TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Lehrstuhl für Rechnernetze

Cloud-basierte Tests zur Überwachung der Integration und Performanz von mobilen Applikationen im Rahmen von AMCS

Niklas Wünsche
Matrikelnummer: 4503429
Immatrikulationsjahr: 2015

Bachelor-Arbeit

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

Betreuer
Dr.-Ing. Iris Braun
Dr.-Ing. Tenshi Hara

Betreuender Hochschullehrer
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

Eingereicht am: 14.06.2018

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit mit dem Titel *Cloud-basierte Tests zur Überwachung der Integration und Performanz von mobilen Applikationen im Rahmen von AMCS* selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in der Arbeit angegebenen Hilfsmittel und Quellen benutzt. Die wörtlichen und sinngemäß übernommenen Zitate habe ich als solche kenntlich gemacht. Es waren keine weiteren Personen an der geistigen Herstellung der vorliegenden Arbeit beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 14.06.2018

Niklas Wünsche

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur / Professur für Rechnernetze

Aufgabenstellung für die Bachelor-Arbeit

Thema: Cloud-basierte Tests zur Überwachung der Integration und Performanz von mobilen Applikationen im Rahmen von AMCS

Name:	Wünsche, Niklas	Studiengang:	Bachelor Informatik
Matrikel-Nummer:	4503429	Projekt/Fokus:	AMCS
verantwortlicher Hochschullehrer:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill		
Betreuer:	Dr.-Ing. Iris Braun, Dr.-Ing. Tenshi Hara		
beginnen am:	9. April 2018	einreichen bis:	25. Juni 2018

Zielstellung

Am Lehrstuhl Rechnernetze wurden in den vergangenen Jahren mehrere prototypische Untersuchungen zum Einsatz technischer Werkzeuge im Lehrbetrieb durchgeführt. Unter anderem wurden Audience Response Systeme (ARS) in Vorlesungen und Übungen getestet; es wurden für den jeweiligen Einsatz valide Ergebnisse gewonnen. Am Ende der bisherigen Entwicklungen steht eine komplexe Anwendungsarchitektur und -plattform: Auditorium Mobile Classroom Service (AMCS) mit zugehörigen mobilen Applikationen (Apps).

Da es sich um eine über mehrere Iterationen gewachsene Plattform handelt, wurde bereits mit einer Master-Arbeit (Buchholz, 2017) untersucht, in weit sich Tests automatisieren und im Entwicklungsprozess effizient einsetzen lassen. Mit diesen integrierten und automatischen Tests ist die Weiterentwicklung der AMCS-Apps zugänglicher geworden.

Im Rahmen dieser Bachelor-Arbeit soll ein Cloud-Szenario für die Testautomation untersucht werden. Ziel soll dabei eine begründete Aussage bezüglich des Mehrwertes von Cloud-basierten Ansätzen (wie bspw. Firebase) im oben beschriebenen Kontext sein. Ausgehend von (Buchholz, 2017) sollen dabei existierende Cloud-Ansätze untersucht und eingeordnet werden. Anschließend soll das Testbett um die notwendige(n) Cloud-Komponente(n) erweitert werden. Eine abschließende Herleitung der gewünschten Aussage soll im Anschluss durch eine geeignete Evaluation erfolgen. Dazu ist die initiale Definition von Evaluationskriterien zwingend erforderlich. Die notwendige prototypische Implementierung soll für eine der AMCS-Apps (bspw. Android) erfolgen.

Braun

Dr.-Ing. Iris Braun

Schwerpunkte

- Analyse und Auswahl bestehender Ansätze und Lösungen,
- Definieren von Anforderungen,
- Definieren von Bewertungskriterien für die Anforderungserfüllung,
- Konzept zur Cloud-basierten Durchführung von Tests der AMCS-Apps,
- prototypische Umsetzung des Konzepts, sowie
- Evaluation und Auswertung der Ergebnisse.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 In dieser Arbeit verwendete Konventionen

In dieser Arbeit werden jene Begriffe *kursiv* geschrieben, welche noch nicht definiert wurden, aber es in einem späteren Teil der Arbeit werden. Bei der Definition eines Begriffes wird dieser **fett** geschrieben. Sobald ein Begriff definiert ist, wird er in der restlichen Arbeit weder kursiv noch fett geschrieben. Des Weiteren werden Eigennamen, welche aus anderen Arbeiten übernommen wurden, **fett** geschrieben oder in „Anführungszeichen" gesetzt.

Quellen, welche einen Verfasser angeben, werden in das Literaturverzeichnis aufgenommen. Quellen, wie etwa Dokumentationen, in denen kein Verfasser des Inhalts angeben wird, werden in einer Fußnote vermerkt.

In dieser Arbeit werden die Begriffe Programm und Software äquivalent genutzt. Außerdem werden Programmierung und Softwareentwicklung gleichwertig verwendet. Dasselbe gilt für Fehler und Bug. Auch Quelltext und Code werden gleichwertig genutzt.

Zuletzt ist zu sagen, dass das Wort App, solange nicht genauer beschrieben, in dieser Arbeit immer eine Android-Applikation beschreibt.

