Distributed GUI Testing of iOS Applications with HadoopUnit

by

Krissada Dechokul

Bachelor of Science
in Information and Communication Technology
Mahidol University
2009

A thesis
submitted to Florida Institute of Technology
in partial fulfillment of the requirements for the degree of

Master of Science
in
Software Engineering

Melbourne, Florida
May 2014
We the undersigned committee hereby approve the attached thesis,

"Distributed GUI Testing of iOS Applications with HadoopUnit"

by

Krissada Dechokul

Scott Tilley, Ph.D.
Professor, Computer Sciences
Thesis Advisor

Marco Carvalho, Ph.D.
Associate Professor, Computer Sciences
Committee Member

Chris Sonnenberg, Ph.D.
Assistant Professor, College of Business
Committee Member

William Shoaff, Ph.D.
Associate Professor, Computer Sciences
Department Head
Abstract

Title:
“Distributed GUI Testing of iOS Applications with HadoopUnit”

Author
Krissada Dechokul

Principle Advisor:
Scott Tilley, Ph.D.

Smartphones have become an important medium of communication for millions of people. They have come to expect high-quality mobile applications, which have increased the importance of software testing in modern software development. While software testing can ensure the quality of a system under test (SUT), it has suffered from lengthy execution time, especially for system-level GUI testing. In general, system-level GUI testing can be time-consuming because it requires walking through every user interface element and every possible variation that users could experience through the GUI of the application. In addition, this type of testing exercises the SUT from the outside of the applications, meaning that it has to constantly wait for the drawing of UI components, delays in transitions between different views and animations throughout the test execution.

Currently with UI Automation, a popular testing framework from Apple for iOS applications, test execution of iOS GUI test suites can take an hour or more to execute if the GUI of the application is complicated. This amount of time is significant if the testing needs to be performed frequently throughout the software development life cycle. More time means less frequent testing, which in turn reduces software quality.
In this thesis, the test process is accelerated by migrating the current test execution environment to a distributed execution environment called HadoopUnit. HadoopUnit has been adopted to perform GUI testing of iOS applications on a small-scale Hadoop cluster. Experimental results have shown that distributed test execution with HadoopUnit can significantly outperform the test execution on a single machine, even if the size of the cluster used for the execution is as small as two nodes. This means that the work of this thesis could be adopted without a huge investment in IT resources, making HadoopUnit a cost-effective solution for reducing lengthy test execution time of system-level GUI testing.
# Table of Contents

List of Figures ........................................................................................................ viii
List of Tables ............................................................................................................... x
Acknowledgment ....................................................................................................... xi
Dedication .................................................................................................................... xii

## CHAPTER 1 ............................................................................................................. 1

1.1 The Problem ......................................................................................................... 2
1.2 The Approach ........................................................................................................ 3
1.3 Research Objectives ............................................................................................. 5
1.4 Related Work ......................................................................................................... 5
  1.4.1 GUI Testing Tools ......................................................................................... 5
  1.4.2 Distributed Testing Platform ....................................................................... 8
1.5 Thesis Outline ....................................................................................................... 10

## CHAPTER 2 ............................................................................................................. 12

2.1 Introduction .......................................................................................................... 12
2.2 Software Testing .................................................................................................. 12
  2.2.1 Regression Testing ...................................................................................... 14
  2.2.2 GUI Testing ............................................................................................... 15
2.3 UI Automation ...................................................................................................... 16
  2.3.1 UI Automation Script ................................................................................. 17
  2.3.2 Command-Line Workflow with UI Automation ........................................ 19
  2.3.3 Rake .......................................................................................................... 22
  2.3.4 Virtualization ............................................................................................. 24
2.4 Hadoop ................................................................................................................ 25
  2.4.1 HDFS .......................................................................................................... 25
  2.4.2 MapReduce ................................................................................................ 26
2.5 HadoopUnit .......................................................................................................... 28
  2.5.1 Architecture of HadoopUnit ..................................................................... 28
  2.5.2 Approaches of HadoopUnit to Software Testing .................................... 30
2.6 Summary .............................................................................................................. 32

## CHAPTER 3 ............................................................................................................. 34

3.1 Introduction .......................................................................................................... 34
3.2 UI Automation Test Suite .................................................................................... 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1</td>
<td>Test Case Design</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Test Case Analysis</td>
</tr>
<tr>
<td>3.3</td>
<td>HadoopUnit Preparation</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Operational Environment</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Test Results</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Design</td>
</tr>
<tr>
<td>3.4</td>
<td>Usage</td>
</tr>
<tr>
<td>3.4.1</td>
<td>The Test Case List</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Rake</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Test Execution with HadoopUnit</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>4.2</td>
<td>Threats to Validity</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Test Suites</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Hadoop Optimization</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Network</td>
</tr>
<tr>
<td>4.3</td>
<td>Experimentation</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Experiment I</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Experiment II</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Experiment III</td>
</tr>
<tr>
<td>4.4</td>
<td>Discussion</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
</tr>
<tr>
<td>5.1</td>
<td>Research Summary</td>
</tr>
<tr>
<td>5.2</td>
<td>Contributions</td>
</tr>
<tr>
<td>5.3</td>
<td>Results</td>
</tr>
<tr>
<td>5.4</td>
<td>Future Work</td>
</tr>
<tr>
<td>5.5</td>
<td>Concluding Remarks</td>
</tr>
</tbody>
</table>

References | 74 |
Appendix A................................................................................................................. 84
Appendix B.................................................................................................................. 92
List of Figures

Figure 1.1 Overall Architecture of Third Party Testing Tool ........................................ 7
Figure 2.1 Software Engineering Life Cycle - Waterfall Model ..................................... 12
Figure 2.2 Level of Testing ............................................................................................ 13
Figure 2.3 Iterative Development Model ....................................................................... 15
Figure 2.4 Sample UI Automation Test Case ................................................................. 18
Figure 2.5 Sample Xcodebuild Command ..................................................................... 19
Figure 2.6 Sample of an Instruments Command ............................................................ 21
Figure 2.7 Sample of a Rake Task for Executing an Instruments Testing ....................... 23
Figure 2.8 Sample of a Command to invoke a defined Rake Task ................................. 23
Figure 2.9 Sample of a Command to invoke a Rake Task with --rakefile ................. 23
Figure 2.10 HDFS Architecture ..................................................................................... 26
Figure 2.11 Overview of How MapReduce works .......................................................... 27
Figure 2.12 Overall Architecture of HadoopUnit .......................................................... 30
Figure 3.1 Architecture of HadoopUnit for GUI testing of iOS applications ............... 42
Figure 3.2 Sample of a Test Case List .......................................................................... 43
Figure 3.3 Sample of Command to Transfer a File to the HDFS ................................. 47
Figure 3.4 The Hadoop Command to Initiate Test Execution with HadoopUnit ... 47
Figure 3.5 The Hadoop Command to Download Files from the HDFS .................... 49
Figure 4.1 A Sample Screenshot of the System Under Test ........................................ 54
Figure 4.2 Sample of a Test Case ................................................................................ 55
Figure 4.3 Code for Executing Test Cases Sequentially with a Rake Task ............... 57
Figure 4.4 Sequential Execution Time on a Single Machine ....................................... 58
Figure 4.5 Concurrent Execution Time on a 2-Nodes Cluster .................................. 60
Figure 4.6 Concurrent Execution Time on a 4-Node Cluster .................................. 62
Figure 4.7 Execution Time Comparison of the 3 Experiments .................................. 64
Figure 4.8 Total Test Execution Time Approximation Equation .......................... 65
Figure 4.9 Ideal Case for Test Execution Time Approximation Equation ............ 66
List of Tables

Table 1 Sequential Execution Time on a Single Machine (in seconds) ........................ 57
Table 2 Concurrent Execution Time on a 2-Node Cluster (in seconds) ..................... 59
Table 3 Concurrent Execution Time on a 4-Node Cluster (in seconds) ..................... 61
Table 4 Performance Comparisons of the 3 Experiments ...................................... 63
Table 5 Performance Factors over Sequential Execution ................................. 64
Acknowledgment

To Dr. Scott Tilley my thesis advisor who persuaded me to work on a thesis and provided me guidance from the very beginning, which resulted in the work of this thesis. Thank you.
Dedication

To my parents who always support me over all these years, I couldn’t even have been here studying in the first place without them.
CHAPTER 1

Introduction

Mobile applications recently gain popularity in software development industries. Apple already marked a historical 50 billion apps downloads record on their platform and reported a total of $10 billion sales in 2013 [1], [2]. Gartner predicted a total of 102 billion mobile apps download in 2013, and the amount will be doubled in 2016 [3]. One of the main keys driving the records and predictions is the growth of the smartphones and tablets. There is a report revealed that the sale of smartphones already surpassed that of personal computers [4], while the sale of personal computers keeps declining overtime indicating a shift in customer behavior on personal computers market [5], [6], [7]. Tablets were also being forecasted to surpass the personal computer market in the fourth quarter of 2013 [8]. The growth of smartphones and tablets raises the importance of mobile applications, as they have become a primary method that people interact with business worldwide [9].

Users also have higher expectations of mobile applications’ quality. One way that helps ensure quality of mobile applications is to through software testing. Software testing is one of the main activities in software engineering life cycle, which are requirements, design, construction, testing, and maintenance. Testing is a very important process that helps assess the functionality and ensure the quality of software before being released by identifying defects and problems [10]. System-level GUI testing, which is a primary focus of this thesis, is especially important in mobile application testing because it tests an application after all components has been integrated to ensure that the application behaves as intended [11].
On the iOS platform, GUI testing can be done through the use of Apple’s UI Automation [12]. It was first introduced with the introduction of iOS 4 as part of Instruments, a suite of profiling tools for tracing application’s behaviors and performance related issues at runtime. UI Automation allow users to write UI Automation scripts to exercise through the user interface of an iOS application by simulating user’s gestures such as tapping, pinching, swiping, or flicking. With UI Automation, testers do not have to repeatedly perform the same set of actions on the GUI of the application every time code got integrated.

1.1 The Problem

Apple once made a very interesting statement when they first introduced UI Automation during the WWDC 2010 conference, “Automating User Interface Testing with Instruments or How to Find Bugs While You Sleep” [13]. This statement implied that test execution of a GUI test suite could take a considerable amount of time to finish. There are two main causes that make test execution to take considerable amount of time; the number of test cases in a test suite, and the execution time of each test case.

A good GUI test suite covers all functionality of an application accessible through its user interfaces and exercises every user interface element being presented to users on screen. The user interfaces of a mobile application as well as the variations through which users can interact with the application can grow large in number and the amount of test cases to cover in a test suite increase accordingly [14]. Moreover, the test suite continues to grow over the course of software development process, especially in agile methodology where more test cases are being added each iteration. The testing also needs to account for different versions of operating systems, device orientations, and form factors of mobile devices such as smartphones and tablets increasing the amount of test cases to cover in a test suite.
The execution speed of this type of testing also can be very slow and time-consuming. This is because testing through the GUI is proceeded at the time closer to end-user speed rather than the processor speed [15]. This type of testing asserts the behaviors of an application from the outside like the users would actually see. This means that the testing tool needs to constantly wait for the drawing of user interface elements, animations, or network delay constantly throughout the test execution. Combining the slow test execution speed with the total number of test cases would result in the test process being even slower.

Test execution of a GUI test suite could take an hour or more to execute every test case in a GUI test suite, which already is a significant amount of time, and if the user interface of an application is complicated, the time required for a test execution to complete might go beyond the times and resources available for a project. This limits how many times testing could be performed and also limits things a software development team could get done during the course of the project.

To solve this problem presented by a large test suite is more complicated than simply having multiple separated processes to execute the more test scripts on a single machine. One reason is because only one test script could be run at a time. Moreover, only one instance of iOS simulator could be run at a time. [12]. The complexity of this problem is investigated and explained throughout this thesis.

1.2 The Approach

There are many solutions proposed to solve the test execution time problem. One solution to mitigate such problems is to reduce the number of test cases to run. There are many researches [16], [17] proposed approaches to reduce the number of test cases that testers have to run instead of executing the whole regression test suite, for instance, selective re-test techniques that tried to execute only a subset of test cases in a test suite. There are several selective re-test techniques such as
test case selection [18], test case prioritization [19], and test suite reduction [20]. However, these kinds of techniques are complicated, as decisions have to be made on which test cases not to run. With such decisions, some of the regression bugs might have been missed with the dropped test cases. Re-test all approach is more straightforward that avoids such errors [17]. A discussion on regression test selection is provided in the Chapter 2 and is not a focus for this thesis.

Another approach to mitigate the problem is to move to a faster execution environment for testing, a distributed execution environment. A pre-existing framework, HadoopUnit [21], that was designed to solve this problem is studied and being adopted to perform GUI testing throughout the experimentation in this thesis. In the previous study, HadoopUnit provided an infrastructure for executing JUnit test cases concurrently in a private cluster or in the cloud to speed up the test execution process by providing a cluster of machines that each could run JUnit test cases. The result from the previous studies have shown that using a medium-scale cluster of 150 nodes, the test execution runtime of a Hadoop’s JUnit test suite with 230 test cases could be impressively improved for up to approximately 55 times faster [22] by executing a single test per node. Because not everyone has access to large amount of IT resources even with a medium cluster as shown from the previous studies, there is also a study of HadoopUnit on a smaller-scale of cluster of 4 nodes and the result have shown that such impressive performance improvement is not always guarantee on such a small-scale cluster. Some modification had been made to HadoopUnit to further fine-tune its performance by letting each node to execute more than one test concurrently [23].

The structure of the HadoopUnit platform and the problems that it solved on reducing the execution time of JUnit test suite have made it a perfect candidate as a platform for system-level GUI testing of iOS applications. In order to migrate testing to a distributed execution environment, several artifacts involved in the testing process of the current test execution environment are identified. The
important artifacts in system-level GUI testing of iOS applications include a set of test cases that composes a test suite, UI Automation that is the testing tool, a trace document which is the test results of each test execution, and the operational test environment. These artifacts are studied for feasibility in migrating to a new environment and HadoopUnit has been customized accordingly to fit their needs.

1.3 Research Objectives

The primary objective of this thesis is to reduce the times it takes to perform system-level GUI testing on iOS platform by considering migrating from a traditional sequential execution environment over a single machine to HadoopUnit, a distributed execution environment over a cluster of machines, that executes a test suite in a concurrent manner.

The secondary objective is to prove the effectiveness of a test execution after migrating to a distributed execution environment over a small-scale cluster, as not every software group has access to large amount of resources. Software groups of any size can get benefits from this work without a huge investment on the infrastructure.

