohnsPassword
debug_rest=true;
```

When correct information is provided in the `client.properties` file, the application can be started from the command line by executing standard Java application launching commands.

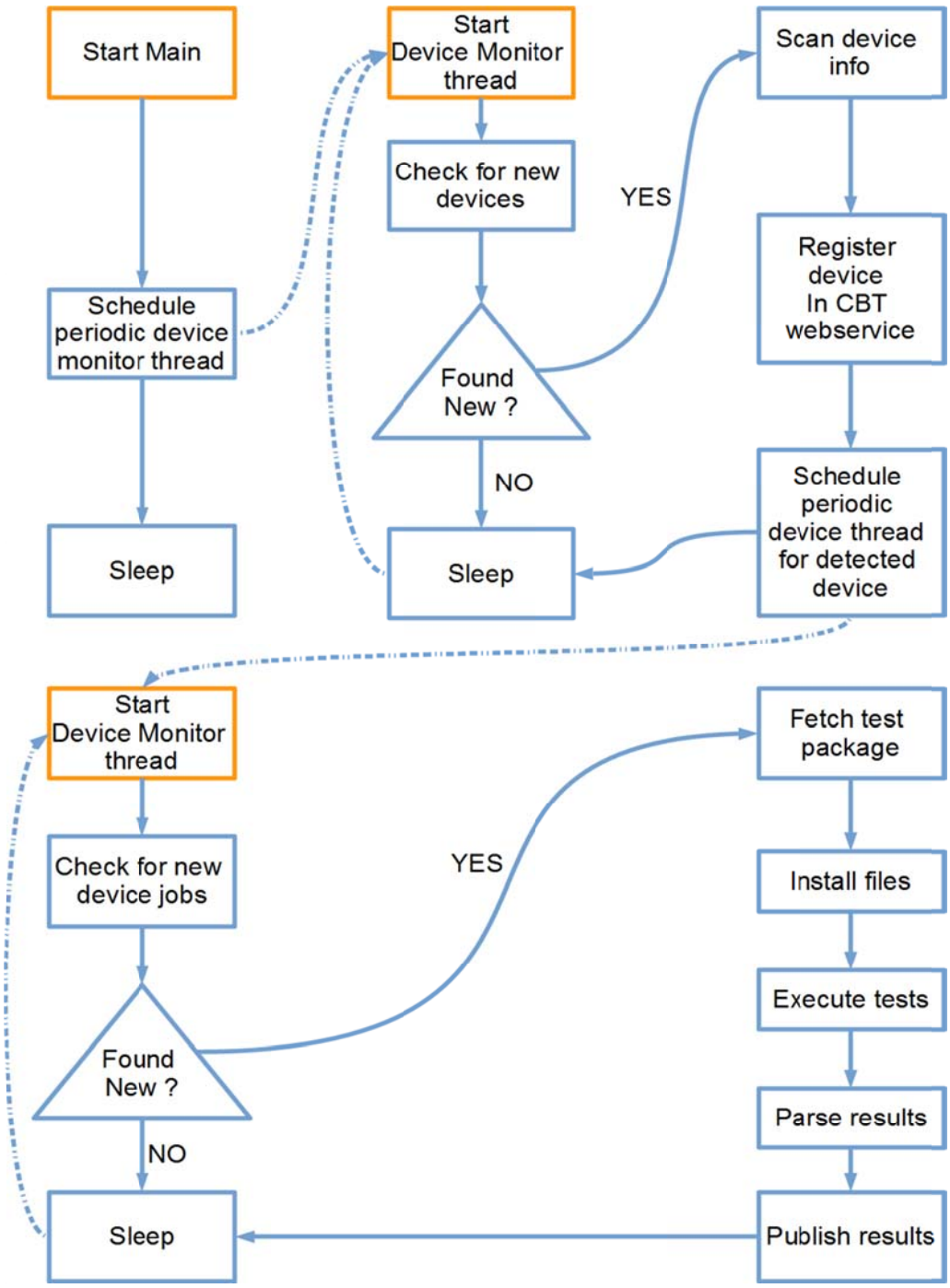

Figure 4-7 CBT agent application flow diagram

4.3.2 Device monitoring

A task that executes in a separate thread periodically checks what devices are currently connected to the host system. This is done by executing the ADB command `devices`, the output of this command lists serial numbers of currently connected devices. *CBT agent* keeps track of which devices have already been registered and if a new device is found it initiates a new device registration procedure. Otherwise this task simply sends an update request to *CBT web service* which acts as a heartbeat for this particular device since last update time is save in database. These update messages are very important so that the system would have some assurance of how many devices are actually online at a particular point in time. This information is used when preparing for new test runs and when selecting devices to run tests on.

Since the *CBT web service* differentiates between devices based on notion of *device type* (which is actually the device name defined by the manufacturer) *CBT agent* needs to acquire this information in order to register new device. The following ADB command is executed to get this information from the device: `shell cat /system/build.prop`. This command returns quite a long list of various device build properties. A subset of these properties are:

```
...
ro.build.type=user
ro.build.user=
ro.build.host=ABM032
ro.build.tags=release-keys
ro.product.model=HTC Desire S
ro.product.brand=htc_europe
ro.product.name=htc_saga
ro.product.device=saga
ro.product.board=saga
ro.product.cpu.abi=armeabi-v7a
ro.product.cpu.abi2=armeabi
ro.product.manufacturer=HTC
ro.product.locale.language=hdpi
ro.wifi.channels=
ro.board.platform=msm7x30
...
```

The properties above were retrieved from a HTC Desire S Android which is evident from property named `ro.product.model`. *CBT agent* requires values of two highlighted properties which are used to define the *device type*. A request to the *CBT web service* is issued to acquire a unique ID number for defined *device type* (the ID is composed of the device model and manufacturer's name). The *CBT web service* searches for and returns an ID of a matching record or creates new record and returns its ID. Now the device registration can proceed which sends another REST API request that creates the required records in the database this way binding device to the user who issues the request.

4.3.3 Downloading and executing a test package

CBT agents decide when and which tests to execute by periodically polling the *CBT web service*. When new test job record is found for one of the devices managed by an agent as a response to a polling request (response also contains the data required for further actions.) a request is sent to download a test package's metadata and to download the test package itself (a ZIP file containing *test-script* and *test-target* files). All of the local files will simply be overwritten as no sophisticated management of the client's workspace files was implemented.