1.2 Motivation der Aufgabe

Android-Smartphones spielen in der heutigen Zeit eine immer größer werdende Rolle. So haben laut bitkom im August 2017 81% der Deutschen ein Smartphone genutzt (vgl. (Haas, 2018)). Weiterhin verkündete statcounter im April 2017, dass die Anzahl an Android-Geräten, im Bezug auf Desktop-PCs, Laptops und mobilen Geräten, zum ersten Mal höher war als die entsprechende Anzahl an Windows-Geräten. [1]

Viele Menschen nutzen ihr Android-Smartphone auch für wichtige Aufgaben. So wurde die Sparkassen Android-Applikation (**App**) für Android schon über fünf Millionen [2] Mal heruntergeladen. Eine der wichtigsten Aufgaben dieser App ist die Einsicht von Kontodaten und das Ausführen von Überweisungen.

Um diese persönlichen Daten geheim zu halten und ein mögliches Fehlverhalten der Apps zu vermeiden, müssen diese getestet werden. Doch die *Fragmentierung* des Smartphone-Marktes im Bereich der Android-Geräte erweist sich als großes Problem. Es stellt sich die Frage, wie man seine App auf möglichst vielen genutzten Android-Geräten testen kann,

[1] vgl. http://gs.statcounter.com/press/android-overtakes-windows-for-first-time, zuletzt besucht am 24.04.2018

[2] https://play.google.com/store/apps/details?id=com.starfinanz.smob.android.sfinanzstatus, besucht am 03.05.2018

ohne dafür viele verschiedene Geräte kaufen zu müssen.
Eine mögliche Antwort darauf bieten die sogenannten *Device Clouds*. Diese erlauben dem Entwickler das Testen seiner App auf einer Vielzahl von Android-Geräten, welche über das Internet genutzt werden können.

Jedoch ist der Markt der Device Clouds stark umkämpft, sodass es eine Menge von verschiedenen Anbietern für diese Art von Dienst gibt. Es gilt die Frage zu klären, welche Device Cloud genutzt werden sollte.

1.3 Zielsetzung der Arbeit

Diese Arbeit beschäftigt sich mit der Frage, ob eine Device Cloud für die Ausführung von *Integration Tests* im Rahmen der Android-App des Projektes Auditorium Mobile Classroom Service (*AMCS*) [3], eingesetzt werden sollte. Außerdem wird eine Antwort darauf gegeben, welcher Device Cloud Anbieter unter welchen Rahmenbedingungen genutzt werden sollte.

Dabei wird zunächst überprüft, welche Vorteile das automatisierte Testen von Apps hat. Zuerst wird dazu auf den aktuellen Stand des Testens von Android-Apps im Allgemeinen, sowie der AMCS Android-App im Speziellen, eingegangen. Darauf aufbauend werden im Hauptteil dieser Arbeit verschiedene Device Clouds betrachtet und eine prototypische Implementierung ausgewählter Beispiele vorgenommen.
Zum Schluss der Arbeit wird die Frage geklärt, welche Vorteile die Nutzung einer Device Cloud mit sich bringt. Es wird konkret für die AMCS Android-App angegeben, welche Device Cloud der Autor der Arbeit weiterhin nutzen würde.

1.4 Motivation der Zielsetzung

Das Ziel, welches sich diese Arbeit setzt, basiert auf drei verschiedenen Arbeiten.
Die Idee der Arbeit wurde in (Buchholz, 2017) formuliert. In dieser Masterarbeit wurde unter anderem ein System für die AMCS Android-App erarbeitet, welches das automatisierte Testen der App ermöglicht. Besonders hervorzuheben ist, dass diese Masterarbeit einen Ausblick gibt, in welchem die Nutzung einer Device Cloud für das Testen der AMCS-App erwähnt wird.
Weiterhin wurde eine ähnliche Leistung wie die in der hier vorliegenden Arbeit von (Martínez, 2017) verrichtet. Das Auswahlkriterium, nachdem die Device Clouds in diesem Blogbeitrag selektiert wurden, ist die Popularität der einzelnen Device Clouds. Die Bachelorarbeit wird sich allerdings auf andere Anforderungen stützen, wie beispielsweise die Kosten der einzelnen Device Clouds.
Zuletzt muss auch die Arbeit (Braun, Elberzhager & Holl, 2017) erwähnt werden. Diese Arbeit kategorisiert viele Android Test-Tools. Insbesondere werden in der genannte Arbeit Device Clouds als eine Kategorie genutzt. Jedoch geht die genannte Arbeit nicht genauer darauf ein, inwieweit die einzelnen vorgestellten Tools miteinander genutzt werden können. Es wird in dieser Quelle explizit geschrieben, dass dieses Zusammenspiel von den Entwicklern selber genauer untersucht werden muss. Da in der AMCS Android-App unter anderem das „Functional Test Automation“ Tool Espresso genutzt wird, wird das Zusammenspiel dieses Testprogramms mit den in dieser Quelle genannten Device Clouds untersucht.

[3] https://amcs.website/, zuletzt besucht am 03.05.2018

1.5 Aufbau der Arbeit

Nach diesem einleitenden Kapitel folgen sechs weitere. Im zweiten Kapitel werden die Grundlagen für diese Arbeit definiert und der aktuelle Stand der Technik vorgestellt. Dabei werden wichtige Kernbegriffe der Arbeit, wie *Testen*, *Device Cloud* und *kontinuierliche Softwareentwicklung* definiert.