1.4 Related Work

There are several other projects that could also be used to serve on the same objective of this thesis, to reduce the test execution cost of a large regression test suite. The following section provides brief explanation of some of them.

1.4.1 GUI Testing Tools

The primary reason that UI Automation was selected as a testing tool for GUI testing tool in this thesis was mainly because it works without requiring users to install any software or customize their Xcode projects. More layers of services would add complexity to the work of this thesis and might get in the way of finding the answers to the questions set forth by this thesis.
There are several third parties testing tools available and could be used in the context of this thesis as well. The main reason for a software development team to consider the third-party testing tools is that the team was already familiar with the tools or language used by the tools. Although they rely on different technologies and implementations, they share similar concepts on how these testing tools work in general.

There are three common components in these testing tools as shown in Figure 1.1.

- **An Agent** – A framework or a small server that will be installed and lived inside the system under test. The installation needs to be performed on an iOS project manually through Xcode IDE.

- **A Script or A Feature** – The main set of test scripts for the system under test. Programming languages used for each tool will be different based on the underlying technologies used by the testing tool, for example, Objective-C [24], JavaScript [25], CoffeeScript [26], or Cucumber [27].

- **An IDE** – This component is a remote server or a separate piece of software running on a tester machine that helps recording the interactions, executing the scripts, and issuing commands that interact with the system under test through the agent component installed inside the application.
GUITAR [28] is a framework for performing automated GUI testing. GUITAR framework has several notable features including automated test case generation and automated execution of test case. It consists of four main components, Ripper for generating a structure model of the GUI of the system under test, Graph Converter for converting the structure model into a graph, the Test Case Generator for generating test cases based on the graph, and Replayer for automated executing the generated test cases. GUITAR has a plugin-based architecture meaning that a specific plugin needed to be developed to support different platform. GUITAR supports multiple platforms such as Java, Web, and Android. iPhone GUITAR [29] is still under-development to support GUI testing of iOS application. One weakness of GUITAR is that it does not support any manual test cases development (both scripted and captured) and lead toward automating the whole process instead.

**MonkeyTalk** [30] is a free, open-sourced object-based recording tool from Gollila Logic, Inc. that looks into the app’s code to understand the interaction being performed. This approach is supposed to be more accurate that image-based recording that detects pixel screen and also result in a more readable and maintainable script when comparing to those script that rely on image recognition technology. MonkeyTalk has its own MonkeyTalk IDE to record or construct a
new script and playback the test script. There are a lot of actions and gestures to select from including wait time specified in milliseconds. MonkeyTalk uses its own proprietary MonkeyTalk Scripts for constructing test cases while still supporting JavaScript. MonkeyTalk also support command-line test execution through either Ant Runner or Java Runner.

**Frank** [31] is another tool for automated iOS testing by Pete Hodgson. It was described by the tool developers as Selenium for native iOS apps. This tool use a combination of Cucumber and JSON commands that are sent to a server running inside the native application. The benefit of using natural language of Cucumber is its readability for non-technical stakeholders in the project. Frank includes a tool called Symbiote that let users to explore the system under test via a web browser. Its focus, however, was specifically for running on simulator making it hard to run the test suite on the actual device.

**KIF** [32] is also another tool for automated iOS testing created by Square, the company behind the Square Credit Card Reader for iOS and Android devices. Like MonkeyTalk and Frank, KIF allows users to automate iOS GUI testing. However, the tool was based on Objective-C built on top of OCUnit instead of the scripts language like JavaScript, Cucumber, or MonkeyTalk Script. But this fact leads to a better integration of the test suite and more powerful that it could call out to almost anything inside the system under test. KIF is more favorable for the teams that consist mainly of developers because programming knowledge is required to construct the test cases.

### 1.4.2 Distributed Testing Platform

There are several other projects that implement a distributed testing platform or framework for software testing and share a similar focus of reducing the test execution time of a large regression test suite. Some of them are briefly discussed in this subsection.
CloudTesting [33] is a framework that provides a solution to parallelize the test execution of a test suite over a distributed infrastructure. It has a similar goal to HadoopUnit, which is to improve the performance of the test execution process by distributing the execution of a test suite over a distributed infrastructure such as cloud or a private cluster. However, CloudTesting took a different approach by integrating the framework with development tools used by software developers or testers with the goal to encapsulate all the underlying work from them. The framework was divided into many different components.

- **Configuration** – This component takes care of all the underlying configuration of the project such as the paths, hosts in the cluster, and load balancing parameters between them as well as other settings necessary to set up the test environment.

- **Reflection** – This component takes care of the test case extraction process that extracts all the test cases to be executed under the distributed infrastructure. The result from this component will be used by the next component, the Distribution.

- **Distribution** – This component sits in the middle between the framework and the distributed infrastructure and takes care of distributing the execution of test suites and test results back and forth between them.

- **Connection** – This component provides communications with the distributed infrastructure. This component is used by the Distribution component to manage each test and its test result back to the client.

- **Log** – This component records every event occurred during the process.
• Main – This component encapsulates other components to make the testing process transparent to the users.

All of these components are packaged into a form of a separated plugin for the IDE used by software developers and testers. The current implementation only supports the Eclipse IDE and the Amazon Web Services infrastructure. Most of the responsibility that these components has was already provided by the Hadoop and to support the other development IDE such as Xcode, another plugin for the IDE needs to be separately developed and so could not be used in the context of this thesis.

**GridUnit** [34], [35] is a grid-based test execution tool for distributing the execution of JUnit test cases to a computational grid environment. GridUnit was developed on top of OurGrid [36], peer-to-peer computational grids that provide access to computational execution power and run parallel application. GridUnit provides functionalities similar to HadoopUnit, that is, it provides automatic test case distribution, test load distribution, and test execution control. However, GridUnit require users to develop test runner to schedule jobs in which for HadoopUnit, the underlying Hadoop already handle this automatically. Moreover, in its current form, GridUnit supports test case executions of JUnit written in Java only, while HadoopUnit was designed to be able to execute test cases independent of programming language.

### 1.5 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 describes background information about software testing, GUI testing with UI Automation, and HadoopUnit. Chapter 3 discusses UI Automation and HadoopUnit. Several issues involving with the design of test cases to utilize the distributed nature of HadoopUnit as well as the test case design and analysis to deal with interface changing more effectively and to improve the maintainability of the GUI test suite.
This chapter also explains rationales behind the implementation of HadoopUnit as well as its usages to perform GUI testing of iOS applications with HadoopUnit. Chapter 4 gives details of the experiments that were performed to determine the value of migrating a traditional execution environment to a distributed execution environment using HadoopUnit. Finally, Chapter 5 discusses the research summary, the results and contributions, and the possible future work of this research. This thesis also contain an appendix and reference sections to describe some topics in more detail as well as how to set up a HadoopUnit execution environment.
CHAPTER 2

Background

2.1 Introduction

The following sections provide some background information on topics related to this work. Information presented lay a foundation and area of study as well as the underlying reason that leads to the focus of this thesis. Those topics include software testing, UI Automation, and HadoopUnit.

2.2 Software Testing

Software testing is a very important process in software development process. It involves a technical investigation and evaluation of a software product to find information about its quality [37]. It is one of the main activities in the classic software engineering life cycle like the waterfall model [38] (Figure 2.1).

![Figure 2.1 Software Engineering Life Cycle - Waterfall Model](image-url)
Software testing may occur at any different level throughout a software development life cycle. According to the ISTQB, there are 4 levels of testing including component testing (or unit testing), integration testing, system testing, and acceptance testing [10] (Figure 2.2).

![Figure 2.2 Level of Testing](image)

Component testing (also known as unit testing) is a level of testing that runs at the lowest level down to individual function or method of a particular software product. There are many different types of tools that help testing at this level, for example, a unit test framework like JUnit for testing a Java application, CppUnit for testing a C++ application, OCUnit for testing an Objective-C application, or even some simple debugging tools could be used to test at this level [39]. Integration testing studies how two or more components work together by inspect the interfaces between different components or between different parts of a system under test. System testing tests the behavior of a system or a software product by concerning them as a complete system after integrating every component together in an attempt to demonstrate whether it meets its objectives. Acceptance testing is a level of testing that test a system or a software product with respect to the users to determine whether or not it should be accepted.
Software testing not only raises the level of confidence about a software product and ensures the quality of the product before it’s being released to the users or customers, it can also be used to ensure that software works correctly after being migrated. Alessandro and Filippo had illustrated an importance of software testing in software migration that it could be used to find potential divergences between the system under migration and the newly migrated system by performing regression testing [40]. They emphasized the importance of creating a system-level test suite used to test the newly migrated system on every iteration of migration to confirm the correctness of the system and the equivalence with the previous version of the system and that it did not introduce any regression. System-level testing helps reassure that changes in detailed implementation of the underlying components such as code refactoring or even internal architectural changes will not affect the overall functionality of the system.

2.2.1 Regression Testing

IEEE defines regression testing as a process of rerunning the test of a system to ensure that any modification of the system does not introduce any unintended side effects [41], [42]. It involves executing those test cases that previously have been executed to confirm that any modifications of the software or the environment have not broken any part of the application. Regression testing can be performed at any of the four levels of testing.

Regression testing is generally performed whenever a new version of an application is produced or a new version of environment likes an operating system upgrade is released. Performing regression testing helps reassure that a new release of an application did not introduce any form of regression. For instance, a fix or modification might causes any other working area elsewhere to break, or causes a previous bug fix to fail [43].
Regression testing is particularly important in iterative software development life cycles where a software project is broken down into several iterations [10] (Figure 2.3).

![Iterative Development Model](image)

In iterative life cycles, newly introduced functionality will need testing and all existing functionality will need regression testing, meaning more testing is required on each subsequent increment, which unfortunately leads to larger regression test suite due to the large amount of test cases being added up over time. With large regression test suites, performing regression testing can become costly and it might not always feasible to test everything in such test suites due to its excessively long execution time. As with this thesis, there are many ongoing researches to try to solve this problem by reducing such costs associated with regression testing.

### 2.2.2 GUI Testing

According to Ayman *et. al.*, GUI testing is a type of testing that validates the visual properties of GUI elements and the functionalities accessible through them [44]. GUI testing is performed on an application to verify whether it functions and behaves correctly, whether it does what it was supposed to do, and whether it provides good user experience to users [45].
The GUI is a very important part of an application as it acts as a bridge that connects together the application and its users. Unlike unit testing that focuses on the component level, an individual unit inside of an application, GUI testing focuses on the system-level, the user interface from the outside of an application, just like the users would see. GUI testing involves walking through every user interface element of an application and interaction that users can take through an application to accomplish some particular tasks. This means there can be a lot of test cases to cover in order to test every possible variation that users could experience. Combining with the fact that this type of testing can be time-consuming because every interaction has to pass through the GUI of the application making the execution of GUI testing to take considerable amount of time to run through a whole test suite [15].

2.3 UI Automation

Manual GUI testing of an application involves manually launching and walking through every feature that an application provide through its user interface displayed on screen, one by one, until it covers all the possible variations, repetitively. This can be cumbersome and error-prone if the user interface of an application get complicated enough. There will be mistakes and some specific paths or variations that users could take would be left untested. Nothing will replace manual-testing activities by skilled testers, however, these repetitive tasks can be automated using available automated testing tools [11].

Apple provides for free the Xcode IDE, a powerful suite of tools for developing iOS and OS X application. On the iOS platform, GUI Testing can be done through the use of UI Automation. UI Automation was built on top of JavaScript to perform automated user interface testing of an iOS application. It was first introduced in iOS 4 as one of many Instruments, a suite of profiling tools for tracing application’s behaviors and performance related issue at runtime. By
writing test scripts, UI Automation be could used to exercise through the user interface of an iOS application as well as simulating user’s gesture such as tapping, pinching, swiping, or flicking. It could also handle unexpected alerts, simulating multitasking, handling orientation changes and set location of the device to a specific coordinate. Instruments creates a document called the trace document and records all the behaviors of the system under test when performing these interaction into this document so that it could be inspected by the testers for misbehaviors [12].

2.3.1 UI Automation Script

Apple represents all user interface elements of an application through the Accessibility framework. UI Automation communicates with a system under test using assistive technologies such as VoiceOver that reads aloud all the user interface elements and gives visually impaired users a clue on how to navigate an application. Anything that the Accessibility framework can see is accessible by UI Automation [11].

User interface elements displayed on a screen of an iOS application are accessible through the JavaScript API provided by UI Automation. Although the language and the syntax are the same, there are some differences between the JavaScript library used by UI Automation framework and the JavaScript library used by typical web browsers. For example, there is no concept of document object model (DOM) in UI Automation framework and there are some additional API provided to facilitate in building up a test suite like #import directive or convenient methods to filter array of elements.

Instruments also provide tools that help recording and playing back interaction with an application, while also generating a line of automation script that perform such actions automatically. This feature greatly helps testers building their test cases faster. However, it doesn’t record timing of the recorded action, though delays could be easily added manually if timing is desired.
Being built on top of JavaScript, UI Automation gives testers flexibility in writing test cases to test their applications. The test script could be composed of a collection of statements to assert the behavior of an application under test or could be grouped together into a simple JavaScript function and get invoked like a normal JavaScript function. An example of a simple test case can be seen in Figure 2.4. The test case asserts that a table cell was actually removed from the table after simulating various actions and gestures.

```javascript
var target = UIATarget.localTarget();
var app = target.frontMostApp();
var window = app.mainWindow();

function testcase() {
  UIALogger.logStart("Simple test case");

  var.navigationBar = app.navigationBar();
  var editButton = navigationBar.leftButton();
  editButton.tap();

  // Grab a table cell with the name "Test"
  var cells = window.tableViews()["InfoList"].cells();
  var testCell = cells["Test"];
  var deleteSwitch = testCell.switches()[0];
  deleteSwitch.tap();
  var deleteButton = testCell.buttons()[0];
  deleteButton.tap();

  if (testCell.isValid()) UIALogger.logFail("Failed");
  else UIALogger.logPass("Passed");
}

testcase();

Figure 2.4 Sample UI Automation Test Case

UI Automation provides ways to report whether a test case passed or failed by using simple logging functions as seen previously in the sample test case. There are two main types of logging functions, logging with Test Status and logging With Severity Levels. Logging with Test Status functions are those functions that indicates that a test has completed successfully, unsuccessfull, or terminated abnormally. Examples of the functions of this type are logStart, logPass, logFail, or logIssue. Logging with Severity Levels function allows test to write debugging message with a specific level of severity. Examples of the function of this type are logMessage, logError, logWarning, and logDebug.
Instruments also automatically grabs a screenshot of the current state of the system under test whenever a test case failed so testers could see what the screen looks like and figured out what went wrong when it failed. A function to capture the screenshot could be invoked manually at any point through the test case. UI Automation records log messages and stores these screenshots in a trace document so that the testers can later inspect the document for misbehaviors.

Although there is no built-in support for assertions in UI Automation, UI Automation scripts could be easily extended by importing an external library like TuneUp JS that provides a collection of assertion functions to be used in our test script or they could be written to provide extra functionality [46]. The written test script can be used to test both iPhone and iPad application on a simulator and also can be used to test on an actual device that connected to Xcode [12].

2.3.2 Command-Line Workflow with UI Automation

Xcode IDE and Instruments provide integrated user interface for all activities start from coding to testing. However, in order to automate the testing process, the test has to be executed through the command-line interface. There are two important commands involving with GUI testing of iOS applications and are used through out this thesis, xcodebuild and instruments.

The **xcodebuild** is a command to build an xcode project through the command-line terminal. A typical structure of this command is as followed. The backslash at the end of each line is just to break the command into multiple lines for the readability.