The test package contents are extracted into a workspace directory which is defined by one of the properties provided when the agent application was started. In order to execute a test, device needs to be prepared. This preparation includes:

- Copying a *uiautomator* test-script JAR package to device. This is done by executing the ADB command: `push <name>.jar /data/local/tmp`.
- Installing the target application by executing the ADB command: `install -r <target-app-name>.apk`. Since the application might already be present on the device because of a previous test run, the "`-r`" option is given which instructs ADB to re-install the application if it is already present. This might not always be a desired behavior in production level system, but it suits the purpose of this prototype and it ensures that the application is freshly installed at the start of each test.

The actual test execution occurs just after the device preparation phase is finished. The ADB command: `shell uiautomator runtest <name>.jar -c <testclasses>` is executed in order to start the *uiautomator* test execution. The agent will explicitly list all of the test classes to be executed based upon the *device-job* metadata. In this way the *CBT web server* controls exactly which test classes are executed.

The agent does not halt during a test's execution since each device gets its own thread for all operations. This enables tests to be executed in parallel on many devices managed by one agent.

4.3.4 Test result reporting

The *uiautomator* testing framework is in its early stage of development. One evidence of this is that test execution result comes in only one form and it is quite hard to understand and parse. The result is just a stream of key-value pairs returned via STDOUT. Nevertheless, this information includes:

- Number of executed tests.
- Class names of executed tests.
- Number of failed tests.
- Number of tests with test execution errors.
- Stack printout in the case of assertion errors.

In addition, custom key-value pairs can be passed from the test case to the test result output. An example of the test result output printout from a *uiautomator* test executions is presented below:

```
INSTRUMENTATION_STATUS: current=1
INSTRUMENTATION_STATUS: id=UiAutomatorTestRunner
INSTRUMENTATION_STATUS: class=com.test.TestButton1
INSTRUMENTATION_STATUS: stream=
com.test.TestButton1:
INSTRUMENTATION_STATUS: numtests=1
INSTRUMENTATION_STATUS: test=test1
INSTRUMENTATION_STATUS_CODE: 1
INSTRUMENTATION_STATUS: dp-height=800
INSTRUMENTATION_STATUS: product=sdk_x86
INSTRUMENTATION_STATUS: msg=Test was succesfull
INSTRUMENTATION_STATUS: dp-width=480
INSTRUMENTATION_STATUS_CODE: -1
INSTRUMENTATION_STATUS: current=1
INSTRUMENTATION_STATUS: id=UiAutomatorTestRunner
INSTRUMENTATION_STATUS: class=com.test.TestButton1
INSTRUMENTATION_STATUS: stream=.
INSTRUMENTATION_STATUS: numtests=1
INSTRUMENTATION_STATUS: test=test1
INSTRUMENTATION_STATUS_CODE: 0
INSTRUMENTATION_STATUS: stream=
Test results for UiAutomatorTestRunner=.
Time: 9.284

OK (1 test)


INSTRUMENTATION_STATUS_CODE: -1
```

In order to evaluate if tests passed or failed, the application needs to parse the test result output and extract required information. Since this information is not very well structured, regular expressions[71] are a great help in this situation. If one or more of the tests failed, then the overall test result is considered to be failure, otherwise, an indication of success is sent to the *CBT web*

service together with the overall result. All of the test result output is sent in its raw format - this way user can examine the printout to determine the cause of failure.

Functionality of application is limited and does not implement sending device logs, screenshots, and other files to the *CBT web service.* In some cases this additional information would be required to determine the result(s) of test execution manually by tester examining collected information. For example, if the tests were supposed to verify that target application's layouts were correct on different types of devices, screenshots could be made at certain points of test execution and in the end, result of this test would probably need to be determined by a tester examining the screenshots.

In spite of the test result output being unstructured, it does provide all information required to determine if and which test passed or failed, as well as information that could be used to locate the part of the code that failed using the stack trace back.

Published test execution results from each device-job are further processed by the *CBT web service* and aggregated.

4.4 Demo application and test script

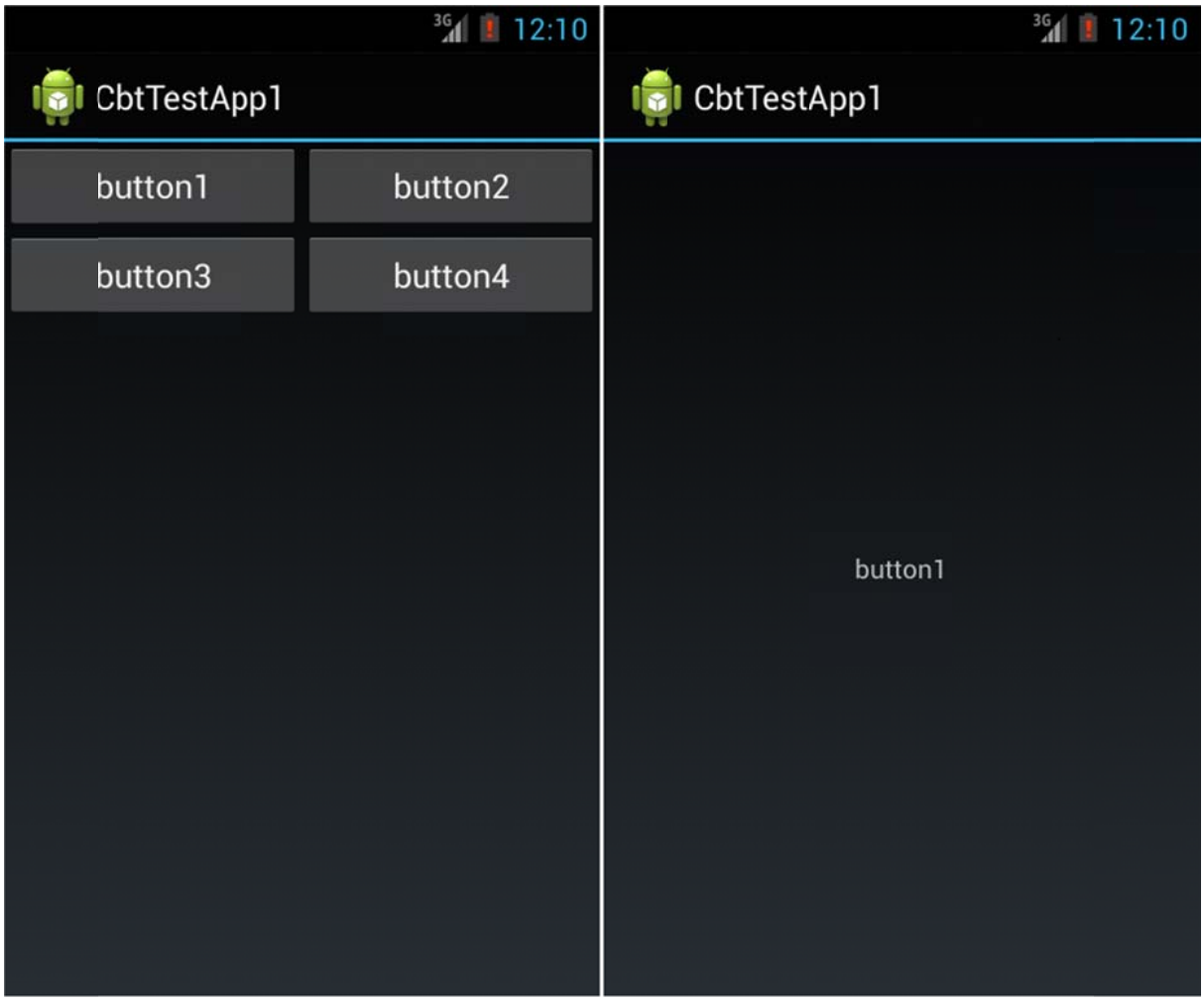

Figure 4-8 Demo app screenshots

In order to test developed system, an Android demo application as well as *uiautomator* based test application to test the app were developed.