Im folgenden Kapitel wird der aktuelle Stand des AMCS-Systems dargestellt. Hierbei dient (Buchholz, 2017) als Grundlage. Es wird betitelt, welche Probleme zurzeit noch bei der Ausführung der Tests auftreten und es werden die aktuellen Rahmenbedingungen des Systems dargelegt. Zuletzt werden in diesem Kapitel die Anforderungen zum Vergleich der Device Clouds zusammengefasst.

Kapitel vier beinhaltet das Konzept der Implementierung und Evaluation. Es wird beschrieben, wie die prototypische Implementierung der Device Clouds aussehen und wie ein Vergleich derer durchgeführt wird.

Im fünften Kapitel der Arbeit geht es um die prototypische Implementierung der herausgearbeiteten Device Clouds anhand des Konzepts. Zunächst wird darauf eingegangen, welche Android-Geräte zum Testen der App genutzt werden sollten, um eine möglichst große Anzahl von Nutzern abzudecken. Es werden die wichtigsten Schritte zur Integration der jeweiligen Device Clouds vorgestellt. Außerdem wird auf auftretende Probleme bei der Implementierung eingegangen.

Das anknüpfende Kapitel sechs besteht aus der Evaluation der Prototypen. Es werden diejenigen Device Clouds benannt, welche die zugrunde liegenden Anforderungen am besten erfüllen. Es wird überprüft, ob aktuelle Probleme bei der Testausführung im AMCS-System gelöst werden konnten.

Das siebte und letzte Kapitel dieser Arbeit umfasst zunächst eine Zusammenfassung dieser Arbeit. Der Autor wird außerdem die Device Cloud benennen, welche im Rahmen von AMCS weiter genutzt werden sollte. Zuletzt wird ein Ausblick auf fortführende Themen gegeben, welche auf der hier vorliegenden Arbeit aufbauen und untersucht werden können.

2 Grundlagen und State of the Art

In diesem Kapitel werden grundlegende Begrifflichkeiten eingeführt, welche im weiteren Verlauf dieser Arbeit genutzt werden. Zuletzt wird anhand von zwei Fallbeispielen die Nutzung von *kontinuierlicher Softwareentwicklung* in zwei Firmen vorgestellt.

2.1 Testen von Apps

Das Testen von Software ist zwingend nötig. Da es laut („ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:Concepts and definitions“, 2013) weitestgehend anerkannt ist, dass es nicht möglich ist, perfekte Software zu schreiben, muss diese ausreichend getestet werden, bevor man sie an den Nutzer verbreiten kann.

Ein **Test** ist eine konkrete Abfolge von Handlungen, um einen konkreten Aspekt eines Systems oder einer Komponente zu überprüfen. Ein Test kann entweder erfolgreich sein oder fehlschlagen. Die Menge all dieser Ergebnisse wird als **Testbericht** oder **Testresultat** definiert. Diese kann noch weitere Informationen, wie etwa die Ausführungszeit der Tests, beinhalten.
Testen ist eine Aktivität, in welcher ein System oder eine Komponente unter vorgegebenen Bedingungen ausgeführt wird. Die Ergebnisse werden überwacht und aufgezeichnet. Eine Begutachtung eines Aspekts des Systems oder der Komponente wird vollzogen (vgl. („ISO/IEC/IEEE International Standard - Systems and software engineering – Requirements for testers and reviewers of information for users“, 2017)). Testen besteht aus der Ausführung von einem oder mehreren Tests.

Da die Definition des Testens noch sehr weit gefasst ist, werden nun die Kategorien *manuelles Testen* und *automatisiertes Testen* eingeführt.

Manuelles Testen ist das Testen von Software durch einen echten Nutzer. Dieser sogenannte **manuelle Tester** versucht genau das zu machen, was ein normaler Nutzer der Software machen würde. Dies schließt sowohl die geplante, als auch die ungeplante Nutzung der Komponenten ein (vgl. (Shrivatri, 2016)).

Automatisiertes Testen ist das Testen von Software unter Zuhilfenahme spezieller Testsoftware bzw. eines Testframeworks. Dabei wird das Ausführen der geschriebenen Tests von dieser Testsoftware übernommen. Das Ausliefern der Testergebnisse an den Entwickler wird ebenso von einem Computer übernommen. Somit wird die Software durch einen Computer getestet und nicht durch einen echten Nutzer (vgl. (Vasylyna, 2011)).

Das automatisierte Testen von Software hat viele Vorteile gegenüber dem manuellen Testen. In (Shrivatri, 2016) genannte Vorteile sind:

- Es nimmt die Belastung von dem Entwickler, welcher die Software sonst per Hand testen muss.
- Automatisierte Tests können über Nacht laufen, ohne dass ein Entwickler deren Ablauf überwachen müsste. Die Auswertung der Testergebnisse kann am nächsten Tag stattfinden.
- Die exakte Wiederholung von Tests ist durch einen Computer einfacher zu realisieren als durch einen Nutzer, welcher (versehentlich) falsche Anweisungen ausführen könnte.

Jedoch hat das automatisierte Testen auch Nachteile. Ein wichtiger Aspekt ist, dass man die Tests zunächst programmieren muss, bevor diese vom Computer umgesetzt werden können. Häufig ist das Anleiten eines menschlichen Testers schneller. Auf spezifische Probleme, welche beim Testen von Apps auftreten, wird in Abschnitt 2.1.3 genauer eingegangen.