```
xcodebuild \
  -project PROJECT_NAME.xcodeproj \ 
  -scheme PROJECT_SCHEME_NAME \ 
  -configuration Release \ 
  -sdk iphonesimulator \ 
  CONFIGURATION_BUILD_DIR=/tmp/PROJECT_DIRECTORY \ 
  TARGETED_DEVICE_FAMILY=1 \ 
  build
```

*Figure 2.5 Sample Xcodebuild Command*
The **-project** option is to specify the project file of the Xcode project.

The **-scheme** option is to specify the scheme for an Xcode to build a project. A scheme is a collection of configurations, build setting, and targets that Xcode uses when building or testing an application. There can be multiple schemes in an Xcode project but only one can be active at a time and could be specified here with this option.

The **-configuration** option is to tell the compiler which build configuration to use. Xcode’s Profile action builds an app for the Release configuration, so for testing, Release is used here.

The **CONFIGURATION_BUILD_DIR** is an Xcode configuration setting to specify which directory the compiler should put all the build-related files which is the application under test into. This directory will also be the directory used by the instruments command to perform testing.

The **TARGETED_DEVICE_FAMILY** is another Xcode configuration setting to specify which device family to target. The value of 1 means the targeted device family is iPhone or iPod touch while the value of 2 means the targeted device family is iPad. The value of 1,2 means the app is universal.

The last **build** option is just to instruct Xcode to build this project.

After the Xcode project has been build with xcodebuild command, the application is now ready for testing.

The instruments is a command to start the Instruments tool through the command-line terminal. A typical structure of the instruments command is as followed. The backslash at the end of each line is just to break the command into multiple lines for the readability.
The **-t** option is to tell the Instruments which trace document template to use for this test execution. A trace-document tells the Instruments to start a right profiling tool for the task, which, in this case, is the UI Automation tool. In the example, the trace-document template is TEMPLATE.tracetemplate and is stored under automation directory.

The **-D** option is to tell Instruments the directory to which it can put all the trace documents into. Instruments needs a directory to store all the trace information into. That directory needs to be created beforehand otherwise the tool won’t start. In the example, the directory is automation_results and the trace document is named TRACE. The trace document is very important as Instruments store all information of the testing into the file including all log messages and screenshots of the application under test that will give clues to testers what’s wrong with the application when it failed the test. When there is a problem with test execution, this directory is where testers can get all the test-related information.

The following line is the app-bundle of the system under test that was previously built using the xcodebuild command. In the example, the app-bundle is named SUT.app and is stored in /tmp/PROJECT_DIRECTORY.

The **-e UIARESULTSPATH** is to specify an environment variable to tell UI Automation the location to write an XML copy of the automation trace log into.

The **-e UIASCRIPT** is to specify an environment variable to tell UI Automation the path to the test script to be used for testing.

The basic structure of these commands indicate what information testers would need for each test execution and that HadoopUnit needs to account for.
2.3.3 **Rake**

Rake is a ruby build tool with capabilities similar to what Make has [47]. Rake works out of the box on Mac OS X without requiring any additional dependency to be installed. With Rake, all complexity of the building an Xcode project with xcodebuild and running the test with instruments commands can be hidden in a very simple rake task. There also are some useful features of Rake that could be added into the process. Those features are:

- Creating or removing a directory
- Storing common paths into variables
- Processing or constructing string
- Sequentially executing a command after another
- Conditionally executing commands based on given parameters
- Killing process after finish executing a task

The following fragment of code is a sample of a Rake task defined for executing a test with Instruments. The code starts by defining a number of common paths of an Xcode project. Then the code defines a Rake task named test, which accepts a UI Automation script’s file name before starts building and performing a test respectively. After finish testing, the code kills the simulator process.
BUILD_DIR = "/tmp/PROJECT_DIRECTORY"
APP_BUNDLE = "#{BUILD_DIR}/AUT.app"
AUTOMATION_TEMPLATE = "automation/TEMPLATE.tracetemplate"
RESULTS_PATH = "automation_results"
OUTPUT_TRACE_DOCUMENT = "#{RESULTS_PATH}/Trace"
PROJECT_DIR = "PROJECT_NAME.xcodeproj"
BUILD_SCHEME = "PROJECT_SCHEME_NAME"

mkdir_p RESULTS_PATH
desc "Run a test given a UI Automation script"
task :test, :file do |t, args|
  file = args[:file]
  sh %{ xcodebuild \ 
    -project "#{PROJECT_DIR}" \ 
    -scheme "#{BUILD_SCHEME}" \ 
    -configuration Release \ 
    -sdk iphonesimulator \ 
    CONFIGURATION_BUILD_DIR="#{BUILD_DIR}" \ 
    TARGETED_DEVICE_FAMILY=1 \ 
    build }
  sh %{ instruments \ 
    -t "#{AUTOMATION_TEMPLATE}" \ 
    -D "#{OUTPUT_TRACE_DOCUMENT}" \ 
    "#{APP_BUNDLE}" \ 
    -e UIARESULTSPATH "#{RESULTS_PATH}" \ 
    -e UIASCRIPT file }
  puts "\nTest Passed"
  sh %{killall "iPhone Simulator" || true}
end

**Figure 2.7 Sample of a Rake Task for Executing an Instruments Testing**

A Rake task defined like in the example could be invoked by executing a following command.

```
rake test["automation/TEST_SUITE.js"]
```

**Figure 2.8 Sample of a Command to invoke a defined Rake Task**

A specific Rake file with a name different from the default one could also be specified by using --rakefile option as seen in