Android demo application is a very primitive application designed only for testing *CBT system*. It consists of two *activities*:

- First *activity* displays 4 buttons (Figure 4-8 left)
- Second activity (Figure 4-8 right) appears when any of the 4 buttons belonging to first *activity* are clicked and displays text which differs depending on which button was pressed.

Buttons of the first *activity* can be interpreted as different functions of application that can be tested separately on multiple devices. Therefore, 4 different tests can be executed in parallel on different devices that test the functionality of specific button.

Test for this app was developed in a way that would enable best use of CBT system. Therefore, it consists of 4 test classes each of them testing different buttons. Figure 4-9 presents the flow of test script actions: initially emulates a user activating "home" button that belongs to Android OS, then enters the "app list", locates demo app by its name and launches it. Then, depending on which one of the classes it is, it clicks one of the buttons, verifies that new *activity* appears and that correct text is displayed (clicked buttons name) and closes application.

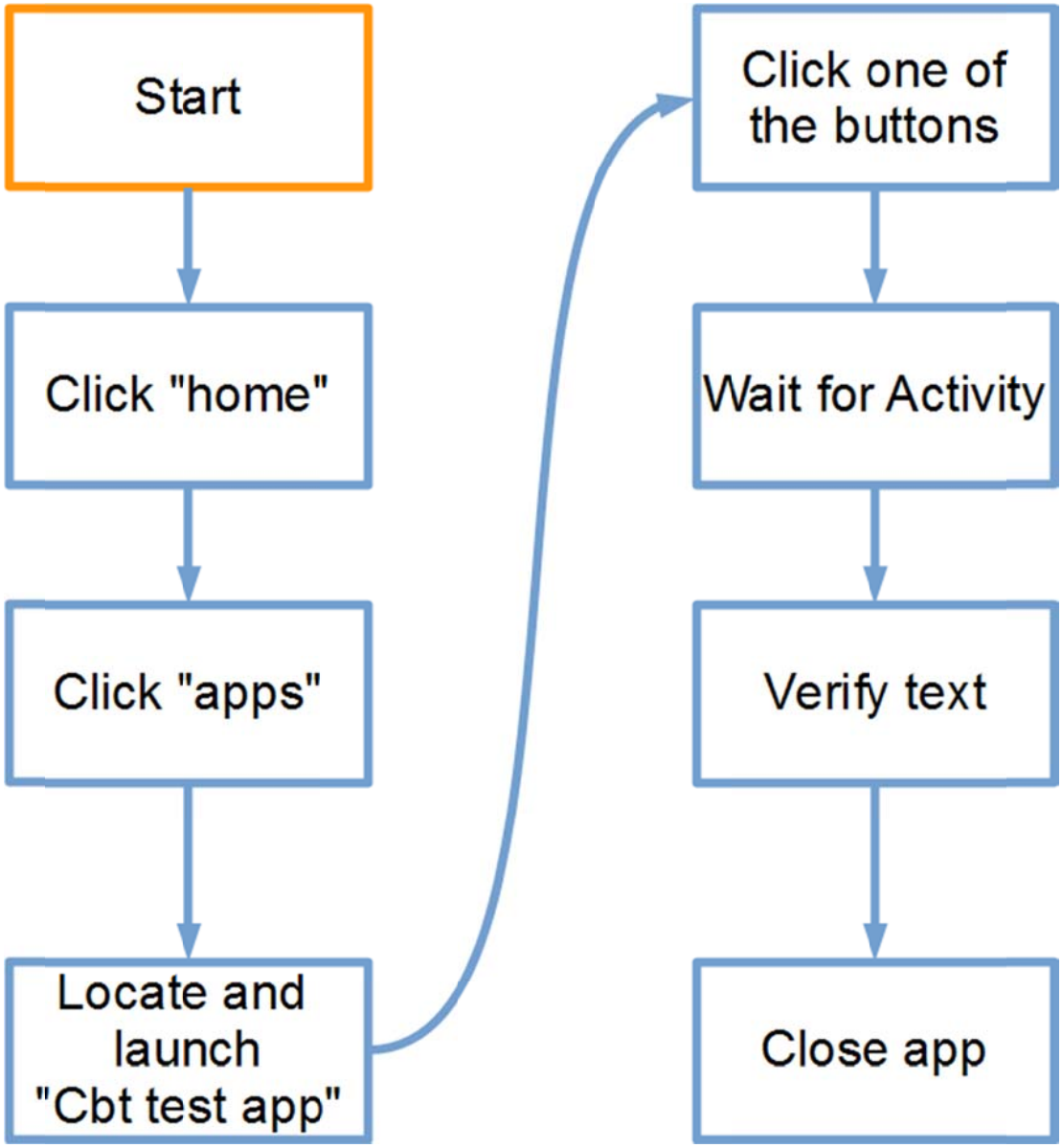

Figure 4-9 CBT demo app test flow

sses

5 Discussion

The main goal of this master's thesis was to implement a prototype of a community based testing platform that exploits Android device sharing possibility for test execution. This goal was successfully reached by investigating how testing of Android applications is performed, analyzing similar platforms, and implementing a prototype of a web service and client application. While this prototype lacks some functionality, it provides the essential core features such as user and device management, test distribution, execution, result collection, and presentation of the testing results that enable multiple users to share Android devices and executed automated test suites.