Im weiteren Verlauf der Arbeit wird der Begriff Testen mit dem automatisierten Testen gleichgesetzt. Sollte von manuellen Tests die Rede sein, wird dies explizit angegeben.

2.1.1 Arten von automatisierten Tests

Um im späteren Verlauf der Arbeit den aktuellen Stand der AMCS Android-App und die daraus resultierenden Anforderungen zu verstehen, ist es zwingend nötig, die Begriffe *Unit Test* sowie *Integration Test* zu definieren. Weitere Testarten wurden in (Buchholz, 2017) vorgestellt.

Ein **Unit Test** ist ein Test, welcher schnell ausführbar ist, da er einen kleinen, isolierten Teil der Software betrachtet. Insbesondere testet er nicht die Beziehungen von Komponenten der Software untereinander (vgl. (Fowler, 2014)).

Ein **Integration Test** ist ein Test, welcher überprüft, ob mehrere Komponenten der Software korrekt miteinander interagieren. Seine Ausführung dauert häufig länger als der eines Unit Tests (vgl. (Fowler, 2018)). Im Rahmen dieser Arbeit sind Integration Tests oft Tests, welche mit der graphischen Oberfläche des Programms interagieren und über diese das Programm steuern. Integration Tests werden in dieser Arbeit oft nur Test genannt.

Die beiden Begriffe werden nun anhand von Beispielen veranschaulicht. Dabei wird eine Funktion getestet, welche für die Validierung eines Namens während eines Registrierungsvorganges genutzt wird. Um dies durchzuführen, wird beispielsweise ein Test geschrieben werden, welcher überprüft, ob die Funktion einen Fehler wirft, wenn ein Name übergeben wurde, welcher eine Zahl beinhaltet. Diesen Test realisiert man durch einen Unit Test. Die einzelne Komponente, die getestet wird, ist hierbei die isolierte Funktion innerhalb der Software.

Möchte man nun testen, ob der Nutzer eine Fehlermeldung angezeigt bekommt, wenn er sich mit einem schon vorhandenen Namen zu registrieren versucht, ist ein Integration Test notwendig. Hierbei wird das Zusammenspiel von mehreren Komponenten, nämlich das vom Eingabefeld, Datenbank und der graphischen Benutzeroberfläche, getestet.

2.1.2 Nutzung von Tests in Softwareprojekten

Obwohl das Testen von Software eine gute Voraussetzung darstellt, um qualitativ hochwertige Programme auf den Markt zu bringen, wird der Aufwand des Testens nicht immer

eingegangen.
So zeigt (Kochhar, Thung, Lo & Lawall, 2014), dass die durchschnittliche Testabdeckung der 300 größten Java Open-Source Projekten gerade einmal bei 42% liegt. Für Android-Apps schreibt (Silva, Endo, Eler & Durelli, 2016), dass von 19 untersuchten Android-Apps nur 47% eine Form von automatisierten Tests nutzen. In (Kochhar, Thung, Nagappan, Zimmermann & Lo, 2015) wird gezeigt, dass von 600 auf F-Droid verfügbaren Android-Apps gerade einmal 14% automatisierte Tests nutzen.
Nachfolgend werden die wichtigsten Gründe angeführt, warum Android-Apps selten getestet werden.

2.1.3 Probleme beim Testen von Apps

In diesem Abschnitt wird eine Übersicht der Probleme gegeben, welche beim Testen von Android-Apps auftreten. Es werden nur Probleme benannt, welche im Rahmen dieser Arbeit relevant sind. Eine umfassendere Aufzählung kann in (Linares-Vásquez, Moran & Poshyvanyk, 2017) gefunden werden.

Es gibt eine Vielzahl von Gesten, um mit seiner Android-App zu interagieren. Leider bieten nach (Linares-Vásquez et al., 2017) viele Testframeworks keine Möglichkeit an, komplexe Gesten wie das Auseinanderziehen von zwei Fingern zum Vergrößern des Bildschirmausschnittes zu simulieren. Demzufolge werden nur die Komponenten einer App automatisiert getestet, welche mit einfachen Gesten, wie dem Drücken eines Buttons auf dem Bildschirm, auskommen. In Kapitel 2.7 wird genauer darauf eingegangen, dass die App-Entwickler Facebook und Etsy aus diesem Grund ihre App manuell testen.

Ein weiteres großes Problem ist nach (Linares-Vásquez et al., 2017) die *Fragmentierung* des Android-Modell Marktes. Die **Fragmentierung** dieses Marktes bezeichnet dabei die große Menge an verschiedenen Android-Geräten, welche existiert. Dabei ist ein **Android-Gerät (Gerät)** eine konkrete Ausprägung des Android-Modells und einer Version des Android Betriebssystems (**OS-Version**).
Während das Samsung Galaxy S8 beispielsweise ein Android-Modell darstellt, ist ein Android-Gerät ein Samsung Galaxy S8 mit der OS-Version 8.0 .
Opensignal[4] zeigte bereits 2015, dass es über 24.000 verschiedene Android-Modelle gibt. Alleine diese Zahl führt zu einer großen Anzahl verschiedener Android-Geräte.
Da es bei Apps vorkommen kann, dass gewisse Fehler nur auf speziellen Android-Geräten auftreten, müsste man alle Tests auf allen Geräten durchführen. Dies kann allerdings aus Zeit- und Kostengründen nur von wenigen Entwicklern umgesetzt werden.