```
rake --rakefile RakefileSequential test["automation/TEST_SUITE.js"]
```

**Figure 2.9 Sample of a Command to invoke a Rake Task with --rakefile**

Rake is being used throughout this thesis for many testing scenarios, for instance, to sequentially execute a set of test cases on a single machine or to test a particular test case given a path to a UI Automation script in a form of a Rake task.
2.3.4 Virtualization

Virtualization is the process of emulating actual IT resources with virtual IT resources. Virtual machines created from the process act exactly like actual IT resources having their own guest operating systems, which are independent of the host operating system that created them. Virtual machines run through the virtualization software, which in turn run on a physical host, and become hardware independent because hardware resources have been emulated by the virtualization software and thus could be easily moved to another virtualization host, cloned, manipulated as needed [48].

There are some limitations on using the normal execution environment to execute UI Automation test cases. First of all, only one test script could be run at a time. All test cases in a test suite can only be run sequentially under this environment. It is also not possible to have multiple instances of the simulators running on a single machine and also not possible to have multiple instances of the debuggers to run the tests on multiple devices attached to the machine. This means that this execution process could not be speeded up simply by having multiple separated processes to execute test cases in parallel under a single machine, since this is not possible.

Many practitioners suggested the use of multiple virtual machines running simultaneously on a single machine to solve the problem that only one instance of UI Automation debugger and simulator can be run at a time. Having multiple virtual machines run on a single machine means that there are multiple debuggers to run the tests. With limited resources, virtualization technologies can increase the efficiency and better utilization of the available resources in a cost effective manner. Due to a strict licensing stated in software license agreement under the section 2 B (iii) [49] however, only two additional virtual machines of Mac OS X could be run on a single Mac machine. This restrictive license limits how far hardware resources could be utilized under this environment.
Although using virtualization technologies increase a number of debuggers and simulators to run the test, this solution still requires manual test distribution, meaning test cases needs to be manually selected and distributed to those virtual machines and manually be executed. A good system should not require its users to be responsible for test case distribution. This also could slow down the process if the size of the cluster becomes large enough [16]. Nevertheless, this work-around directs us toward a distributed execution environment where there can be multiple instances of debuggers running in a cluster of machines, while test cases also get automated distribution as a side benefit. Virtualization technologies are being used to set up such cluster in various configurations on the case studies of this thesis.

2.4 Hadoop

Hadoop is a collection of open source tools, libraries, and methodologies designed to run on commodity hardware or in the cloud served as a scalable platform for big data analysis [50]. Hadoop consists of two core components, the Hadoop Distributed File System (HDFS) and MapReduce. A set of machines running these two components is known as a Hadoop cluster. Individual machine in a Hadoop cluster is known as a node and a cluster can consist of just one node or as many as a thousand of these nodes.

2.4.1 HDFS

Data stored in a Hadoop cluster is stored in the Hadoop Distributed File System (HDFS). In HDFS, data files are split into blocks, each block usually has a size of 64MB or 128MB, significantly larger than a conventional file system that is usually has a size around 64KB. These blocks are distributed across many different machines in the Hadoop cluster. Hadoop was designed with an assumption that hardware will fail. So some of these blocks are also replicated throughout other different machines meaning that there are more than one machine storing the same copy of these blocks. In case of hardware failure, the same block of data could still
be extracted from the other nodes in the cluster. The node in the cluster that stores these blocks of data files is called DataNode. There is a single master node in the cluster called the NameNode that controls and keeps track of the locations of all these data blocks stored in different DataNode in the cluster as well as which blocks compose a file stored in the HDFS. There is also one or more Secondary NameNode that acts as a checkpoint for the NameNode. In the case where the NameNode failed, the NameNode could be restarted using a backup snapshot stored in the Secondary NameNode [51] (Figure 2.10).

![HDFS Architecture](image)

**Figure 2.10 HDFS Architecture**

### 2.4.2 MapReduce

Hadoop provides big data processing capability through the MapReduce programming model. The MapReduce programming model was built upon the concept that a computation is applied over a large number of records distributed all over the cluster to generate partial results that, in turn, are aggregated to produce the final solution to the users [52]. The MapReduce programming model hides underlying execution details from users and provides automatic parallelization and
distribution where developers can concentrate on writing data processing functions in a form of MapReduce jobs. A MapReduce job consists of two functions: a Map function and a Reduce function. A Map function processes a given split of data derived from a block of data in a form of key/value pairs and produces intermediary output, also as a set of key/value pairs. The shuffle and sort works in the background based on the key from the intermediary output and feed these output as an input to a Reduce function. A Reduce function aggregates the values of the processed intermediary output from the Map function based on the key part of the given key/value pairs and provides the ultimate results of the MapReduce job. Figure 2.11 illustrates the overview of how MapReduce works.

MapReduce jobs are controlled by a software process running on the master NameNode called the JobTracker. Clients submit a MapReduce job through the JobTracker, which the JobTracker then assigns Map and Reduce tasks to the other DataNodes that stored a block of data to be processed in the cluster. These DataNodes each runs a software process called the TaskTracker. The TaskTracker receives Map or Reduce tasks from the JobTracker and instantiates the given task on the node and report progress back to the JobTracker. In case of task failures or
in case when a particular task take unusually long, the tasks are restarted on the
other available node in the cluster automatically in the background by the
JobTracker. Under the speculative execution, if there is a free node available in the
cluster, a redundant task will be assigned to it and the result will be collected from
the node that finish executing the task first.

2.5 HadoopUnit

HadoopUnit is a distributed execution environment tailored specifically for
executing JUnit test cases concurrently on the cloud proposed by Tauhida Parveen,
Scott Tilley, Nigel Daley, and Pedro Morales [21]. HadoopUnit was originally
developed on the need to mitigate the lengthy test execution of the Hadoop
production code by using the Hadoop platform itself because the method of test
execution at that time took very long times to run and could not provide timely
feedbacks to the developers. Hadoop was created using Java and its test cases are in
the form of JUnit, hence HadoopUnit was originally created specifically for
executing JUnit test cases [22].

2.5.1 Architecture of HadoopUnit

HadoopUnit was built upon Hadoop with an adapted concept from Hadoop
that a test suite composes of a large set of test cases. Test cases are distributed all
over machines in a cluster and the test execution is considered computing or
processing upon them. The execution of test cases generates partial test results,
which are part of the test suite. The final result is aggregated from those partial test
results from the execution of test cases that are parts of a complete test suite.

HadoopUnit provides the infrastructure for the JUnit test cases distribution
and execution over a cluster of machines. HadoopUnit hides all the complexity of
the cluster management and maintenance, job scheduling, resource allocation, and
fault tolerance from the users, thus letting them focus on what is important to them,
the testing and the test result.
There are three core components in HadoopUnit.

- **Test Case Extraction** – This component of HadoopUnit is responsible for gathering all the test cases to be executed from the test suite and generating a test case list in the form of line-delimited string with each line composed of test case name and test execution command pair, a key/value pair format suitable to be processed by Hadoop. The original version of HadoopUnit use Ant, a Java build tool, as a component to provide this functionality.

- **The Map Function** – This component of HadoopUnit is a Hadoop Map function responsible for receiving and processing given test case name/test execution command pairs by using the test case name as a key and execute the corresponding command as a separate process. The Map function produce intermediary result in a form of a test case name/test result pair, a key/value pair format suitable to be processed by the next component, the Reduce function.

- **The Reduce Function** – This component of HadoopUnit is a Hadoop Reduce function responsible for collecting the intermediary test results from the Map function, aggregating into one report, and placing it in the HDFS where testers could extract the report to their machine.

Figure 2.12 provides an overall architecture of HadoopUnit.
Figure 2.12 Overall Architecture of HadoopUnit

The process begins with testers creating a HadoopUnit project and uploading the test suite to HadoopUnit where all the production code (a.k.a. the system under test), test cases for testing the production code, and dependent libraries were already in place in the HadoopUnit cluster. The test suite then will be extracted and split into multiple sets of test cases that are being distributed to machines in the Hadoop cluster. Each node executes the given set of test cases and returns back the test result. All the test results of all the execution of test cases are then being merged and reported back to the tester the ultimate test result of the test suite.

2.5.2 Approaches of HadoopUnit to Software Testing

There are several approaches that HadoopUnit took right into the design to tackle several problems in software testing.
Retest-All Approach – HadoopUnit was designed to reduce the cost regression testing of a large regression test suite. While selective re-test techniques such as test case selection (test only what have been changed), test case prioritization (test the one with higher priority earlier), and test suite reduction (eliminate test cases from the test suite) could reduce the cost of regression testing, they also reduce the chances that regression bugs could be revealed. Some set of test cases that could potentially expose those bugs might be eliminated in the process [17], [22], [53]. Retest-All approach is the simplest approach that avoids such errors and was selected by HadoopUnit while HadoopUnit shifts the focus to making the testing more efficient.

Test Case Independence – To be able to be executed concurrently under a distributed environment such as HadoopUnit, a set of test cases must be independent from one another. They cannot be executed at the same time without violating one another if these test cases are dependent on each other. The tests should not affect each other and the order of execution should also not affect the final test result. This means that the test should be self-contained, isolated, and fully functional. Although having this quality in a set of test cases means that it can take longer to execute each test case, such isolated tests are valuable and provide higher quality feedback to the testers. This is because the failed test cases are not caused by any other dependent test cases in the test suites, which would not require testers to further interpret the meaning of each failed test case [22], [54].

Automated Test Case Distribution – Requiring manual test case distribution not only slows down the whole testing process but also
demands substantial amount of tasks to be done by the testers. After finishing all the execution, testers have to also gather the results from different machines they distributed the tests manually. This could be overwhelming especially if the size of cluster becomes large enough. HadoopUnit provides automated and transparent test case distribution, which is an ability that HadoopUnit inherited from Hadoop that it could be able to push input data into nodes in the cluster automatically as well as to collect results and report back when finish. Test Load Distribution also is handled automatically by the underlying Hadoop platform for load balancing of the cluster. This is because some of test cases might take longer than one another; a set of test cases is distributed to other available nodes in the cluster automatically [23]. This approach of HadoopUnit better utilizes the available resources and prevents bottlenecks in test executions.

These approaches and attributes of HadoopUnit make it a fine test execution environment for migrating a sequential test execution into and should speed up the testing process while, at the same time, should demands less works from testers.

### 2.6 Summary

This chapter provided background information on methodologies, technologies, and theories that have been combined in order to carry out the work of this thesis. The topics included:

**Software Testing** – consisted of a discussion on software development process, level of testing, regression testing, and GUI testing.

**UI Automation** – composed of an explanation of Xcode, Instruments, UI Automation, UI Automation script, its command-line workflow, and Rake, as well as their problems that need to be solved by the work of this thesis.
**Hadoop** – provided explanation of underlying technologies of Hadoop, the HDFS, and the MapReduce programming model.

**HadoopUnit** – provided discussion on the HadoopUnit framework, its underlying technology, architecture, as well as those approaches that HadoopUnit took to solve problems in software testing.

The next chapter provides explanations and procedures on how HadoopUnit could be customized to fit the needs to perform GUI testing of iOS applications.
CHAPTER 3

HadoopUnit and UI Automation

3.1 Introduction

In migrating from a traditional execution environment to a distributed execution environment like HadoopUnit, careful up-front planning needs to be taken, especially when planning to automate the testing process. Simply using a good set of tools does not guarantee the success of the project. The following sections provide information on test case design and analysis, HadoopUnit preparation, and the processes involved in using HadoopUnit as a platform for performing GUI testing of iOS applications with UI Automation.

3.2 UI Automation Test Suite

In order to migrate from a traditional GUI testing environment towards a distributed execution environment, the System Under Test (SUT) and its test suite are needed to be prepared so that an execution of different test cases could be performed independently in parallel under the new environment. In other words, the test cases need to be designed in such a way that they could be used in HadoopUnit environment. It is a recommended practice that a test suite be developed in conjunction with the application so that the test could be run frequently as the development process goes on [55]. It is worth mentioning the that an automated GUI testing project should be treated as a genuine programming project [56].

Once an application is released, it may be used or serviced for years. During the course of time, an application may inevitably need to be fixed (patches
or bug fixes), changed (improvement or enhancement changes, or environment changes like operating system upgrade e.g. the version of iOS), or extended (adding new functionality to the application) [10]. The tests, once written, must also be maintained along with the application. Using capture/replay tools helps testers getting up and run in building up a test suite quickly. In order for it to be maintainable, which is to be able to grow and endure changes, some considerations on the test case design is needed so that the test case will have such qualities. The test suite needs to evolve along with the software project [57].

Moreover, the UI of an application will change throughout the life cycle of software development, especially in agile methodologies where changes in requirements are welcome, as long as the changes satisfy the users. So test cases also need to be designed to be able to cope with this possibility and that the cost of such modification to the test case should be kept to the minimum, for instance, less amount of work required to update the test scripts [15], [57].

3.2.1 Test Case Design

Every feature of an application should be defined and written as a function within a test suite where test cases that need to test these features will need to test through these functions. In other word, instead of having test cases to contain scripts for tapping a particular button to achieve some particular tasks, a function for such tasks should be created to wrap the underlying interactions with the UI around inside the function. This practice hides the low-level implementation of how the feature could be accessed and thus when the UI of the feature changes, the function that access that feature is the only place that needs to be updated instead of every test case that tests the feature. It’s also a good idea to pull out common scripts and create utility functions that other test cases can use, so when more functionality is being added to the function like additional logging then all test cases that use this function gain the additional functionality automatically [58], [59].
Jonathan Penn suggested an interesting technique in designing UI Automation script for better maintenance by representing each screen of an application as a screen object with necessary methods to access GUI elements displayed on its screen [11]. The test cases that need to have access to the screens could then import the script containing the screen object and perform testing on GUI elements as necessary with the provided methods. This way, if the user interface of that screen got changed, only the script that representing the screen needs to be updated rather then every other place that needs to access that changed user interface thus reduce the damages from UI changes.

For HadoopUnit, while the dependency within the same test case is acceptable, different test cases in a test suite needs to be designed in a way that different test cases can be executed independently. Because the test case execution will be executed on different node on a cluster, such dependency can cause problems in test execution. This is also the requirement of the test cases to be executed under the HadoopUnit environment itself as mentioned in Chapter 2. The test cases should be independent from each other meaning the order of execution of test cases should not affect the test result. This characteristic is also called test case contamination: the test case $t_A$ should not contaminate the test case $t_B$ no matter which one got executed first [22], [16]. A test case should set up necessary data and other information required for testing without requiring other test case (self-contained). There is also another benefit by keeping test cases independent, a test case could be updated, improved, or refactored without affecting the other test cases [60]. And since each test case is independent of each other, in the case when a test case failed, testers know exactly which test case to look for rather than requiring the tester to analyze which other test cases also depend on the failed test case.
3.2.2 Test Case Analysis

Test case analysis might be necessary for an application with existing sets of UI Automation test suites in order to migrate to HadoopUnit environment. The existing test suites might contain all test cases in one UI Automation script file. Distributing test cases in such a test suite having only one master UI Automation script file is not possible because it requires a mechanism to selectively execute test cases inside the test suite. This is due to the limitation of UI Automation that a parameter could not be send to the UI Automation engine through the command line terminal to selectively execute a specific test case within a test suite. So such type of test suites needs test case analysis to break the test suite down into several files of test cases in order to prepare the test suites to run on this distributed environment like HadoopUnit.

As mentioned in the previous section, each UI Automation test script file representing each test case should be self-contained and needs to be independent of each other file. It can contain many tests but it should not require the other test file to work. If there are such dependencies between them, consider grouping them together into one script file.

The result of test case analysis is a set of UI Automation test script files that each represents a single test case. Each file should be executable with Instruments.

3.3 HadoopUnit Preparation

The original version of HadoopUnit was designed specifically to execute JUnit test cases of Java applications, hence, it operates under a different environment from that of iOS applications. Some customization has to be made to prepare HadoopUnit to be a suitable platform for GUI testing of iOS applications.
3.3.1 **Operational Environment**

In order to migrate a traditional GUI testing of iOS applications environment towards a distributed execution environment like HadoopUnit, a Hadoop cluster needs to be set up in a way that each node could be able to compile, build, and run iOS applications. This requires each node in the cluster to have the Xcode, Instruments, iOS simulator, iOS SDK, as well as the Command line tool for OS X installed on the node. These tools operate under the Mac OS X operating system, an Apple’s proprietary operating system based on UNIX system. This environmental requirement leads to several limitations and considerations that HadoopUnit has to accommodate.

- Only one test case could be run at a time during test execution. This limitation means that regardless the number of test cases in a test suite, they have to be executed sequentially [12]. This is not a problem for a distributed execution environment like HadoopUnit since there are many different nodes of machines available in the cluster to execute the test case concurrently. However, HadoopUnit needs to be configured to execute only one Map function at a time to accommodate this limitation, rather than the default value of two.

- Only one instance of iOS simulator could be run at a time. This is a limitation of the iOS simulator, a tool for running the application under test itself. This limitation is similar to the previous limitation, which means that another instance of iOS simulator could not be started up to run another set of tests on a single machine. In this case, the number of nodes available in the cluster would directly correlates to the number of simulator available to execute the test.

- Only two additional instances of OS X virtual machines could be run and they have to run on Apple’s hardware only [49]. Apple has restrictive license agreements on their operating system. This
requirement limits the size of the Hadoop cluster that could be constructed given available hardware resources. This restrictive requirement also makes it difficult to move this test execution environment to the cloud. Mac OS X isn’t available as an instance type on major cloud providers like Amazon’s EC2 or Microsoft’s Azure. This means that a Hadoop cluster of Mac OS X machines for testing iOS applications currently needs to be constructed and used internally only.

HadoopUnit as a platform for distributed test execution needs to account for these limitations of this operational environment so that it could be able to support GUI testing of iOS applications. More detailed information on customizing a Hadoop cluster can be found in the Appendix A.

3.3.2 Test Results

The Instruments tool, as mentioned in Chapter 2, generates test results not only as stream of log messages displayed on command-line terminal, but also records everything and puts them in a trace document. This trace document can be later opened with the Instruments GUI tool for tracking where and when the application under test went wrong, hence this document is very important and valuable for follow-up testing and inspections. The original version of Hadoop records test results only from that being output from the command-line output stream. Generating test reports from the test results only from this source make it difficult for testers to trace the behaviors of the application under test. HadoopUnit should retrieve this document after finish executing each test.

The original version of HadoopUnit, after the test case extraction process, receives a test case list in a form of line-delimited string of test case name and test execution command pairs. The location of a trace document is specified within the executing instruments command, as already mentioned in Chapter 2. However, to analyze the given execution command to extract the location of the trace document
from it would make HadoopUnit less generic. Therefore, an additional parameter is
required to tell HadoopUnit to retrieve additional documents on each execution
node that perform Map function and put them into the HDFS, so that testers can
conveniently download the documents into their machine for further investigations.

Since the cost of moving files in and out of HDFS is costly and time-
consuming because it relates to many I/O and network activities between many
different nodes in the cluster [61], this version of HadoopUnit has been designed in
such a way that only when a test case failed will trigger HadoopUnit to retrieve
such documents from the node. In other words, if the whole test suite passes, no
document is needed to be retrieved. This decision is also reasonable because the
purpose of regression testing is to find the test that failed, test cases that pass is not
an interest event in the process of regression testing. Given above considerations,
the test case list for HadoopUnit used in this thesis then is updated into the form of
a line-delimited string of test case name and test execution command pairs with the
location of the trace document to be retrieved by HadoopUnit.

3.3.3 Design

HadoopUnit after being customized to run iOS GUI testing still has three
core components like the original version of HadoopUnit, with slightly additional
functionality and some customized behaviors according to its limitations and
consideration.

- Test Case Extraction – Test case extraction is an important
  component of HadoopUnit in that it generates a test cases list in a
  form of line-delimited string of test case name and its execution
  commands. In this updated version, this list also includes the
  location of the output trace document to be retrieved by HadoopUnit
  after finish execution. For the sake of simplicity, however, Ant is
  not being used during the course of the experiment of this project.
Instead, test case extraction process was being performed manually using a simple script to construct the list beforehand. The execution command is in a form of a simple Rake task that wrapped around all the important classpaths, the location of the application bundle, the location to store the result trace document, as well as other scripts to take care of underlying work such as building the application under test before testing or closing the simulator after finish executing each test. The code of Rake task used in this experiment as well as the format of the list could be found in the Appendix B of this document.

- The Map Function – This component of HadoopUnit is still responsible for actually executing each test case given line-delimited string of test case names, test execution commands, and the location of trace-documents to be retrieved by the HadoopUnit. The given test case name is used as a unique key for each test execution. Test results will be merged by the Reduce function based on this key, so if there are lines with the same test case name, their test results will be merged together into one line of report. For this experiment, the test case name is merely a name of the UI Automation script file used in the execution. In the case when a test execution failed meaning that the application did not behave in an expected way as provided in the test script, the location of the trace document of that line will be used by HadoopUnit to retrieve and store the trace-document into the HDFS where testers can easily download the document for further investigation.

- The Reduce Function – This component of HadoopUnit simply merges all the test results into one report using keys, which is the test case name of the test execution, and put it in the HDFS.
Figure 3.1 illustrates the overall architecture of HadoopUnit after being customized for GUI testing of iOS applications. The process begins with clients uploading a test case list to be used by HadoopUnit to the HDFS. The test case list is in a form of line-delimited string of test case name and test execution commands with the location of the trace document to be gathered by HadoopUnit. Each node in the cluster was equipped with all necessary tools required for GUI testing of iOS applications, such as Xcode, Instruments, and iOS simulator, as well as an Xcode project of an application under test already in place on every node. The test case list is split and forms a set of map tasks to be distributed. After finishing executing the test, the output is being produced and stored in the HDFS as well as any additional documents specified with the test execution command. Clients then download the test report from the HDFS to their local machine. If some nodes failed, the map task given to the node will be automatically restarted on the other node in the cluster.

![Figure 3.1 Architecture of HadoopUnit for GUI testing of iOS applications](image-url)
3.4 Usage

The following explains how HadoopUnit is used as a platform for distributed GUI testing of iOS applications including discussion on the test case list, Rake, and how to perform various tasks with HadoopUnit.

3.4.1 The Test Case List

The list of test cases to be executed is the most important piece of information to be provided to HadoopUnit. It is a text file consisting of line-delimited text of test case names and their corresponding test execution command. It is provided to HadoopUnit in this form because it is the simplest form that could be easily handled by Hadoop. A sample of a test case list is demonstrated in Figure 3.2.