5.1 Performance

CBT system performance can be measured by measuring performance of certain events. Events can be uploading files, downloading files, ordering a test execution and etc. However, most of these measurements depend greatly on external factors such as network connection speed, hardware used for hosting a web service and running *CBT agent*, sizes of files that need to be exchanged, load of the CPU and other processes that are running in parallel during execution of the test on *CBT agent* and possibly other factors. Due to variable nature to these factors, it is very hard or even impossible to measure exact performance of the overall system. However, some of the system performance aspects can be measure by making certain assumptions. The most important outcome of implemented system is ability to distribute test classes and run them in parallel on different Android devices in order to decrease the overall testing time. Therefore, this is the most important measurable object as well. Table 5-1 presents measured test execution times when running tests on varying number of devices and different execution modes (these results are plotted in Figure 5-1). Measurements were collected while running *CBT web service*, *CBT agent* and at most 4 Android emulators on the same PC (with Intel i5-2540M CPU and 8GB of RAM), therefore, measurements them self's do not reflect production environment, however, can be used for comparison and deriving performance gains in different situations. Demo application and test script presented in section 4.4 were used as a system payload for measurements.

The biggest difference can be seen when comparing results from executions on 1 and 2 devices in parallel which show a ~44% decrease in testing time. One could assume that if a test is split into two classes and executed in parallel on two identical devices, it should take half the total time however, measurement results were different. Figure 5-1 shows that increasing the number of parallel executions do not increase execution speed linearly which is the result of certain events being sequential, therefore, blocking parallel execution for certain fraction of overall time. Amdahl's law was applied in order to calculate what fraction of the process cannot be parallelized and estimates were calculated based derived fraction value. It is important to note that the measurements are relative to environment used for testing as well as sizes of files since these parameters have major impact on execution time.

Table 5-1: Performance comparison of different execution modes

Number of Android devices (same type)	Execution mode	Average overall execution time in seconds
1	fast	32
1	normal	29
2	fast	17
2	normal	30
4	fast	10
4	normal	31

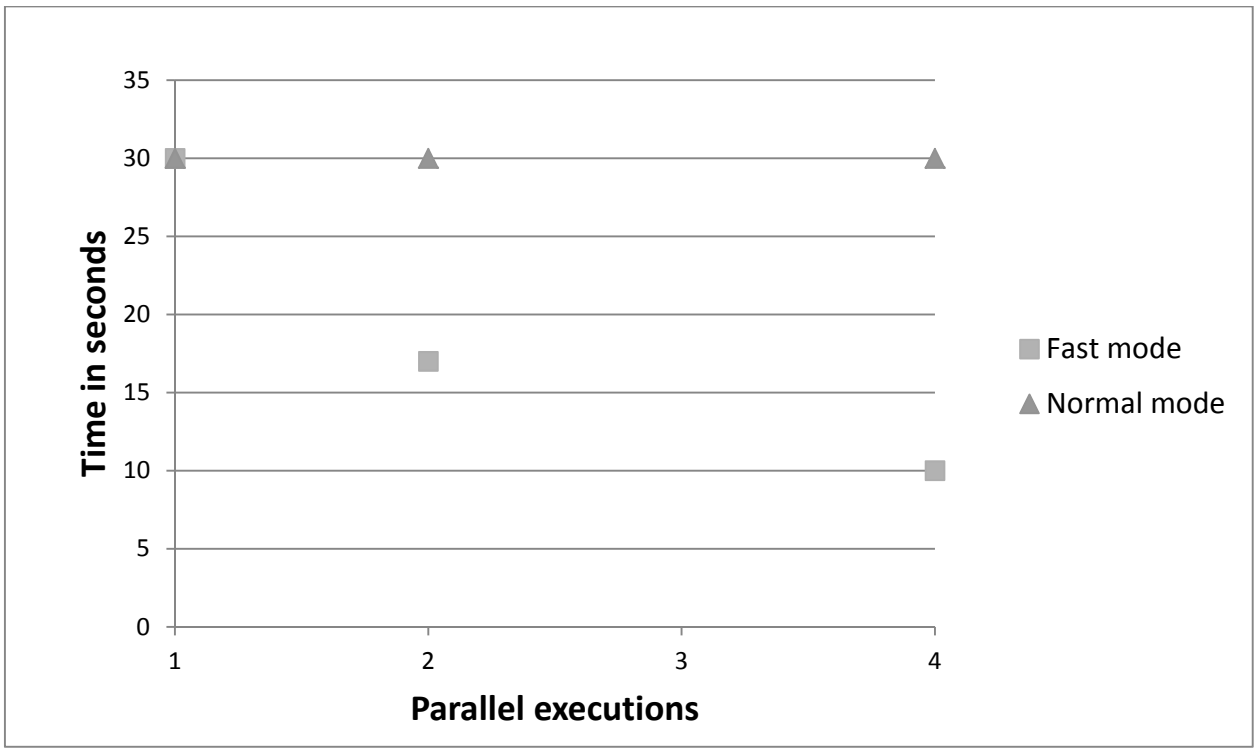

Figure 5-1 Performance comparison graph

Figure 5-2 Maximum speed-up in relation to test classes

It was calculate (from measurements presented in Table 5-1) that fraction of the process that is sequential is around 0.12 in which case, according Amdahl's law, maximum speed increase that can be achieved increasing parallel execution is equal to 1/0.12=~8 which means with each increase in a number of parallel executions, overall time will decrease less. Moreover, in this particular case (test class splitting) maximum number of parallel executions is limited by a number of test classes in test package. More estimation was done in order to reveal actual potential of the CBT system

which is presented in *Figure 5-2*. This figure presents estimated speed-up of overall test execution in relation to number of parallel executions and test classes available for parallelization. In order to calculate these estimations, it was needed to assume that fraction of sequential process events doesn't change while number of test classes changes (this is valid since test package size increase from having 4 to 20 test cases is unnoticeable compared to packages size of application itself), also, that it takes same amount of time to execute each and every test class.