Eine günstige Alternative bieten *Emulatoren* und *Simulatoren*. Diese beiden Begriffe werden in dieser Arbeit synonym verwendet. Ein **Emulator/Simulator** ahmt ein echtes Android-Gerät mitsamt aller Software des Gerätes auf einem Computer nach.
Durch Emulatoren kann man die Tests einer App parallel auf einer Vielzahl von emulierten Geräten auf ein und demselben Computer ausführen, ohne diese physisch erwerben zu müssen. Des Weiteren kann laut (Rohrman, 2017) der zeitliche Abstand zwischen dem Entwickeln von neuen Komponenten der App und der Verifikation durch Tests durch das Nutzen von Emulatoren drastisch verringert werden. Dies liegt daran, dass man sich nicht um die Auslieferung der App an ein echtes Gerät kümmern muss, sondern der Emulator meist auf dem Computer des Entwicklers ausgeführt werden kann.
Der größte Nachteil ist, dass Emulatoren in ihrem Umfang sehr eingeschränkt sind. So erläutert (Hechtel, 2016), dass komplexe Gesten nur schwer simulierbar sind, Performance-

[4] https://opensignal.com/reports/2015/08/android-fragmentation/, zuletzt besucht am 02.05.2018

Eigenschaften wie der Verbrauch des Akkus nur schwer erfasst werden können und sie langsamer als die echten Android-Geräte arbeiten. Diese Aspekte können nur mit einem echten Android-Gerät untersucht werden.

Jedoch ist bei der Nutzung echter Geräte der Kostenaufwand ein Problem. Weiterhin entsteht ein höherer logistischer Aufwand als bei dem Einsatz von Emulatoren, da man sich darum kümmern muss, dass die echten Geräte dauerhaft einsatzbereit sind und eine Verbindung von der Testsoftware zu allen Geräten hergestellt werden kann.

Um den Kosten- und Logistikaufwand der echten Geräte zu reduzieren, kommen (Hechtel, 2016) und (Kelly, 2016) zu dem Schluss, dass man eine *Device Cloud* nutzen kann, welche echte Android-Geräte zum Testen über das Internet anbietet. Im nächsten Abschnitt wird sich deshalb mit dem Gebiet des *Cloud Computings* auseinandergesetzt.

2.2 Cloud Computing

Das Ziel von **Cloud Computing** ist „die Bereitstellung skalierbarer IT-Dienste über das Internet für eine potenziell große Zahl externer Kunden mit sehr heterogenen Anwendungen“ (Baun, Kunze & Ludwig, 2009). Eine **Cloud** stellt dabei ein großes Rechenzentrum mit entsprechend hoher Speicherkapazität dar, welches diese Anwendungen bereitstellt (vgl. (Baun et al., 2009)). Ein IT-Dienst, welcher die Anforderungen des Cloud Computings erfüllt, heißt **Cloud Dienst**. Viele dieser Cloud Dienste sind über Webseiten im Browser nutzbar. In dieser Arbeit wird häufig von Diensten statt von Cloud Diensten gesprochen.

2.2.1 Kategorisierung von Clouds

Eine wichtige Teilung der Rubrik der Cloud ist die Untergliederung in *public Clouds* und *private Clouds*. Eine Firma, die eine **public Cloud** betreibt, bietet die damit verbundenen Cloud Dienste der Öffentlichkeit an (vgl. (Mell & Grance, 2011)). Die Nutzer des Dienstes teilen sich die zugrunde liegende Infrastruktur. Im Kontrast zu den public Clouds stehen die Dienste von **private Clouds** nur ausgewählten Nutzern zur Verfügung (vgl. (Mell & Grance, 2011)). Es sei darauf hingewiesen, dass ein Unternehmen auch eine private Cloud nutzen kann, welche von einem anderen Unternehmen bereitgestellt wird. Dann muss allerdings die Bedingung erfüllt sein, dass die entsprechend genutzte Infrastruktur bei dem Betreiber explizit dem Interessenten zugeordnet werden kann. Wenn jedoch der Nutzer und der Anbieter der private Cloud das gleiche Unternehmen darstellen, so spricht man in diesem Fall von einer **on-premises Cloud** (vgl. (Hoffmann, 2016)). Dabei kann die Software, welche die on-premises Cloud anbietet, auch von einer fremden Firma gekauft sein (vgl (Karlstetter, 2017)). Wenn man eine on-premises Cloud nutzt, so muss sich der Nutzer mit der zugrunde liegenden Infrastruktur auseinandersetzten. Dies ist bei den anderen Arten von Clouds nicht der Fall.

Es sei angemerkt, dass es auch die **hybrid Clouds** gibt, welche eine Mischform von public und private Clouds darstellen (vgl. (Mell & Grance, 2011)).

Eine Untergliederung in die oben genannten vier Kategorien Public, Private, On-Premises und Hybrid ist auch bei den Cloud Diensten möglich. Hierbei richtet sich die Kategorie des Dienstes nach der zugrunde liegenden Cloud.