```plaintext
testcase_01.js @ rake test["automation/testcase_01.js"] @ test_results/
testcase_02.js @ rake test["automation/testcase_02.js"] @ test_results/
testcase_03.js @ rake test["automation/testcase_03.js"] @ test_results/
testcase_04.js @ rake test["automation/testcase_04.js"] @ test_results/
testcase_05.js @ rake test["automation/testcase_05.js"] @ test_results/
```

*Figure 3.2 Sample of a Test Case List*

Each line of text in a test case list consists of three components separated by the symbol '@'.

- Test Case Name – a test case name is used as a key to uniquely identify each test execution by the Map function and is used as a key to merge together its test result into one report by the Reduce function. There can be multiple executions of different commands for a single test case name meaning that there can be multiple lines in the list with the same test case name. In this case, after finish executing all the test cases, each test execution will be merged into one line by the Reduce function. When a test case fails, this test case name is used as a name of directory to store its trace document in the HDFS.
• Test Execution Command – a test execution command is a command to execute a test case. This test execution command will be executed as an external process by the Map function of HadoopUnit. This component can be in a form of a simple instruments command or any type of executable command. The execution commands demonstrated in Figure 3.2 are in a form of executable Rake task receiving a path to a UI Automation test script. More information about Rake will be explained in the next section.

• Location of Trace Document – This extra component is added to each line of text in this version of HadoopUnit to accommodate the way Instruments treat the test result of each test case execution. This component is the directory that Instruments uses to store a trace-document of this test execution. When a test case failed, HadoopUnit will use this piece of information to retrieve the trace-document of the test execution and copy the document into the HDFS.

The original version of HadoopUnit utilizes Ant, a build tool for Java project, to extract test cases to be executed from a test suite, as well as, other information needed to run the test cases such as necessary classpaths or libraries required to build and run a particular project. This test case list, however, can be constructed manually using a simple text editor or using a simple script to generate the list programmatically, no special tool is necessary in the process.

Once the test case list has been created, it needs to be uploaded to the HDFS. Its path in the HDFS will be used as one of a parameter in the command to initiate a Hadoop job. The command to initiate a Hadoop job will be explained in detail in the following sections.
3.4.2 Rake

As explained in the Chapter 2, Rake is a ruby build tool similar to Make. It makes the commands to execute each test case much simpler. Rake was chosen in the experiment because it is already available in Mac OS X without any further installation required by the users. In this thesis, a Rake task is created and used to wrap around long execution commands of instruments and provide additional functionality before and after executing each test case. An executable Rake task is used as an executable command in the test case list instead of a regular instruments command to run the test. In addition to executing shell scripts, functionality that are being utilized in this experiment are as followed.

- Define common paths in variables – The important paths necessary to execute UI Automation scripts are, for example, the location of the application bundle, the automation template, the location to store test results, as well as the location of Xcode project file to be able for a Rake task to compile and build the project. These paths can be stored in variables and used by commands within Rake tasks.

- Perform operations on directories – Some operations on directories are necessary in order to ensure a success in executing of a UI Automation test. For instance, the directory to store the test result needs to be created before executing each test case, a command to create a directory can be executed before executing an instruments command. Another directory operation before executing each test case is to clean up the test result of the previous test before each execution. This ensures that even in the case of corrupted trace-document of the previous test will not affect the current test execution.

- Receive additional parameters from the command-line terminal – Additional parameters can be sent to a Rake to customize the
behavior of its execution, for example, to execute a test given a path to a UI Automation script, to execute on a different device family, iPhone or iPad, or to execute the test using a different test scheme.

- Kill process – After finish executing each test case, the simulator process is killed to clear out all setting and configuration that might have left from the test so that it is ready for the next test case to be executed. This task is to ensure independency between different test cases.

There are two main Rake tasks created for two different scenarios for this experiment, a Rake task to sequentially execute test cases on a single machine or a baseline for this experiment and a Rake task that receive a path to a UI Automation script as a parameter to execute different test case in a test suite. The Rake file is placed along side with the Xcode project so that a Rake task could have access to the actual test script on each node in the cluster. The full source code of the Rake task for executing UI Automation test with explanation comments can be found in the Appendix B.

3.4.3 Test Execution with HadoopUnit

The following section provides an explanation of how HadoopUnit can be used to perform various tasks involved with performing distributed GUI testing of iOS applications with HadoopUnit.

1.1.1.1 Uploading an iOS project to each node in the cluster

Each node in the cluster is required to have a working iOS project in place within the node. This can be done using any kind of means that could transfers a set of files over the network such as File Transfer Protocol (FTP) or Server Message Block (SMB). In this experiment, each node is configured to enable File Sharing with Apple Filing Protocol (AFP), a built-in protocol that enables easy file sharing
in a Mac OS X network through Finder. This enables the project files to be easily distributed to each node over the network.

3.4.3.1 Uploading a test case list to the HDFS

The test case list previously created needs to be made available in the HDFS. It has to be uploaded only through the Hadoop file system command on the master NameNode, otherwise the test case list won’t be accessible by the Hadoop itself to perform its operations. An example of Hadoop file system command to transfer a file from a local machine to the HDFS can be seen in Figure 3.3.

```
hadoop fs -copyFromLocal <LOCAL_DIRECTORY>/testlist.txt <HDFS_DIRECTORY>/testlist.txt
```

**Figure 3.3 Sample of Command to Transfer a File to the HDFS**

3.4.3.2 Starting test executions with HadoopUnit

To initiate a Hadoop job to start a test execution process with HadoopUnit, after the iOS project of an application under test and the test case list have been properly uploaded, the Hadoop jar command as seen in Figure 3.4 could to be executed through the command-line terminal. The HadoopUnit.jar used by the command should be located on local machine used to execute the command.