Overall, performance measurements and derived estimations show that testing time can be dramatically decreased. Moreover, the more test classes are available, the more parallel executions are possible, and therefore, the higher speed increase is possible. Having tests of longer duration would decrease the fraction of the process that is sequential, therefore, increasing maximum speed-up.

5.2 Limitations

The prototype that has been implemented is not production ready. However, this was not a goal of this thesis project. It does not employ HTTPS[72] or any other techniques to provide secure communication between the systems' components and users. Since HTTP was chosen as the main communication protocol and no custom encoding is used, messages are sent in clear text and can be easily reverse engineered in order to find weak spots. At the same time, it is relatively straight forward to change from using HTTP to using HTTPS, thus this is an obvious near term improvement that could be made in the implementation.

Comparing this service to other similar services additional limitations can be found. For example, it does not provide a way to download screenshots or any other files except for the test execution's results output. Handling logs and screenshots is a necessity when testing UI layouts and might be a major obstacle preventing potential users from using this prototype.

Another limiting factor is the lack of support for other testing frameworks. As was mentioned previously, implemented *CBT web service* only supports *uiautomator* based tests. This testing framework was introduced and is available since Android OS version 4.1, thus the system is unusable to test applications running on devices with older versions of the Android OS. However, this limitation is only temporary as more and more devices are being updated to newer Android versions. At the time of writing, approximately 55% of all Android devices are already using version 4.0 or higher[73].

Most Android tools are compatible with three most popular operating systems (Windows, Linux, and OSX). Development of any new tools must take this multiplatform capability into consideration and should continue to support all these platforms, in order to reach all potential users (thus attracting a larger number and variety of volunteers who will share their Android devices for testing). *CBT agent* was built using Java so theoretically it run across a wide variety of platforms, however, development and testing was performed on Linux Ubuntu[74] only. Nevertheless, no major changes should be required in order to adjust it to run on different operating systems. The *CBT web service* was also developed on Ubuntu, but the platform for the *CBT web service* is not as important since it will most likely be deployed in datacenters which usually have no problems hosting a Linux application.

No measurements were made to determine the maximum number of Android device that one *CBT agent* is capable of handling due to the need to have a large number of Android devices to test with. However, it successfully supported 5 Android emulators and 2 hardware devices. The same computer was also hosting the web service and database applications at the same time. These tests were performed on a laptop equipped with an Intel Core i5-2540M CPU and 8GB of RAM. Possible factors limiting the maximum number of devices supported by a *CBT agent* could be limitations of the Android debug bridge application (maximum number of devices supported by ADB is not documented). Moreover, multiple instances of ADB application could be started on the same computer, this way minimizing the effect of potential bottleneck.

The *CBT web service* was built to be scalable and does not have a hardcoded maximum number of *CBT agents*, Android devices, or system users defined. Since this web service uses local file system to store tests & applications and MySQL database for real-time data and metadata, there could be a limitation due to the available free space for local file storage and the database.

No other major limitations are known to the author at the time of writing.

5.3 Required reflections

The result of this master thesis is a working prototype of the system that enables sharing of Android OS based devices within developer community, therefore, providing cheaper and faster ways of executing automated test scripts of physical and emulated devices. Cheaper, because service that could be deployed based on developed prototype would not need to purchase real physical devices, but rather, community member would give access to their owned devices.

One of the positive social effects is increased developer satisfaction by being able to execute tests script on real physical devices much cheaper than using similar services. Service also opens a possibility to radically improve feature tests by employing parallel executions on identical devices. Ability to execute tests faster would lead to increased quality of application which improves overall user experience.

Utilization of existing Android devices rather than purchase of new one dramatically decreases overall of costs of mentioned service which is a positive economic effect for developer community as well as service providers.

Malicious usage of this service opens up possibility to retrieve private information from devices that were made available to malicious user, however, further system improvements could partially solve this problem, but implemented prototype relies on a fact that developers would trust each other or would not keep any private information on devices that have been share with other developers.

6 Conclusions and future work

This chapter discusses possible future improvement as well as presents final conclusions.

6.1 Conclusion

One of the main achievements was a *fast* execution mode that enables testers to radically reduce the amount of test execution time by splitting a test suite into multiple tests, distributing these tests over many devices of the same type and aggregating the final results.

The implemented platform is a working prototype. However, it is usable as a solid base for future developments and experiments. Further improvements, mainly in the area of security, need to be made in order to bring it closer to a production level when the system could be publicly deployed.

This was a challenging and interesting project since it involved research and development in many different areas, ranging from writing Android applications and test scripts to developing a console Java application, followed by implementation of a full blown web service interfacing a database and providing a web-based user interface.

An analysis of similar services revealed that no service offers the same core feature of prototype platform, i.e., none leverage existing Android devices by enabling device sharing by the platform's users for automated test execution. However, the maturity of these competing services makes it hard to produce an adequate product during the time and resources which were be devoted for this master's thesis.

The result of this project can be used to implement and deploy a service that would greatly reduce the costs of testing Android applications for compatibility with many different types of devices, hence lowering the barrier for small companies and individual developers to produce high quality Android applications that work across devices with different Android OS versions, screen sizes, and hardware components.

6.2 Future work

The results of the work carried out during this master's thesis project should provide a good base for future developments. Since the prototype system is in working condition, improvements could be made step by step, while keeping the system functional and usable. Some of the areas for future work are described in the following subsections.

6.2.1 Security

One of the most important areas to improve in order to make the CBT platform closer to production ready is to improve overall system security. It should be improved in order to:

- Ensure secure communication between the web service and clients. Theoretically, employing the HTTPS protocol would be enough to ensure that messages could not be easily intercepted or read (there of course should be mutual authentication of the web service and the clients).