Für Cloud Dienste ist das Bezahlmodell *On-Demand* weit verbreitet. Wenn ein Cloud Dienst **On-Demand** ist, bedeutet dies, dass man für die konkrete Nutzung des Dienstes bezahlt. So bezahlt man zum Beispiel bei der Nutzung des Blaze Plans der *Device Cloud*

Firebase Test Lab[5] für die Nutzung eines Android-Geräts pro Stunde.

2.2.2 Vor- und Nachteile von Cloud Diensten

Im folgenden Abschnitt soll genauer betrachtet werden, wann es sich lohnt, einen Cloud Dienst einer anderen Firma zu nutzen und wann eine interne Lösung zu bevorzugen ist.

Ein Vorteil von Cloud Diensten ist die starke Skalierbarkeit. Wenn man beispielsweise testen möchte, ob die eigene Software mit 1000 verbundenen Geräten genauso funktioniert wie mit einer Million verbundenen Geräten, dann kann man diese Zahl bei einem Cloud Dienst mühelos erhöhen. Bei einer internen Lösung müsste man sich selber um die Inbetriebnahme der hinzuzufügenden Geräte kümmern.[6]

Die Nutzung von Cloud Diensten hat zudem den Vorteil, dass die Kostenverteilung über die Zeit der Nutzung gleichmäßiger als bei dem Aufbau einer internen Lösung ist. Hier sind die Anschaffungskosten zunächst sehr hoch, während bei einem Cloud Dienst ein konstanter Betrag zu bezahlen ist. Des Weiteren fallen Zeitprobleme, welche durch den Ausfall der internen Infrastruktur hervorgerufen werden können, fast vollständig weg.[6].

Jedoch führt die Nutzung eines Cloud Dienstes dazu, dass man keine volle Kontrolle über die Konfiguration der genutzten Infrastruktur hat. Es ist zum Beispiel nicht möglich, bei einem Cloud Dienst ein neues Android-Gerät zu nutzen, wenn dieses nicht explizit vom Betreiber bereitgestellt wird. Bei einer internen Lösung könnte man dieses Android-Gerät mit weniger Aufwand an den eigenen Server anschließen.

Problematisch bei der Nutzung von Cloud Diensten ist zudem die Datensicherheit. Als Motivation soll dienen, dass 49% der weltweit befragten „IT-Entscheidungsträger" große Angst vor den Auswirkungen an die Sicherheit haben, wenn sie einen Cloud Dienst nutzen würden.[7] Der Diebstahl von sensiblen Daten, wie zum Beispiel bei dem Finanzdienstleistungsunternehmen Equifax[8], geben zu bedenken, ob man großen Unternehmen bei der Speicherung sensibler Daten vertrauen sollte.

Im Gegensatz dazu schreibt (Geroch & Zabieglinska–Lupa, 2017), dass auch die Nutzung einer internen Lösung nicht zwingend sicher sein muss. Wenn man seine eigenen Sicherheitsstrukturen von Grund auf errichten muss, nimmt dies viel Zeit und Geld in Anspruch. Werden diese Ressourcen nicht investiert, können die Daten in einer Cloud sicherer verwahrt sein als auf einem internen Server.

Wie gezeigt wurde, bieten sowohl Cloud Dienste als auch interne Lösungen Vor- und Nachteile. Jeder muss hierbei selber abwägen, wie er die angeführten Eigenschaften wichtet und welche der beiden Lösungen er bevorzugt.

2.3 Arten von Cloud Diensten

Dienste, welche über eine Cloud angeboten werden, können sehr vielfältig sein. Es bietet sich deshalb an, diese zu untergliedern. Im Folgenden werden dazu *∗-as-a-Service (∗aaS)*-Kategorien etabliert. Hierbei wird, abhängig von der jeweiligen Kategorie, der Asterisk durch einen anderen Begriff ausgetauscht. Weitere ∗aaS-Kategorien können in (Grohmann, 2010) gefunden werden.

[5] https://firebase.google.com/pricing/, zuletzt besucht am 07.05.2018

[6] vgl. https://www.spiceworks.com/it-articles/iaas-and-saas-vs-onprem/, zuletzt besucht 09.05.2018

[7] vgl. https://www.globalservices.bt.com/uk/en/news/business_trust_in_data_security_in_cloud_at_all_time_low, zuletzt besucht am 07.05.2018, Kopie unter: https://bitbucket.org/NWuensche/ba-enclosures/src/master/AngstSecurityCloud.pdf

[8] https://www.ftc.gov/equifax-data-breach, zuletzt besucht am 07.05.2018

2.3.1 Software-as-a-Service

In die Kategorie **Software-as-a-Service (SaaS)** fallen all die Cloud Dienste, bei denen der Nutzer Zugriff auf Software bekommt, welche in einer Cloud läuft (vgl. (Mell & Grance, 2011)). Insbesondere läuft die Software nicht auf dem Computer des Nutzers. Er bekommt nur eine Oberfläche geboten, um mit der Software zu interagieren.