```
hadoop jar <LOCAL_DIRECTORY>/HadoopUnit.jar TestDriver <HDFS_DIRECTORY>/testlist.txt <HDFS_DIRECTORY>/output
```

**Figure 3.4 The Hadoop Command to Initiate Test Execution with HadoopUnit**

There are four important arguments that need to be specified with this command.

- A JAR file that contain a MapReduce code. The HadoopUnit.jar is actually a MapReduce program packaged in a form of JAR file.
- The class name in the JAR file that contains a method to drive the MapReduce code, which in this case is TestDriver.
• The input file, which is the test case lists previously created and uploaded to the HDFS. The path of the input file is the path of the file located in the HDFS. This file will be accessed by Hadoop to perform tasks necessary to split and distribute test cases to different nodes in the cluster.

• The output directory. This is a directory where Hadoop store the results of execution in. This directory also is the place where HadoopUnit will store the trace documents of failed test cases in. The directory, however, needs to be empty in the HDFS or the Hadoop job can’t be started. So this directory might need to be cleaned up before starting a new job or entirely new directory might be selected for each execution.

After the command has been initiated, a Hadoop job is created. The input file will be accessed by Hadoop to perform tasks necessary to split the test cases list into what Hadoop calls input splits. Those input splits are then distributed to several nodes in the cluster automatically. The number and the size of each split are determined by the value of mapred.max.split.size pre-configured in the Hadoop configuration file. Detailed configurations of the HadoopUnit can be found in the Appendix A. After finish executing every test case, test results are merged by the Reduce function and put in the output directory specified in the command.

3.4.3.3 Gathering test results

After finish executing every test case, all the test results can be gathered from the HDFS using simple a Hadoop file system command as shown in Figure 3.5. The test results from Hadoop as well as a trace document of failure test cases will be downloaded from the directory in the HDFS to the specified directory on a local machine.
hadoop fs -copyToLocal <HDFS_DIRECTORY>/output <LOCAL_DIRECTORY>/output

Figure 3.5 The Hadoop Command to Download Files from the HDFS

3.4.3.4 Test results analysis

Test results from test executions are stored in a text file with a name given by Hadoop similar to the part-r-00000. The content of the file is a list of test case names and the result of each test whether its execution passed or failed. If a test case was labeled as failed, its corresponding trace document also be downloaded and stored in a directory named by its test case name. The results from these trace-documents need to be investigated and perform follow-up testing of the failure test cases as necessary by trying to execute individual test case separately on a local machine. There is also a chance of a false alarm where there was an error from the test execution resulting from the testing tool instead of from the misbehavior of an application under test. At the time of writing with Xcode 5.1, it has currently been marked as known issues in its release note. After the failure has been identified and fixed accordingly, an updated source code for the system under test will need to be re-uploaded as well as some updated sets of test cases (if there exist) to the each node in the Hadoop cluster so that another test execution can be performed again.

3.5 Summary

This chapter has provided necessary information to prepare a UI Automation test suite for migrating from a traditional execution environment to a distributed execution environment, limitations and consideration of the operational environment of GUI testing of iOS application, HadoopUnit customization and its usages. The topics included:

Test Case Design & Analysis – consisted of a discussion on how to design test cases in order to be able to cope with UI changes, as well as, how to design test cases in order to be executed on HadoopUnit.
**HadoopUnit Preparation and Customization** – consisted of a discussion on how the original version of HadoopUnit needs to be customized to meet the requirements and limitations of the operational environment of iOS GUI testing as well as its design to accommodate these issues.

**Usage** – provided discussion on how HadoopUnit could be used as a platform to perform various tasks for GUI testing of iOS applications.
CHAPTER 4

Experimentation

4.1 Introduction

The traditional test execution of GUI testing with UI Automation can only execute each test sequentially on a single machine. This requires considerable amount of time for the test execution process to complete if there are a lot of test cases in a test suite. This thesis considers HadoopUnit as a platform for GUI testing of iOS applications in an attempt to make the test execution faster. This thesis also considers the performance of HadoopUnit on a small-scale cluster to show that HadoopUnit could be adopted for system-level GUI testing without a huge investment in IT resources.

Toward these goals, several experiments were setup to run test cases on different scenarios. These experiments include running sets of test cases with a different number of test cases: 100, 250, 500, and 1,000 test cases sequentially on a single machine to provide a baseline for the experimentation. These numbers of test cases were chosen to simulate different size and complexity of the system under test. A large and complex application would contain a lot of test cases whereas a smaller one would contain a fairly smaller set of test cases. The same sets of test cases are then being run concurrently on a small 2-node and 4-node cluster with HadoopUnit and have their execution times compared with the baseline.
4.2 Threats to Validity

Some decisions regarding the experimentation were made as this thesis progressed and could threaten the validity of data and conclusions drawn from the experiment. Such concerns are discussed in this section.

4.2.1 Test Suites

For the experimentation being performed, test suites were built by duplicating a same test case to create a test suite with 100, 250, 500, and 1,000 test cases respectively. Each test case performs exactly the same set of tasks with slightly different data to indicate a test case name currently being performed on the node for monitoring its progress during a test execution.

The goal was to distribute loads equally on every node in the cluster to prove the effectiveness of each node being added to the cluster, each node was expected to get the same amount of work and to get fully utilized throughout the test execution. Additionally, the execution time could be directly compared after the test execution.

However, using this practice might not reflect real-life scenarios where each test case runs at a variable amount of times rather than the execution times that are closed to each other. There might be some test cases that run excessively longer than the other, whereas some might run in a shorter amount of time. There also is a type of test cases where networking or location services are involved. This could affect how Hadoop distribute the workloads while make it harder to conclude its effectiveness on test executions. Using a controlled set of test cases removes this factor that might influence the performance of the system.

4.2.2 Hadoop Optimization

Tuning the performance of Hadoop for a particular type of execution is subtle and is not a simple task. It involves with many different combinations of
parameters available for administrators to be configured [62]. A modification of one parameter could result in Hadoop to behave in a subtly different way that is hard to monitor. Some parameters are used as just a hint and it’s totally up to Hadoop to decide how it should behave given a particular situation. This is the trade-off of the simplification that Hadoop provided for developers to implement a distributed execution system [63], [61].

The focus of this thesis is to prove the usefulness of HadoopUnit in reducing the execution time of GUI testing of iOS applications under a small-scale cluster. By that reason, only a few critical parameters that could help accomplish this goal were selected, which are the mapred.tasktracker.map.tasks.maximum and the mapred.max.split.size. There are other Hadoop parameters that could have been considered to further optimize the performance of the test execution.

4.2.3 Network

The network used in this experiment to run a Hadoop cluster was a private network dedicated for used in the experiment only. Each node in the cluster was used solely for executing the tests. However, under a real production network used by many other individuals, the network traffic of the shared network might become a parameter that users who would like to adopt this system need to consider since Hadoop rely heavily on the communications between the nodes in the cluster.

4.3 Experimentation

A Hadoop cluster of two nodes and four nodes were set up for this experiment. Each node was running as a virtual machine of Mac OS X 10.8.5 Mountain Lion allocated with two processor cores and 3 GB of RAM on VMware Fusion 6.0.2. The system under test, which was developed solely by the author for this experiment, was a simple money management app where users could add, edit, or remove their daily money related activities. The application is shown in Figure 4.1.
The experiments were conducted using four different test suites consisting of 100 test cases, 250 test cases, 500 test cases, and 1,000 test cases running at separate times. Test cases in each test suite were the result of duplicating the same test cases performing the same set of tasks. Each test case consists of three tests as shown in Figure 4.2.
test("Creating a 'new' transaction", function() {  
  var main = MainScreen;
  var newtrans = NewTransactionScreen;

  main.gotoTransactionsScreen();
  main.addNewTransaction();

  newtrans.addAmount(123);
  newtrans.addNote('Test Case 1');

  newtrans.submitTransaction();
  main.gotoOverviewScreen();
});

test("Seeing a transaction's details", function() {  
  var main = MainScreen;
  var trans = TransactionsScreen;
  var detail = TransactionDetailScreen;

  main.gotoTransactionsScreen();
  trans.tapTransaction("Test Case 1, $ 123");
  trans.target().captureScreenWithName("Test Case 1");
  detail.goBack();
});

test("Removing a transaction", function() {  
  var main = MainScreen;
  var trans = TransactionsScreen;

  main.gotoTransactionsScreen();

  trans.removeTransaction("Test Case 1, $ 123");
  trans.assertNoTransaction("Test Case 1, $ 123");
});

Figure 4.2 Sample of a Test Case

The first test was to add a new expense transaction with a test case name as its description and a sample amount of expense. Then go back to the main list after finish adding a new transaction. The second test was to assert that a new transaction is actually being added to the list by tapping the added transaction in the list and check the detailed information of the transaction and go back to the main list. The last test was to remove the added transaction from the list and asserted that the transaction was successfully removed from the list. These three tasks compose a single test case in this experiment and being duplicated to create a test suite. Each test case create a transaction with different note, for example, a test case number two would add a note as ‘Test Case 2’ while a test case number 100 would add a
note as ‘Test Case 100’ and so on, for the purpose of monitoring as already mentioned.

Each test case used in this experiment takes, on average, 32 seconds to complete its execution of the three tasks as mentioned earlier. The execution includes the time it takes for the simulator to launch the application, perform transitions between different views, type text using virtual keyboard, perform animations during removing a transaction, and the times it takes for UI Automation itself to make an assertion.

The following sections provide detailed information on the experiments being performed to determine the effectiveness of the HadoopUnit as a platform for GUI testing of iOS applications as well as its performance gain from the migration.

4.3.1 **Experiment I**

The first experiment was conduct to provide a baseline for the rest of the experiment of the HadoopUnit experiment by executing each set of test suite sequentially on a single machine without using HadoopUnit. A Rake task was created to run each test case sequentially, one after another. The tasks being performed before and after each test execution, as mentioned in the Chapter 3, was the same with what being used in another Rake task for running under the HadoopUnit environment. The execution time was recorded and reported after finish executing every test case in the test suite. The code of this Rake task can be seen in Figure 4.3.
decs "Run a set of test cases sequentially"
task "test" do
  start_time = Time.now
  automate "automation/test_case_1.js"
  automate "automation/test_case_2.js"
  automate "automation/test_case_3.js"
  automate "automation/test_case_4.js"
  automate "automation/test_case_5.js"
  total = Time.now - start_time
  put "Finish executing 5 tests, took \{total\} seconds"
end

Figure 4.3 Code for Executing Test Cases Sequentially with a Rake Task

The experiments were run using 100, 250, 500, and 1,000 test cases respectively to simulate the different size and complexity of the application under test as mentioned earlier. The execution time was then determined whether it scaled linearly as the number of test cases grows large.

4.3.1.1 Results

Each set of experiments was conducted 10 times and the average execution time from the experiments was computed and recorded. The result’s execution times of the experiments are shown in Table 1 and Figure 4.4.

Table 1 Sequential Execution Time on a Single Machine (in seconds)

<table>
<thead>
<tr>
<th></th>
<th>100 Test Cases</th>
<th>250 Test Cases</th>
<th>500 Test Cases</th>
<th>1,000 Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>3,018.81</td>
<td>7,788.68</td>
<td>16,505.96</td>
<td>35,352.39</td>
</tr>
<tr>
<td>Runtime (s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

57
The result of 100 test cases took 3,018.81 seconds or 50 minutes and 18 seconds. The result of 250 test cases took 7,788.68 seconds or 2 hours, 9 minutes and 48 seconds. The result of 500 test cases took 16,505.96 seconds or 4 hours, 35 minutes, and 5 seconds. The result of 1,000 test cases took 35,352.39 seconds or 9 hours, 49 minutes, and 12 seconds.

4.3.1.2 Analysis

The results have shown that the execution time increased sequentially and approximately scaled linearly as the number of test cases in a test suite grows larger. It took excessively long time almost 10 hours to run a set of 1,000 test cases sequentially on a single machine. This clearly indicates the needs for a better execution environment for this kind of testing, which is the main purpose for this experimentation in the first place. These execution times represent the time that could be achieved when running a set of test cases only on a single machine and without the aid of HadoopUnit and will be used as a baseline for comparing with the execution time using HadoopUnit.
4.3.2 **Experiment II**

The second experiment was conducted with the same set of test suites and with an addition of HadoopUnit. A 2-Node cluster running HadoopUnit was set up to run GUI testing of iOS applications. This experiment was set up to reflect a scenario where a group of developers added a machine such as a Mac hardware capable of running two virtual nodes or simply added two physical machines to set up a Hadoop cluster with HadoopUnit. A set of test case lists was created with the number of test cases corresponding for each test, 100, 250, 500, and 1,000 lines in the list and supplied as input to HadoopUnit.

Each map task is a process that parses each line of input and executes each test case. Each map task runs on a node in the Hadoop cluster. Since there can be only one test case execute at a time in one node, the number of concurrent test execution is determined by the number of node capable of execute the test available in the cluster. For which node got which split of the test case list is determined solely by Hadoop.

4.3.2.1 **Results**

Each set of experiments was conducted 10 times and the average execution time from the experiments was computed and recorded. Execution times of the experiments were reported by the job’s progress that Hadoop constantly sends back to the command-line terminal during the course of execution. The result’s execution times of the experiments are shown in Table 2 and Figure 4.5.

<table>
<thead>
<tr>
<th>2-Node Concurrent Runtime (s)</th>
<th>100 Test Cases</th>
<th>250 Test Cases</th>
<th>500 Test Cases</th>
<th>1,000 Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,523</td>
<td>3,876</td>
<td>7,820</td>
<td>16,023</td>
<td></td>
</tr>
</tbody>
</table>
The result of 100 test cases took 1,523 seconds or 25 minutes and 23 seconds. The result of 250 test cases took 3,876 seconds or 1 hours, 4 minutes and 36 seconds. The result of 500 test cases took 7,820 seconds or 2 hours, 10 minutes, and 20 seconds. The result of 1,000 test cases took 16,023 seconds or 4 hours, 27 minutes, and 3 seconds. There were, however, an observable delay at the beginning of each execution for about 5 to 15 seconds before a new Map process be spawned on each node in the cluster. This delay is further discussed in the next section.

4.3.2.2 Analysis

The results have shown the reduction of execution time approximately in a half comparing with the sequential test case execution on a single machine. The execution time of a test suite with 100 test cases was reduced from 50 minutes and 18 seconds down to 25 minutes and 23 seconds. The execution time of a test suite with 250 test cases was reduced from 2 hours, 9 minutes and 48 seconds down to 1 hour, 4 minutes and 36 seconds. The execution time of a test suite with 500 test cases was reduced from 4 hours, 35 minutes, and 5 seconds down to 2 hours, 10 minutes, and 20 seconds. And finally, the execution time of a test suite with 1,000
test cases was reduced from 9 hours, 49 minutes, and 12 seconds down to 4 hours, 27 minutes, and 3 seconds. The result shows that a performance of 2x could be gained just by adding a Hadoop cluster running with HadoopUnit, even if the size of the cluster is as small as just two nodes.

4.3.3 **Experiment III**

The third experiment was conducted with the same set of test suites the first and the second experiment, with a 4-Node cluster running HadoopUnit set up to run GUI testing of iOS applications. This experiment could reflect a scenario where a group of developers could afford to set up a larger Hadoop cluster, with regarding to the license agreement that Apple has for their operating system, by adding two more virtual nodes running on another Mac hardware or simply adding two more physical machines to the cluster. The same set of test case lists used in the second experiment was also used in this experiment as input to the HadoopUnit.

4.3.3.1 **Results**

Each set of experiments was conducted 10 times and the average execution time from the experiments was computed and recorded with the same method as the second experiment. The result’s execution times of the experiments are shown in Table 3 and Figure 4.6.

<table>
<thead>
<tr>
<th>Table 3 Concurrent Execution Time on a 4-Node Cluster (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 Test Cases</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>4-Node Concurrent Runtime (s)</td>
</tr>
</tbody>
</table>

61
The result of 100 test cases took 866 seconds or 14 minutes and 26 seconds. The result of 250 test cases took 2,185 seconds or 36 minutes and 25 seconds. The result of 500 test cases took 4,332 seconds or 1 hour, 12 minutes, and 12 seconds. The result of 1,000 test cases took 8,452 seconds or 2 hours, 20 minutes, and 52 seconds. There were also observable delays at the beginning of each test execution as found in the second experiment and will be discussed in the next section.

4.3.3.2 Analysis

The result from the third experiment has shown an approximate 3.5x to 4x reduction in test execution time comparing with the sequential test case execution on a single machine, the value closed to the number of nodes used in the experiment. The execution time of a test suite with 100 test cases was reduced from 50 minute and 18 seconds down to 14 minutes and 26 seconds. The result of 250 test cases was reduced from 2 hours, 9 minutes, and 48 seconds down to 36 minutes and 25 seconds. The result of 500 test cases was reduced from 4 hours, 35 minutes, and 5 seconds down to 1 hour, 12 minutes, and 12 seconds. The result of 1,000 test cases was reduced from 9 hours, 49 minutes, and 12 seconds down to 2 hours, 20
minutes, and 52 seconds. The results from this experiment are promising and suggesting that a performance gain approximately by the number of nodes available in the cluster could be gained even from a small-scale implementation like in this experiment.

### 4.4 Discussion

The results from the second and the third experiment has shown that given a small scale Hadoop cluster of four nodes or even as small as two nodes, there can be a performance gain for up to the number of nodes in the clusters. The amount of times it took to execute every test case increased linearly as the number of test cases in a test suite getting bigger, the same behavior for all three experiments. The Table 4 and Figure 4.7 provide the comparison of execution time of these three experiments. Additionally, Table 5 provides the improvement factor comparing with the traditional sequential execution on a single machine.

<table>
<thead>
<tr>
<th></th>
<th>100 Test Cases</th>
<th>250 Test Cases</th>
<th>500 Test Cases</th>
<th>1,000 Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Runtime (s)</td>
<td>3,018.81</td>
<td>7,788.68</td>
<td>16,505.96</td>
<td>35,352.39</td>
</tr>
<tr>
<td>2-Node Concurrent Runtime (s)</td>
<td>1,523</td>
<td>3,876</td>
<td>7,820</td>
<td>16,023</td>
</tr>
<tr>
<td>4-Node Concurrent Runtime (s)</td>
<td>866</td>
<td>2,185</td>
<td>4,332</td>
<td>8,452</td>
</tr>
</tbody>
</table>
Figure 4.7 Execution Time Comparison of the 3 Experiments

Table 5 Performance Factors over Sequential Execution

<table>
<thead>
<tr>
<th></th>
<th>100 Test Cases</th>
<th>250 Test Cases</th>
<th>500 Test Cases</th>
<th>1,000 Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-Node</td>
<td>1.98</td>
<td>2.00</td>
<td>2.11</td>
<td>2.20</td>
</tr>
<tr>
<td>Concurrent Runtime (s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-Node</td>
<td>3.48</td>
<td>3.56</td>
<td>3.81</td>
<td>4.18</td>
</tr>
<tr>
<td>Concurrent Runtime (s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As compared to the sequential test execution on a single machine, distributed GUI testing with HadoopUnit showed an approximate 2x performance gained when executed with a 2-Node cluster and an approximate 3.5x to 4x performance gained when executed with a 4-Node cluster. The following summarizes the execution time reduction when executing GUI testing with HadoopUnit when comparing with traditional test execution on a single machine.
• 50 minutes of execution time could be reduced to about 25 minutes on a 2-Node cluster and about 14 minutes on a 4-Node cluster.

• 2 hours of execution time could be reduced to about 1 hour and 4 minutes on a 2-Node cluster and about 36 minutes on a 4-Node cluster.

• 4 hours of execution time could be reduced to about 2 hours and 9 minutes on a 2-Node cluster and about 1 hour and 12 minutes on a 4-Node cluster.

• An excessively long 9 hours and 49 minutes of execution time could be reduced to 4 hours and 27 minutes on a 2-Node cluster and to just 2 hours and 20 minutes when comparing with such a baseline on a single machine on a 4-Node cluster.

As seen from the experimental results, HadoopUnit can outperform the traditional test execution that can only execute each test case sequentially on a single machine in every case because more than one test cases could be executed in parallel, thus accelerate the process. Based on this experiments, the total test execution time could be roughly approximated with the following equation in Figure 4.8.

\[ T = \left( \frac{\sum_{\text{test}}}{\sum_{\text{node}}} \times \bar{t} \right) \]

Figure 4.8 Total Test Execution Time Approximation Equation

Where:

• \( T \) is the total execution time

• \( \sum_{\text{test}} \) is the total number of test cases in a test suite

• \( \sum_{\text{node}} \) is the total number of nodes in the cluster

• \( \bar{t} \) is an average execution time for each test case
Because each test case can only be executed sequentially on a single machine, in a normal case, the total amount of time it takes to execute the whole test suite can be computed from the total number of test divided by the total number of nodes available in the cluster for executing the test and multiply by the average execution time of a test case.

\[ T = t_{\text{longest}} \]

Figure 4.9 Ideal Case for Test Execution Time Approximation Equation

The ideal case for a distributed execution environment as shown in Figure 4.9 is the situation where the total number of test is equal to the total number of nodes. Then the time it takes for a whole test suite to finish is the execution time of the longest test case in the test suite. By that reason, it is then not recommended to have a single test case to be excessively longer than the other test cases in the test suite. Ideally they should be split evenly in term of time space eventually during the test case analysis process because the speed of executing a test suite will eventually be determined by the time it takes for the longest test case in a test suite to finish executing.

As seen from the experimental results, there were cases where the performance sometimes could not reach the number of nodes in the cluster as expected. This is because there is an overhead time introduced by Hadoop. Hadoop sacrifices some performance of its execution as an expense to provide underlying features such as automatic input splitting, task scheduling and distribution, and fault tolerance to free developers from worrying about these factors while developing MapReduce application [63]. The processes that contribute overheads to the execution performance with Hadoop are the transformation of the input file input key/value pairs, the statistical tasks throughout the execution for decisions on job scheduling, and the processing logic of the MapReduce programming model that is broken down into several phases such as map, shuffle and sort, and reduce
phase. All of these mechanisms of Hadoop contribute additional overheads to the overall execution time with Hadoop.

There were also noticeable delays at the beginning of each execution when testing with HadoopUnit. It was observed during the course of experiments that it took around 5 - 15 seconds at the beginning of each test execution before a new process for executing a test case be spawned on each node in the cluster. This is an overhead time introduced by the Hadoop to set up tasks. This is not the case for sequential test execution on a single machine where the test execution got started up right away when the Rake task to sequentially execute each test case got invoked from the command-line terminal.

It is stated in the Hadoop documentation that it could take some time for each Hadoop’s map task setup to complete. The document further suggests that the maps should take at least a minute to execute for large jobs [64], [65]. This statement means that the more map tasks, the more the setup times there will be for a single Hadoop job. Configuring the right number of map tasks for a Hadoop job can be subtle. It depends on many parameters and configuration of the Hadoop environment, such as the number of node available in the cluster and the size of the input split used by Hadoop. The number of map tasks for each job is determined by the size of the input file, in which in this experiment is the test case list being provided to HadoopUnit. The size of the input test case list comes from the length of each line of text in the test case list, which depends on the length of the test case name, the test execution command, and the trace document directory composing a single line-delimited string used by HadoopUnit.

Hadoop splits the given input file by determining the size of the HDFS block, so the split size is simply the size of the HDFS block [51]. However, the split size can also be controlled by modifying the value of mapred.max.split.size parameter. Normally the split size for each execution is set as default to the maximum value that could be represented by a Java long type and will have its
effects when the value is less than the HDFS block size. In this experiment, the parameter mapred.max.split.size was set to 1,000 bytes since the test case list is significantly smaller than the default block of 64 MB in HDFS. Setting this parameter to a value less than the block size will force the split size to be smaller than a block and increase the number of map tasks. The Rake task used in the test case list to execute the test is also significantly smaller than a typical instruments command to execute UI Automation making the size of the test case list file smaller than usual. Under a circumstance where the Rake task is not used, this parameter should be further adjusted by increasing the split size. Increasing the split size results in less number of map tasks per job and more test execution per split. There are trade-offs to consider when setting this parameter. If the value is too small, the test execution wastes startup overhead and results in an inefficient shuffle and sort phase. If the value is too big, Hadoop can’t provide enough parallelism to efficiently utilize the cluster to execute the given Hadoop job [66]. There is also a chance that some nodes in the cluster might fail, if such circumstance happens, a map task given to the node will be restarted on the other node in the cluster. If the split size is large, the more execution time is wasted with the node and might impact the overall performance of the test execution. The following suggestions provide information to help determine the value to fit different situations.

- If there amount of test cases in the test suite is small resulting in a small test case list, it is suggested to consider reducing the split size to be smaller than the size of the test case list to force concurrency in the test execution, otherwise Hadoop won’t even bother distributing the map tasks to the other nodes in the cluster making a test execution with HadoopUnit to be ineffective due to the introduced overheads.

- If the number of nodes in the cluster increases, it may be desirable to consider reducing the split size appropriately to add the degree of
concurrency to the test execution to take advantage of the additional IT resources. This practice might, however, increase the amount of overheads to the overall test execution, as there is more map tasks to set up per a single Hadoop job. On other hand, if the number of nodes in the cluster is small, it might be beneficial to increase the split size to reduce the number of map tasks, so the overhead incurred by them will be reduced. The configuration as found in this experiment can be used for such scenario.

- If the average execution time of the test cases in the test suite is small. In other words, each test case takes only little time to finish its execution, consider increasing the split size to give more map tasks to be executed per node to reduce the effect of Hadoop’s task setup and introduced overhead as mentioned earlier.

For this experiment, each test case took around 32 seconds to finish executing its set of tests. The results from the experiment have shown the delay and the introduced overhead to be insignificant comparing to the performance that could be gained from migrating the traditional test execution to HadoopUnit.

### 4.5 Summary

The experiments have shown a promising result of test execution time after moving from a sequential test execution on a single machine to a distributed test execution with HadoopUnit, even if the size of the cluster is small. A performance gain of up to 4x could be realized on a cluster of four nodes and up to 2x on a cluster as small as two nodes when using HadoopUnit as a testing platform. These improvements could be achieved without a huge investment and make it worth considering moving GUI test of iOS applications to this distributed execution environment.
CHAPTER 5

Summary

5.1 Research Summary

Mobile applications have become the primary tools that are being used to interact with business entities and their quality has become more and more important to the users. However, their associated GUI regression test suites continue to grow larger in number once they are released and maintained for years. At some point, the time requires to execute such test suites can become undesirable and slow down the whole testing process, yet it’s a very important process that could not be avoided.

In an effort to make the testing process faster, HadoopUnit has been adopted to provide a distributed execution infrastructure for system-level GUI testing with a cluster of machines to execute UI Automation test cases concurrently. With faster test execution, groups of developers or software development teams no longer have to wait excessively long for the testing process to complete in order to get their feedbacks with large regression test suites. Moreover, this also means that they can integrate their code more frequently per day thus increases productivity of the teams.

Several experiments were conducted to simulate various scenarios where a small-scale implementation of HadoopUnit could be used in a software project. The setup included the following configurations.

- A single machine, sequential execution
• A 2-Node cluster with HadoopUnit, concurrent execution
• A 4-Node cluster with HadoopUnit, concurrent execution

The first configuration provides a baseline execution time for HadoopUnit, while the other two configurations were chosen to prove the usefulness of HadoopUnit with a much smaller cluster, with just additional two and four nodes. Comparing to the result of normal sequential execution on a single machine, HadoopUnit could provide a performance increase of up to 2 times on two nodes cluster and up to 4 times on four nodes cluster. This indicates that, without a huge investment, the testing process could be made faster with HadoopUnit. This also indicates that the use of HadoopUnit can be greatly effective for agile methodologies since testing is being performed frequently throughout the life cycle. The information in Chapter 4 provides evidences of the success of this thesis.

5.2 Contributions

The primary objective of this thesis was to reduce the times it takes to perform system-level GUI testing of iOS applications by considering moving from a normal sequential test execution over a single machine to a distributed test execution environment with HadoopUnit. Throughout the experimentation, HadoopUnit has been customized and configured to break through several limitations and considerations that testing of iOS applications have on the Mac OS X platform. The results of the experimentation prove the usefulness of HadoopUnit on system-level GUI testing in reducing its test execution time. This contribution opens up a new possibility on what HadoopUnit can be used since HadoopUnit was previously used on the component-level testing like unit testing only.

The secondary objective was to prove the effectiveness of HadoopUnit on a small cluster. The experimental results have shown that even without a huge investment on IT resources, HadoopUnit could be used to reduce the test execution time of iOS applications. This contribution makes HadoopUnit a cost-effective and
a pragmatic choice that any software development teams could adopt to make their testing process faster.

5.3 Results

The objectives that were set for this thesis have been met. HadoopUnit has been successfully set up to perform system-level GUI testing of iOS applications with UI Automation concurrently in a distributed manner. Testing execution time with HadoopUnit outperformed that of the normal test execution sequentially on a single machine. Although some overheads introduced by Hadoop has been discovered during the course of the experimentation, a performance improvement of up to 2x could be realized with a 2-Node cluster and up to 4x with a 4-Node cluster running HadoopUnit.

Huge investment also is not required in order to implement HadoopUnit, since the result of the experiments has shown that the test execution time of the testing could be significantly reduced with just an addition of two nodes or four nodes cluster running HadoopUnit.

In addition to a faster test execution, HadoopUnit also provides testers with several benefits such as automated test case distribution, test result gathering, and test load distribution. And since HadoopUnit was developed on top of Hadoop, a Hadoop cluster running HadoopUnit could be easily scaled up or down as needed.

5.4 Future Work

In addition to the benefits that HadoopUnit provides to an iOS software development project, there are several areas where the work of this thesis with HadoopUnit could be improved. For example, the process and workflow presented in this thesis is done in a semi-automated fashion involving several manual processes to drive the test execution. It could be even better if HadoopUnit could be
integrated with Continuous Integration server, like Hudson or Jenkin, so that HadoopUnit could even help streamline the software development process.

The idea here is that whenever a developer commits a code to the server, the server automatically checks for any syntactical errors and starts rebuilding the project. If the build was a success, some kinds of scripts on the server then could be started up to upload the Xcode project and the source code to every preconfigured node in the clusters via any file transfer protocol such as FTP or any other protocol. Any updated test scripts for automation would also be uploaded along with the project. After finish uploading, each node then automatically starts building the project within the node and install the app in the simulator. When finish, each node reports back to the server. The server updates the test case list to be used by HadoopUnit and start the distributed test execution with HadoopUnit on every node in the cluster. After finishing execution, test results are reported to the server while any trace document of the failure test case is accessible through the server. All of these processes happen in the background without requiring any user-interaction with the system.

It might also be useful to extend this work to support the other major mobile platforms such as Android or Windows Phone so that HadoopUnit could help driving the level of quality of mobile applications in the market forward.

5.5 Concluding Remarks

We now live in a “Post-PC” era with the proliferation of mobile devices; “the rising ubiquity of mobile applications” [67]. The speed of mobile application developments and the quality of applications are now also more important than ever. Software testing is one way to help improve the quality of the applications. If the execution times get in the way to properly performing software testing, HadoopUnit could is a viable option that can help breaking through the obstacle.
REFERENCES


46. Vollmer, A.: Tuneup JS. In: Tuneup JS. Available at: http://www.tuneupjs.org


APPENDIX A

Setting up a HadoopUnit Cluster on Mac OS X

Prerequisites

**Operating System**
Each node that constitutes a HadoopUnit Cluster was operated primarily under the Mac OS X 10.8.5 Mountain Lion run under VMware Fusion Professional 6.0.2.

**Xcode**
Xcode is required to be installed on each node in a cluster in order to have an ability to compile and build iOS working projects and to run the test via Instruments, which is also bundle with Xcode. At the time of writing, Xcode version 5.0.2 (5A3005) was the latest version available, but due to the instability issue with the currently available versions of Xcode, Xcode version 5.1 Beta 4 (5B90f) was primarily used to perform experiments in this thesis.

The following command is a command to switch between versions of Xcode install alongside on a single machine.