- Improve the authorization mechanism in order to make ensure that only authenticated users can modify only their own data.
- Protect devices from malicious applications. Since developers are sharing their own devices that might contain private data, devices must be protected from malicious applications and harmful test scripts. This is quite a complicated problem to solve and various techniques could be employed. One of these techniques would be to run applications and tests on dedicated devices, either on physical or virtualized emulators and analyze the code's behavior in order to detect malicious access to data. Only after passing initial check, tests would be uploaded and executed on user devices. Additionally, a mechanism could be

introduced that enables developers to set permissions[75] that applications being downloaded to their devices must conform to. The *CBT web service* could analyze these settings and exclude applications that do not conform to the permission set by the user sharing his device. Another or complimentary approach could be to back-up all the information on device and reset it to factory defaults before making it available for testing. Back-up time would depend on device: assuming that device has 32GB of internal memory rated as Class 10 by SD[76] standard (around 10MB/s write and 30MB/s read speed) and a USB[77] 2.0 connection it would take around 20min for back-up and 3 times that (1 hour) to restore the device since write speeds are usually much slower. However, assuming wider spread of USB 3.0 and faster UHS-I type memory (~ 100MB/s) introduction in Android devices, back-up time could be decreased to 6 minutes which might be a perfectly acceptable time if device is being connected to CBT system for a longer period of time, for example, during the night.

- Protect the *CBT web service* and its file storage. In the case of unauthorized actions, both applications and tests stored on the system could be stolen or modified. This might result in loss of users intellectual property, since ideas and technology used in their applications could be revealed without their consent. Similar event would also destroy the reputation of such service and would greatly decrease the number of potential users. Encrypting files stored within the system would probably be a good idea, along with putting more effort into intrusion detection & prevention.

6.2.2 Missing features

In order to improve the prototype to a level that could compete with existing similar (and more expensive) services, CBT system must provide at least functionality that is considered to be essential functionality based on analysis of potential competitors, this functionality includes:

- Add a feature to be able to download screenshots made by the testing framework. There are no known technical obstacles for implementing this feature. Moreover, existing infrastructure already supports this functionality.
- Add feature to download device logs produced during test execution.
- Add feature to collect device resource state statistics during execution. This functionality is not presently available in all similar services, but would definitely improve the application's usability.
- Add support for additional testing frameworks. Currently only *uiautomator* based tests are supported and this is a relatively new framework which has not achieved high popularity yet. The Android instrumentation testing framework seems to be most widely supported framework, as there are more frameworks built on top of it (such as Robotium and Calabash).
- Add the possibility to integrate CI tools such as Jenkins. Since tests provide the best value when executed automatically, it is highly desirable to eliminate additional human effort for verification of an application. It is important to provide a means of automatically uploading tests and applications, as well as initiating test execution. This is already possible as the *CBT web service* can be operated using a Restful API. However, it can be quite complicated to implement the client side for automating tasks, therefore a special command line client or plug-ins for Jenkins could be implemented in order to take the automation to next level.

6.2.3 Coverage

The prototype implementation (of the CBT web server) was only tested on Linux OS and only with Android devices. Theoretically this code could be used on other desktop OSs that are supported by the Android SDK this way increase the number of potential users. However, specific components were design only for executing tests on Android devices. This makes this system Android specific, but in general, this kind of system could be adopted for other mobile platforms

such as Microsoft's Windows Phone, Blackberry, Apple's iPhone and iPad. Since all of these systems suffer from varying degrees of fragmentation , developers would benefit from being able to run tests on devices of their friends rather than being forced to buy expensive hardware or use cloud services.

6.2.4 Business models

There are various business models that could be used in order pay the bills for the development, maintenance, and hosting of the CBT platform's services. These models need to be carefully thought through in order to choose one that fits this system best and would keep the service running and expanding.

One of the options would be to keep the platform free for basic use, but charge for value added features, such as increased security or private deployments for corporate clients. Organizations could leverage existing devices of their employees and for privacy reasons might want to use a privately deployed system. This business model could be further explored in order to find what other requirements an organization might put on the system.

Another potential business model would be to use a point system where users get points for sharing their devises and points are expended for executing tests on the device provided by other users. Potentially, people might be interested in giving access to their devices not only to their friends, but to anyone who is prepared to pay for it.

6.2.5 Avoiding the need for attaching the Android device to a computer

Ideally, it would be desirable that the *CBT agent* was running inside the Android device, thus avoiding the need for a computer running the android debug bridge in between the *CBT web service* and the *CBT agent*. However, based on available information, Android does not provide any means of installing applications without manual intervention other than via the android debug bridge. This is a show stopper for automatically downloading applications and executing tests via some other means. Perhaps in the future this possibility will open-up. However until then, there must always be some way to perform manual testing or testing applications that are already installed on device. These possibilities could be explored in order to widen the scope of the platform's use cases.

References

[1] 'APK (file format)', *Wikipedia, the free encyclopedia*, 30-January-2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=APK_(file_format)&oldid=535636731. [Accessed: 03-February-2013].

[2] 'Android home page', *Android Home page*. [Online]. Available: http://www.android.com/. [Accessed: 12-January-2013].

[3] 'Gartner Mobile Sales report Q3 2012'. [Online]. Available: http://www.gartner.com/newsroom/id/2237315. [Accessed: 29-January-2012].

[4] 'Linux official web page'. [Online]. Available: http://www.linux.org/. [Accessed: 29-January-2012].

[5] Y. D. Liang, *Introduction to JAVA programming: comprehensive version*. Boston: Prentice Hall, 2011, ISBN: 9780132130806 0132130807.

[6] B. W. Kernighan and D. M. Ritchie, *The C programming language / ANSI C Version*. Englewood Cliffs, N.J.: Prentice Hall, 1988, ISBN: 0131103628 : PAP 9780131103627 : PAP.

[7] S. Prata, *C++ primer plus*. Upper Saddle River, NJ: Addison-Wesley, 2012, ISBN: 9780321776402 0321776402.

[8] 'Android NDK | Android Developers'. [Online]. Available: http://developer.android.com/tools/sdk/ndk/index.html. [Accessed: 29-January-2013].