2.3.2 Platform-as-a-Service

Alle Cloud Dienste, welche in die Kategorie **Platform-as-a-Service (PaaS)** fallen, bieten einem Nutzer Zugriff auf eine Entwicklungsumgebung. Auf dieser kann er eigene Software veröffentlichen (vgl. (Mell & Grance, 2011)). Unter diesem Betrachtungspunkt kann man diese Kategorie als Teil von SaaS zählen. Es werden Betriebssysteme, Datenbanken und Werkzeuge zur Ausführung von eigenen Programmen bereitgestellt, jedoch kann die veröffentlichte Software vom Nutzer selber verändert werden.

2.3.3 Infrastructure-as-a-Service

Die Kategorie **Infrastructure-as-a-Service (IaaS)** bezeichnet all diejenigen Cloud Dienste, welche dem Nutzer Zugriff auf physische Infrastruktur oder eine virtuelle Maschine über das Internet bieten. Insbesondere kann der Nutzer selber entscheiden, welche Software auf der bereitgestellten Infrastruktur laufen soll und welche nicht (vgl. (Mell & Grance, 2011)). IaaS bildet unter diesem Betrachtungspunkt eine Unterkategorie von PaaS, da der Nutzer eines IaaS Dienstes die bereitgestellte Software noch genauer anpassen kann.

2.3.4 Testing-as-a-Service

In die Kategorie **Testing-as-a-Service (TaaS)** fallen all die Cloud Dienste, welche dem Nutzer Zugriff auf Testsoftware bieten, die in einer Cloud läuft (vgl. (Linthicum, 2009)). Es wird sowohl die Infrastruktur, als auch die entsprechende Testsoftware bereitgestellt, welche das Testen der eigenen Software ermöglicht.

2.3.5 Einordnung der Device Clouds

Die Device Clouds können, je nach der zugrunde liegenden Technologie, in mehrere der genannten $*$aaS-Kategorien fallen.

Zunächst ist nach (Mtibaa, Fahim, Harras & Ammar, 2013) eine **(Mobile) Device Cloud** eine Infrastruktur, in der Berechnungen auf einer Menge von mobilen Geräten stattfinden.

Definiert man eine Device Cloud so, dass ein Zugang zu einer Vielzahl von verschiedenen Android-Geräten bereitgestellt wird, deren Software man beliebig verändern kann, dann fällt eine Device Cloud in die Kategorie IaaS. Hierbei müsste man beispielsweise die entsprechende Testsoftware oder auch die OS-Version des Android-Gerätes manuell installieren.

Wenn man jedoch Android-Geräte über die Cloud angeboten bekommt, welche voll funktionsfähig sind, und mit Hilfe eigener Software eine Ausführung von Tests auf Android-Geräten durchführen kann, so kann eine Device Cloud auch ein PaaS Dienst sein. Hierbei werden beispielsweise schon Android-Geräte mit dem entsprechenden Betriebssystem angeboten und der Nutzer kann die Android-Geräte zum Testen seiner App nutzen. Jedoch könnte er die Android-Geräte auch für andere Zwecke nutzen. Im Jahre 2014 hat Etsy zum Beispiel versucht, eine solche on-premises Device Cloud selber aufzubauen (vgl. (Kammah, 2014)).

Zuletzt kann man eine **Device Cloud** jedoch auch so definieren, dass man die Definition einer (Mobile) Device Cloud um die Ziele von „Mobile TaaS“ aus (Gao, Tsai, Paul, Bai & Uehara, 2014) erweitert. Somit kann man Android-Geräte oder Emulatoren über einen Cloud Dienst für die Ausführung von Tests nutzen. Zusätzlich kann man die Android-Geräte nicht direkt ansprechen. Dann zählt man Device Clouds in die Kategorie TaaS. Dies ist auch die Definition, welche von nun an in dieser Arbeit genutzt wird.

Ein großer Vorteil bei der Nutzung von Device Clouds besteht darin, dass diese den Entwickler bei dem im Abschnitt 2.1.3 benannten Problem der Fragmentierung des Android-Geräte Marktes unterstützen. Durch die Vielzahl der angebotenen Android-Geräte in der Device Cloud können Entwickler ihre App auf vielen verschiedenen Geräten testen, ohne dass sie sich diese tatsächlich kaufen müssen.

Betrachtete Device Clouds

Die Device Clouds, welche im Rahmen dieser Arbeit untersucht werden, wurden der Arbeit (Braun et al., 2017) entnommen. Es handelt sich dabei um 16 verschiedene Device Clouds, welche in der folgenden Abbildung dargestellt sind.

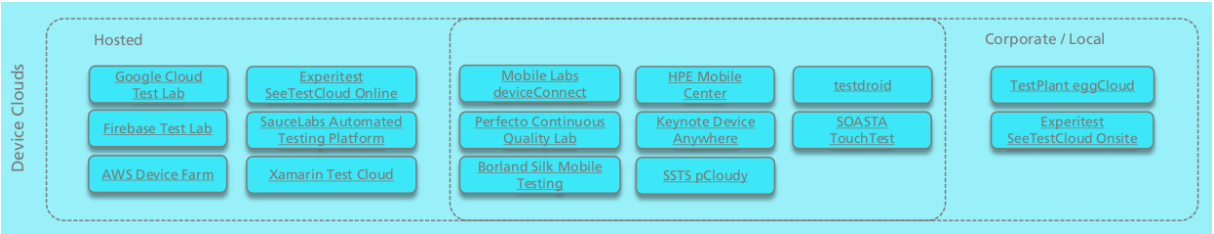


Abbildung 2.1: Device Clouds (Braun, Elberzhager & Holl, 2017)

Wie in der Abbildung zu sehen ist, werden diese Device Clouds in die Kategorien „Corporate/Local“ und „Hosted“ eingeteilt. Da in (Braun et al., 2017) nicht genau definiert wird, für was Corporate/Local steht, wird davon ausgegangen, dass es sich hierbei um on-premises Clouds bzw. interne Lösungen handelt. Demzufolge beschreibt die Kategorie Hosted alle Device Clouds, welche als Cloud Dienst angeboten werden. Einige der Device Clouds fallen in beide Kategorien, da sie in mehreren Varianten angeboten werden. Eine genauere Betrachtung dieser 16 Device Clouds findet in Kapitel 5 statt.