```
sudo xcode-select --switch /Applications/Xcode51-Beta4.app
```

To check the current version of Xcode that the machine is working on, use the following command.

```
xcode-select --print-path
```

or

```
xcodebuild -version
```

**Command Line Tools for Xcode**
Command Line Tools for Xcode is required to enable a command-line workflow with UI Automation, including compiling, building, and running the UI Automation testing through the command-line interface. Command Line Tools for Xcode could be obtained from Apple Developer website or directly inside of Xcode under the Download Components Section. Command Line Tools (OS X Mountain Lion) for Xcode – October 2013 was used in this thesis.
unix_instruments
The instruments command, at the time of writing, returns the same status code no matter whether the test case passes or fails. Unix_instruments is a wrapper script that being used in this thesis to detect an error in execution by eavesdrop the output from the instruments commands and return status code appropriately. For the work of this thesis, this command is executed instead of normal instruments command. (https://gist.github.com/jonathanpenn/1402258)

Java
Java is required to be installed to be able to run Hadoop on each node in the cluster. A working installation of Java SE 6 version 1.60_65 for OS X Mountain Lion was used in this thesis. (http://support.apple.com/kb/dl1572)

SSH
Hadoop relies on SSH to communicate between nodes in the cluster and to perform cluster-wide operations. In order to work seamlessly, SSH should be configured to allow keyless/passwordless login for users from machines in the Hadoop cluster.

On Mac OS X, we first need to enable Remote Logins by
1. Go to System Preferences
2. Go to Sharing
3. Check at Remote Logins option
4. Also note the Computer Name here that it will be used as a HOST_NAME during the set up

Then we need to set up a RSA public/private key pair to be able to ssh into the node. On each node (both master and slave), type the following commands to generate RSA key pair of the node.

```bash
ssh-keygen -t rsa -P "" -f $HOME/.ssh/id_rsa
```

The private key is stored in the file specified by the –f option, in this case is $HOME/.ssh/id_rsa, and the public key is stored in the file with the same name but with a .pub extension appended, in this case will be $HOME/.ssh/id_rsa.pub.

Then, we need to make sure that the public key is authorized by copying the public key into $HOME/.ssh/authorized_keys using the following command.

```bash
cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

We need to make sure that host names in the cluster are being configured correctly in the host file at /etc/hosts so that each node can be communicated by its name rather than its IP-address. We then ssh to localhost machine and the actual host
names to make sure that ssh is working correctly. Both the master node and the slave node must be able to ssh to each other. This step will also add the hosts’ fingerprint into the known_hosts file. Note that the HOST_NAME of a Mac OS X machine can be found in Sharing option in System Preferences.

```bash
ssh localhost
ssh USERNAME@HOST_NAME
```

Finally, we need to distribute the public key of the master node to all slave nodes in the cluster. On the master node, execute the following command to append the public key to a remote host that act as a slave node.

```bash
cat $HOME/.ssh/id_rsa.pub | ssh USERNAME@HOST_NAME 'cat >> $HOME/.ssh/authorized_keys'
```

## Hadoop Installation

At the time of writing, the Hadoop version 1.2.1 is the current stable version and was used in this thesis. The current release of Hadoop can be downloaded from the Apache Hadoop Releases website (http://hadoop.apache.org/releases.html), then the downloaded package can be unpacked under the location of our choice. The quick installation of Hadoop can be found in Hadoop Wiki (http://wiki.apache.org/hadoop/QuickStart).

However, the easiest way for Mac OS X to install Hadoop is through the Homebrew tool, a package management manager for OS X (http://brew.sh). The tool can be obtained on the fly using the following command through the command-line terminal.

```bash
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/homebrew/go/install)"
```

After Homebrew has been installed on the machine, Hadoop can be installed with Homebrew simply by using the following command.

```bash
brew install hadoop
```

or using the following command to check the available versions first in order to install a specific version of Hadoop.

```bash
brew search hadoop

brew install homebrew/versions/hadoop121
```
Homebrew will install Hadoop in `/usr/local/Cellar/hadoop/<version>` and will also set $JAVA_HOME to `/usr/libexec/java_home`.

**Hadoop Configuration**

There are 6 configuration files that need to be customized and they are located in `/usr/local/Cellar/hadoop/1.2.1/libexec/conf`:

1. hadoop-env.sh
2. core-site.xml
3. hdfs-site.xml
4. mapred-site.xml
5. masters
6. slaves

**hadoop-env.sh**

This file sets environment variables that are used in the scripts to run Hadoop. Homebrew has already done all the works for us during that simple installation, but there is an issue appeared in Mac OS X Lion and Mountain Lion that requires some configuration in this file to resolve the issue (https://issues.apache.org/jira/browse/HADOOP-7489). We need to add the following line into the file.

```
export HADOOP_OPTS="-Djava.security.krb5.realm=-Djava.security.krb5.kdc="
```

**core-site.xml**

This file set the configuration setting for Hadoop Core such as I/O settings of the nodes. This file is needed to be configured on every node in the cluster.

```xml
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://[MASTER_HOST_NAME]:9000</value>
  </property>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>hdfs://[MASTER_HOST_NAME]:9000</value>
  </property>
</configuration>
```

The `fs.default.name` must point to the master node only with the correct port that the master node is listening to.
The `hadoop.tmp.dir` is the directory got Hadoop to write working temporary files into.
**hdfs-site.xml**
This file controls the configuration for Hadoop Distributed File System process, the name-node, the secondary name-node, and the data-nodes.

```xml
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>4</value>
  </property>
  <property>
    <name>dfs.permissions</name>
    <value>false</value>
  </property>
</configuration>
```

The `dfs.replication` control the number of replication when a file is created in HDFS. In this thesis, our data is the test case list and we want our data to be replicated on every node in the cluster, so this value should be set to the value equals to the number of nodes that we have in the cluster.