[9] 'Android - supporting multiple screens'. [Online]. Available: http://developer.android.com/guide/practices/screens_support.html. [Accessed: 29-January-2012].

[10] 'Software testing', *Wikipedia, the free encyclopedia*, 01-February-2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Software_testing&oldid=535990302. [Accessed: 02-February-2013].

[11] D. Torres Milano, *Android application testing guide build intensively tested and bug free Android applications*. Birmingham, U.K.: Packt Pub., 2011, ISBN: 9781849513517 1849513511 1849513503 9781849513500, Available atz http://site.ebrary.com/id/10482255.

[12] 'Welcome to Jenkins CI! | Jenkins CI'. [Online]. Available: http://jenkins-ci.org/. [Accessed: 04-February-2013].

[13] 'How Do Top Android Developer QA Test Their Apps ?' [Online]. Available: http://techcrunch.com/2012/06/02/android-qa-testing-quality-assurance/. [Accessed: 27-January-2013].

[14] 'Automated Testing Tool for Android - Testdroid'. [Online]. Available: http://testdroid.com/. [Accessed: 03-February-2013].

[15] P. M. Duvall, *Continuous integration: improving software quality and reducing risk*. Upper Saddle River, NJ: Addison-Wesley, 2007, ISBN: 9780321336385 0321336380.

[16] 'Google'. [Online]. Available: http://www.google.com/about/company/. [Accessed: 11-June-2013].

[17] 'Open Handset Alliance'. [Online]. Available: http://www.openhandsetalliance.com/. [Accessed: 03-February-2013].

[18] 'Exploring the SDK | Android Developers'. [Online]. Available: http://developer.android.com/sdk/exploring.html. [Accessed: 03-February-2013].

[19] 'Android API: Activity'. [Online]. Available: http://developer.android.com/reference/android/app/Activity.html. [Accessed: 02-February-2013].

[20] 'Android API: Intent'. [Online]. Available: http://developer.android.com/reference/android/content/Intent.html. [Accessed: 03-February-2013].

[21] 'GUI Architectures'. [Online]. Available: http://martinfowler.com/eaaDev/uiArchs.html#ModelViewController. [Accessed: 30-June-

2013].
[22] ‘Android API: Bundle’. [Online]. Available: http://developer.android.com/reference/android/os/Bundle.html. [Accessed: 03-February-2013].
[23] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, ‘Testing android apps through symbolic execution’, *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, p. 1, November 2012, DOI:10.1145/2382756.2382798.
[24] ‘Android Instrumentation class’. [Online]. Available: http://developer.android.com/reference/android/app/Instrumentation.html. [Accessed: 11-June-2013].
[25] ‘JUnit’. [Online]. Available: http://junit.org/.
[26] ‘ADT Plugin | Android Developers’. [Online]. Available: http://developer.android.com/tools/sdk/eclipse-adt.html. [Accessed: 03-February-2013].
[27] ‘ActivityUnitTestCase | Android Developers’. [Online]. Available: http://developer.android.com/reference/android/test/ActivityUnitTestCase.html. [Accessed: 04-February-2013].
[28] ‘ActivityInstrumentationTestCase2 | Android Developers’. [Online]. Available: http://developer.android.com/reference/android/test/ActivityInstrumentationTestCase2.html. [Accessed: 04-February-2013].
[29] R. Patton, *Software Testing*, 2nd ed. Sams Publishing, 2005, ISBN: 0672327988.
[30] ‘Android UI testing’. [Online]. Available: http://developer.android.com/tools/testing/testing_ui.html. [Accessed: 02-February-2013].
[31] ‘Android API: UiAutomatorTestCase’. [Online]. Available: http://developer.android.com/tools/help/uiautomator/UiAutomatorTestCase.html. [Accessed: 02-February-2013].
[32] ‘Robotium - Android test framework’. [Online]. Available: http://code.google.com/p/robotium/. [Accessed: 12-January-2013].
[33] ‘Maven - software project management tool’. [Online]. Available: http://maven.apache.org/. [Accessed: 11-June-2013].
[34] ‘Ant - build manager’. [Online]. Available: http://ant.apache.org/. [Accessed: 11-June-2013].
[35] ‘Gradle - Build Automation Evolved’. [Online]. Available: http://www.gradle.org/. [Accessed: 25-May-2013].
[36] ‘The Jython Project’. [Online]. Available: http://www.jython.org/. [Accessed: 03-February-2013].
[37] ‘ARM architecture’, *Wikipedia, the free encyclopedia*, 01-February-2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=ARM_architecture&oldid=535993691. [Accessed: 03-February-2013].
[38] ‘Code coverage’, *Wikipedia, the free encyclopedia*, 09-May-2013. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Code_coverage&oldid=554368745. [Accessed: 26-May-2013].
[39] ‘Xamarin Test Cloud’. [Online]. Available: http://xamarin.com/test-cloud. [Accessed: 11-June-2013].
[40] ‘PerfectoMobile’. [Online]. Available: http://www.perfectomobile.com/. [Accessed: 12-January-2013].
[41] ‘KeynoteDeviceAnywhere’. [Online]. Available: http://www.keynotedeviceanywhere.com. [Accessed: 11-June-2013].
[42] ‘AppLover’. [Online]. Available: http://applover.me/. [Accessed: 11-June-2013].
[43] ‘uTest’. [Online]. Available: http://www.utest.com/. [Accessed: 12-January-2013].
[44] ‘CloudMonkey’. [Online]. Available: https://www.gorillalogic.com/cloudmonkey. [Accessed: 11-June-2013].
[45] ‘cisimple’. [Online]. Available: https://www.cisimple.com/. [Accessed: 04-February-2013].
[46] ‘Calabash-android’, *GitHub*. [Online]. Available: https://github.com/calabash/calabash-

android. [Accessed: 12-January-2013].
[47] ‘Tit for tat’. [Online]. Available: http://en.wikipedia.org/wiki/Tit_for_tat. [Accessed: 28-January-2012].
[48] ‘Representational state transfer’, *Wikipedia, the free encyclopedia*, 17-May-2013. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=555343371. [Accessed: 19-May-2013].
[49] ‘Guice dependency injection’. [Online]. Available: https://code.google.com/p/google-guice/. [Accessed: 11-June-2013].
[50] ‘Jetty - Servlet Engine and Http Server’. [Online]. Available: http://www.eclipse.org/jetty/. [Accessed: 18-May-2013].
[51] ‘Jersey official web page’. [Online]. Available: https://jersey.java.net/. [Accessed: 11-June-2013].
[52] ‘MD5’, *Wikipedia, the free encyclopedia*, 21-May-2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=MD5&oldid=556112902. [Accessed: 25-May-2013].
[53] ‘HTTP cookie’, *Wikipedia, the free encyclopedia*, 24-May-2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=HTTP_cookie&oldid=555260389. [Accessed: 25-May-2013].
[54] ‘HTML’, *Wikipedia, the free encyclopedia*, 25-May-2013. [Online]. Available: https://en.wikipedia.org/w/index.php?title=HTML&oldid=556707822. [Accessed: 25-May-2013].
[55] ‘Hypertext Transfer Protocol’, *Wikipedia, the free encyclopedia*, 24-May-2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Hypertext_Transfer_Protocol&oldid=555926626. [Accessed: 25-May-2013].
[56] ‘JSON official web page’. [Online]. Available: http://www.json.org/. [Accessed: 11-June-2013].
[57] ‘JSON’, *Wikipedia, the free encyclopedia*, 25-May-2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=JSON&oldid=556703766. [Accessed: 25-May-2013].
[58] ‘JAR (file format)’, *Wikipedia, the free encyclopedia*, 31-January-2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=JAR_(file_format)&oldid=533322895. [Accessed: 03-February-2013].
[59] ‘Java bytecode’, *Wikipedia, the free encyclopedia*, 28-April-2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Java_bytecode&oldid=546979611. [Accessed: 20-May-2013].
[60] ‘DEX format’. [Online]. Available: http://source.android.com/tech/dalvik/dex-format.html. [Accessed: 11-June-2013].
[61] ‘DexFile in Android API’. [Online]. Available: http://developer.android.com/reference/dalvik/system/DexFile.html. [Accessed: 11-June-2013].
[62] ‘Smali library for Android’s dec format’. [Online]. Available: https://code.google.com/p/smali/. [Accessed: 11-June-2013].
[63] ‘Data access layer’, *Wikipedia, the free encyclopedia*, 24-February-2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Data_access_layer&oldid=538071771. [Accessed: 25-May-2013].
[64] ‘Core J2EE Patterns - Data Access Object’. [Online]. Available: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html. [Accessed: 11-June-2013].
[65] ‘Hibernate - ralational persinstance’. [Online]. Available: http://www.hibernate.org/. [Accessed: 11-June-2013].

[66] ‘JDBC’. [Online]. Available: http://www.oracle.com/technetwork/java/overview-141217.html. [Accessed: 11-June-2013].
[67] ‘JOOQ’. [Online]. Available: http://www.jooq.org/. [Accessed: 11-June-2013].
[68] ‘SQL’, *Wikipedia, the free encyclopedia*, 24-May-2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=SQL&oldid=556630776. [Accessed: 25-May-2013].
[69] ‘MySQL official web page’. [Online]. Available: http://www.mysql.com/. [Accessed: 11-June-2013].
[70] ‘Database engine rankings’. [Online]. Available: http://db-engines.com/en/ranking. [Accessed: 11-June-2013].
[71] ‘Regular expression’, *Wikipedia, the free encyclopedia*, 22-May-2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Regular_expression&oldid=556241919. [Accessed: 23-May-2013].
[72] A. Freier, P. Kocher, and P. Karlton, ‘The SSL Protocol Version 3.0’. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-ssl-version3-00. [Accessed: 11-June-2013].
[73] ‘Android dashboards’. [Online]. Available: http://developer.android.com/about/dashboards/index.html. [Accessed: 11-June-2013].
[74] ‘Ubuntu official web page’. [Online]. Available: http://www.ubuntu.com/.
[75] ‘Android Permissions’. [Online]. Available: http://developer.android.com/guide/topics/security/permissions.html. [Accessed: 11-June-2013].
[76] ‘Home - SD Association’. [Online]. Available: https://www.sdcard.org/home/. [Accessed: 30-June-2013].
[77] ‘USB.org - Welcome’. [Online]. Available: http://www.usb.org/home. [Accessed: 30-June-2013].

Appendix

Source code

Source code of all applications and components that were developed during this work is available in GitHub. Links to repositories are provided in Table 4.

Table 4 Links to source code

Repository description	Link to Git repository
CBT web service	https://github.com/noiseoverip/cbt-ws
Data access objects used by *CBT web service* and *CBT agent*	https://github.com/noiseoverip/cbt-ws-dao
CBT agent	https://github.com/noiseoverip/cbt-client
Test (*uiautomator*) for demo Android application used for testing CBT system	https://github.com/noiseoverip/cbt-example-app-uiautomator
Demo Android application	https://github.com/noiseoverip/cbt-example-app

Test speed-up calculations

Amdahl's law speed-up formula:

$T(n) = T(1)\left(B + \frac{1-B}{n}\right)$; n – number of parallel executions, B- fraction of sequential process.

Since we have durations of execution with 1 device which is roughly 30 seconds (Figure 5-1), 2 devices – 17 seconds and 4 devices - 10 seconds, we can calculate the sequential process fraction. We can derive formula of fraction $B = \frac{n*T(n)}{T(1)*(n-1)} - \frac{1}{n-1}$. Than we can calculate that B~0.13 in

case of 2 parallel execution and B=0.11 in case of 4 parallel executions. These numbers are quite similar so we can average them. Since we do not care about precision B=0.12 was taken as an average value and used in further calculations. Having this, we can say that it takes 3.6 seconds (12% of 30 seconds) to fetch and install required files and it takes 6.6 (30 seconds – sequential part and divide by 4 since it was executing 4 tests) seconds to execute one test. Now we can calculate estimated B for different number of test classes:

Table 5 Estimations of sequential process fraction

Number of test classes	Sequential process fraction
4	0.12
8	0.06383
12	0.043478
16	0.032967
20	0.026549

Since we have fractions of sequential process estimates, Amdahl's law can be used to estimate the potential speed-up.

TRITA-ICT-EX-2013:133

www.kth.se