The `dfs.permissions` is set to false to avoid permission issue during the execution of our experiment. This value means that any user can do anything to HDFS but since the users need to be able to login to the Mac OS X in the first place, turning this off seems to be reasonable to get rid off all the problems we might encounter.

**mapred-site.xml**
This file controls the configuration of MapReduce process, the job tracker and the tasktrackers.

```xml
<configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>[MASTER_HOST_NAME]:9001</value>
  </property>
  <property>
    <name>mapred.tasktracker.map.tasks.maximum</name>
    <value>1</value>
  </property>
  <property>
    <name>mapred.tasktracker.reduce.tasks.maximum</name>
    <value>1</value>
  </property>
  <property>
    <name>mapred.max.split.size</name>
    <value>1000</value>
  </property>
</configuration>
```

Again, the `mapred.job.tracker` must point to the master node only with the correct port since only the master node runs the job tracker in Hadoop cluster.
The `mapred.tasktracker.map.tasks.maximum` controls the number of map tasks running per node. Since the limitation of iOS simulator that we can only have 1 simulator run at a time, we cannot have more than one map task that run the test on a single node. The default value of this property is set to 2; we have to explicitly set the `mapred.tasktracker.map.tasks.maximum` to the value of 1.

The `mapred.tasktracker.reduce.tasks.maximum` controls the number of reduce tasks running per node. Since the job of the Reduce function is trivial, the `mapred.tasktracker.reduce.tasks.maximum` is set to the value of 1 as well.

The `mapred.max.split.size` is directly correlated to how Hadoop splits and distributes the input file throughout the HDFS. Since our input file, which is just a list of test execution commands, is not big (probably in a unit of a few MB rather than GB or TB). Each test execution is considered computing extensive and time-consuming. We need to set this value so small that it will force Hadoop to distribute the job to the other nodes as well, otherwise, Hadoop won’t even bother distributing the jobs and just run on a single node because Hadoop will consider given input test case list to be a small data. For this experiment, we set this value to 1000 or 1 KB, around 8-10 lines per split.

**masters**
The masters file is a list of machine’s host names or IP-address that each run a secondary name-node (not the machine that runs as the master name-node but the secondary name-node, it could be the master node though). In this experiment, the master node not only acts as the name-node but also as a secondary name-node as well, probably not the best practice since the job of a secondary node is to provide checkpoints of the name-node in case if the name-node fails. So the masters file contains just the following line. This file needs only be set on the master node.

```
MASTER_USERNAME@MASTER_HOST_NAME
```

**slaves**
The slaves file is a list of machine’s host names or IP-address that each run a data-node and TaskTracker in the cluster. The master node can also act as a data-node so the master node can appear in this list as well. This file needs only be set on the master node. The number of nodes in the cluster can be easily added or removed by modifying this file.

```
MASTER_USERNAME@MASTER_HOST_NAME
SLAVE_USERNAME_1@SLAVE_HOST_NAME_1
SLAVE_USERNAME_2@SLAVE_HOST_NAME_2
SLAVE_USERNAME_3@SLAVE_HOST_NAME_3
```
Test Running Hadoop

Before starting our Hadoop cluster, we need to initialize the HDFS first by using the following command.

```
hadoop namenode -format
```

Then to start our Hadoop cluster, we simply need to execute the following command.

```
/usr/local/Cellar/hadoop/1.2.1/libexec/bin/start-all.sh
```

We can test whether our Hadoop cluster is working correctly by running some samples provided with Hadoop installation, for example, with the following command. The command should give us the result text “Estimated value of Pi is 3.14800000000000000000”

```
hadoop jar /usr/local/Cellar/hadoop/1.2.1/libexec/hadoop-examples-*.jar pi 10 100
```

To stop our Hadoop cluster, we can use the following commands.

```
/usr/local/Cellar/hadoop/1.2.1/libexec/bin/stop-all.sh
```

Test Execution with HadoopUnit

The HadoopUnit source code is packaged in .jar format. There are 3 classes, TestDriver.java that contains the main method and drive this MapReduce application, TestMapper that execute the Map function to perform actual test execution, and TestReducer that execute the Reduce function to gather those test results from the Map function and combines them into one report.

To execute testing with HadoopUnit, we need to upload our test case list to the HDFS with the following command.

```
hadoop fs -copyFromLocal <LOCAL_DIRECTORY>/testlist.txt <HDFS_DIRECTORY>/testlist.txt
```

Then we can execute the following command to start the process.

```
hadoop jar <LOCAL_DIRECTORY>/HadoopUnit.jar TestDriver <HDFS_DIRECTORY>/testlist.txt <HDFS_DIRECTORY>/output
```

The HadoopUnit.jar should reside locally in a client machine that wants to run the tests. We do not need to upload this package into the HDFS because Hadoop will
try to find the jar file on the local machine only. The TestDriver is the name of the class that contains the main method that drives our MapReduce application. The testlist.txt is the test case list that was previously uploaded to the HDFS. The output is the directory in the HDFS to which HadoopUnit will store the test results in.

After finish executing the test, the test results can be gathered from the HDFS by using the following command.

```
hadoop fs -copyToLocal <HDFS_DIRECTORY>/output
<LOCAL_DIRECTORY>/output
```

The previous test results could be removed from the HDFS with the following command before the next test could be run (or we could just specify other directory in the HDFS).

```
hadoop fs -rmr <HDFS_DIRECTORY>/output
```

There is an additional step necessary to automate iOS GUI testing on an individual node in the cluster. When the testing with Instruments gets started, there will be a dialog asking for a permission to control another process. This security feature gets in the way of automating our process, as we need to manually acknowledge this dialog every time we need to run the test. To by pass this process, the following command needs to be executed on every node in the cluster.

```
sudo /usr/libexec/PlistBuddy \
-c "Set rights:system.privilege.taskport:allow-root true" \
/etc/authorization
```

The PlistBuddy command will set a specific key necessary to grant permission for Instruments to control other processes in the /etc/authorization file. The permission dialog might appear at first after executing the command, but the process will be on autopilot after that.

Note that in order for a machine to be able to run a test, we first have to agree to the Xcode iOS license, which will require an admin privlege. This is done simply by opening the Xcode the first time and Xcode will prompt us with a dialog asking for an admin password. Also note that we also need to enable the Developer Mode on the machine we want to run the test. This is done by simply opening an Xcode project and starting to build it. Xcode will prompt us with another dialog asking to enable this mode. This is a necessary trivial step we need to take in order to get every ready for testing.
APPENDIX B

HadoopUnit Source Code for iOS GUI Testing

TestDriver.java

```java
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class TestDriver extends Configured implements Tool {

    // The Driver Class
    public int run(String[] args) throws Exception {
        String input, output;
        if (args.length == 2) {
            input = args[0];
            output = args[1];
        } else {
            System.err.println("Incorrect number of arguments. Expected: input output");
            return -1;
        }

        Configuration conf = getConf();
        Job job = new Job(conf);
        job.setJarByClass(TestDriver.class);
        job.setJobName(this.getClass().getName());

        // Specify the input directory from which data will be read,
        // and the output directory to which output will be written.
        FileInputFormat.setInputPaths(job, new Path(input));
        FileOutputFormat.setOutputPath(job, new Path(output));

        // Specify which classes are to be the Mapper and Reducer
        job.setMapperClass(TestMapper.class);
        job.setReducerClass(TestReducer.class);

        // Specify the types of the intermediate
        // output key and value produced by the Mapper
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
    }
}
```
// Specify the types of the Reducer's output key and value.
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);

// Finally, run the job
boolean success = job.waitForCompletion(true);
return success ? 0 : 1;
}

// The main method simply calls ToolRunner.run(),
// passing in the Driver class and the command-line arguments.
public static void main(String[] args) throws Exception {
    TestDriver driver = new TestDriver();
    int exitCode = ToolRunner.run(driver, args);
    System.exit(exitCode);
}

TestMapper.java

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.util.StringTokenizer;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class TestMapper extends Mapper<LongWritable, Text, Text, Text> {

    public void map(LongWritable key, Text value,
            Context context)
            throws IOException, InterruptedException {
        // Get the line
        String inputLine = value.toString();
        StringTokenizer stok = new StringTokenizer(inputLine, "@");

        String testName = stok.nextToken().trim();
        String command = stok.nextToken().trim();
        String resultPath = stok.nextToken().trim();

        // Regular Expression for separating string with white space
        // but not that between single quotes
        // (Process does not play well with a space)
        // String with white space should be put inside single quote

    }
}
ProcessBuilder builder = null;
Pattern p = Pattern.compile("(?<=\\s|\s|^)('|\|\s*)(?=$|\s|\s$)\s");
Matcher m = p.matcher(command);
int index = 0;
while (m.find()) {
    if (index == 0) {
        builder = new ProcessBuilder(m.group(1));
    } else {
        builder.command().add(m.group(1));
    }
    index++;
}
StringBuilder testResult = new StringBuilder();
builder.redirectErrorStream(true);
Process process = builder.start();
OutputStream stdin = process.getOutputStream();
InputStream stderr = process.getErrorStream();
InputStream stdout = process.getInputStream();
BufferedReader reader = new BufferedReader(new InputStreamReader(stdout));
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(stdin));
String line = reader.readLine();
while (line != null) {
    testResult.append(line);
    line = reader.readLine();
}
process.waitFor();
int exitValue = process.exitValue();
if (exitValue == 1) {
    testResult.insert(0, "Failed ");
    // If the test failed, copy the trace document to HDFS
    try {
        // Put some wait time until instruments finish writing to the trace file.
        Thread.sleep(5000);
        Path phdfs_input = new Path(FileOutputFormat.getOutputPath(context).toString().trim()+Path.SEPARATOR+testName.trim());
        Path plocal_input = new Path(resultPath);
        FileSystem fs = FileSystem.get(context.getConfiguration());
        fs.copyFromLocalFile(false, false, plocal_input, phdfs_input);
    } catch (Exception e) {
        e.printStackTrace();
    }
} else {
    testResult.insert(0, "Passed ");
```java
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class TestReducer extends Reducer<Text, Text, Text, Text> {
    @Override
    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        StringBuilder testResult = new StringBuilder();
        for (Text value : values) {
            testResult.append(value.toString());
        }
        context.write(key, new Text(testResult.toString()));
    }
}
```

### Rakefile

```ruby
BUILD_DIR = "/tmp/ExpenseKit"
APP_BUNDLE = "#{BUILD_DIR}/ExpenseKit.app"
AUTOMATION_TEMPLATE = "ExpenseKit/automation/MyTemplate.tracetemplate"
RESULTS_PATH = "ExpenseKit/automation_results"
OUTPUT_TRACE_DOCUMENT = "#{RESULTS_PATH}/Trace"
PROJECT_DIR = "/ExpenseKit/ExpenseKit.xcodeproj"
BUILD_SCHEME = "ExpenseKit"

# If the automation_results directory isn't there, Instruments complains.
mkdir_p RESULTS_PATH
desc "Run appropriate tests for iPhone and iPad Simulators"
task :test, :file do |t, args|

    #receive an argument named file, access it using args[:argument_name]
    file = args[:file]
```
sleep 3

clean_results
build "iphone"
automate file
reset_sim

  puts "\nWin condition acquired!"
end

#
# Composable steps
#

# Remove the automation_results directory and start fresh
def clean_results
  rm_rf RESULTS_PATH
  mkdir_p RESULTS_PATH
end

def clean
  run_xcodebuild "clean"
end

def build type
  case type
  when "iphone"
    sdk = "iphonesimulator"
    fam = "1"
  when "ipad"
    sdk = "iphonesimulator"
    fam = "2"
  when "device"
    sdk = "iphoneos"
    fam = "1,2"
  else
    raise "Unknown build type: #{type}" end

  run_xcodebuild "build -sdk #{sdk} TARGETED_DEVICE_FAMILY=#{fam}"
end

def automate script
  #reset_sim

  if $is_testing_on_device
    device_arg = "-w #{connected_device_id}"
  end

  #env_vars = extract_environment_variables(script)

  sh %{
    ExpenseKit/bin/unix_instruments \ "\#{device_arg}\"
    -t "\#{AUTOMATION_TEMPLATE}\"
    -D "\#{AUTOMATION_TRACE_DOCUMENT}\"
    -D "\#{APP_BUNDLE}\"
    -e UIARESULTSPATH "\#{RESULTS_PATH}\"
  }
def automate_normal script
    # reset_sim
    if $is_testing_on_device
        device_arg = "-w #{connected_device_id}"
    end
    env_vars = extract_environment_variables(script)
    sh %{ instruments \\
         #{device_arg} \\
         -t "#{AUTOMATION_TEMPLATE}" \\
         -D "#{OUTPUT_TRACE_DOCUMENT}" \\
         "#{APP_BUNDLE}" \\
         -e UIARESULTSPATH "#{RESULTS_PATH}" \\
         -e UI_TESTS 1 \ 
         -e UIASCRIPT "#{script}" }
end

def close_sim
    sh %{killall "iPhone Simulator" || true}
end

def reset_sim
    close_sim
    sim_root = "~/Library/Application Support/iPhone Simulator"
    rm_rf File.expand_path(sim_root)
end

# Utility Methods
#

def run_xcodebuild extra_args
    sh %{ xcodebuild \\
            -project "#{PROJECT_DIR}" \\
            -scheme "#{BUILD_SCHEME}" \\
            -configuration Release \\
            CONFIGURATION_BUILD_DIR="#{BUILD_DIR}" \\
            #{extra_args} }
end

def extract_environment_variables script
    lines = File.readlines script
    arguments = []
    lines.each do |line|
        line.match(%r{^// (.+)=(.+)$})
        if $1
arguments << "-e " + $1 + " " + $2
end
end

arguments.join(" ")
end

def ioreg_output
  `ioreg -w 0 -rc IOUSBDevice -k SupportsIPhoneOS`
end

def connected_device_is_ipad?
  !ioreg_output.match(/"USB Product Name" = "iPad"/).nil?
end

def connected_device_id
  ioreg_output.match(/"USB Serial Number" = "([A-z\d]+)"/) && $1
end

def install_on_device
  # I got fruitstrap originally from here:
  # https://github.com/ghughes/fruitstrap
  # # It's no longer supported and you might need to use a fork on Github
  raise "No device connected" if !connected_device_id
  sh %{bin/fruitstrap -b #{APP_BUNDLE} -i #{connected_device_id}}
end