Device Clouds bieten die Möglichkeit, Software *kontinuierlich* auf echten Android-Geräten ohne Einrichtungsprobleme bei der Infrastruktur zu testen. Der folgende Ausblick in die *Agile Softwareentwicklung* soll hierbei als Motivation für die *kontinuierliche Softwareentwicklung* dienen.

2.4 Agile Softwareentwicklung

Die **Agile Softwareentwicklung** bezeichnet eine Menge von Ansätzen der Softwareentwicklung, welche zu einem schnelleren Einsatz der Software führen sollen. Dafür muss man schnell (**agil**) auf Probleme eingehen und häufig eine lauffähige Version der Software veröffentlichen können (vgl. (Eckstein, 2012)). Um dieses Konzept näher zu begründen, wird zunächst der Begriff der *kontinuierlichen Softwareentwicklung (KS)* definiert. **Kontinuierliche Softwareentwicklung** ist hierbei ein Oberbegriff für eine Reihe von Praktiken, an deren Basis es steht, dass Entwickler häufig (oft mehrfach pro Tag) ihren geschriebenen

Quelltext mit einem Server abgleichen.[9] Auf diesem Server wird die aktuelle Version des Quelltextes gespeichert. Der Server kann zusätzlich z.B. die Testumgebung ausführen, um schnell auf Fehler hinweisen zu können. Eine Ausweitung dieses Konzepts wird in Abschnitt 2.6 stattfinden, jedoch ist dies zunächst für die Erklärung der agilen Softwareentwicklung nicht nötig. Es gibt viele verschiedene Ansätze, um agil zu programmieren. Als Beispiel seien hier Scrum und Extreme Programming (XP) genannt. [10] All diese Ansätze bauen auf dem „Manifest für Agile Softwareentwicklung“ (vgl. (Fowler & Highsmith, 2001)) auf. Für diese Arbeit sind im Besonderen die „Prinzipien hinter dem Agilen Manifest“ interessant, welche nun genau betrachtet werden. Sie werden folgend nur Prinzipien genannt.

2.4.1 Wichtige Prinzipien

Einige der Prinzipien zeigen, dass die kontinuierliche Softwareentwicklung sehr wichtig ist, um agil zu arbeiten.

Im ersten Prinzip wird gesagt, dass die Zufriedenheit des Kunden die oberste Priorität genießt. Diese möchte man unter anderem durch kontinuierliche Softwareauslieferung sicherstellen.

Das dritte Prinzip sagt aus, dass man Software regelmäßig und in kurzen Zeitspannen veröffentlichen soll. Dabei ist von Zeitspannen im Wochen- bis Monatsbereich die Rede. Die KS kann eine Unterstützung zum Erreichen dieses Ziels bieten.

Das siebte Prinzip schildert, dass das wichtigste Maß zum Anzeigen von Fortschritt funktionierende Software ist. Auch hier kann einem die kontinuierliche Softwareentwicklung helfen, da Tests ständig ausgeführt werden.

Wie gezeigt wurde, ist die kontinuierlicher Softwareentwicklung ein wichtiges Hilfsmittel der Agilen Programmierung. Ohne KS kann man diverse Prinzipien des Agilen Manifests nicht umsetzen.

Das Beherrschen dieser Prinzipien ist wichtig, da in vielen Firmen agil programmiert wird. So zeigt eine Umfrage der Firma Version One[11], dass 97% der Befragten angaben, dass in ihrem Unternehmen ein agiler Entwicklungsansatz genutzt wird. Des Weiteren wird geschrieben, dass 54% dieser Firmen *kontinuierliche Integration* betreiben, was eine Form der kontinuierlichen Softwareentwicklung darstellt.

Somit ist es durchaus wichtig, sich mit der KS auseinander zu setzen, da diese in immer mehr Softwareprojekten Einzug hält. Aus diesem Grund wird hier die erste Anforderung definiert, welche an die Device Clouds gestellt werden wird. Entwickler müssen schnell erfahren, ob der von ihnen entwickelte Quelltext fehlerfrei ist (**A1**). Aus diesem Grund soll überprüft werden, wie viele Sekunden zwischen der Ausführung von Tests und der Übertragung der Testresultate an den Entwickler vergehen. Eine ähnliche Anforderung wurde auch in (Kuo, Liu & Yu, 2015) gestellt. In dieser Quelle wurde untersucht, ob die Nutzung einer virtuellen Maschine die Ausführungszeit von Tests von Android-Apps tatsächlich verringert.

Da die *Versionskontrolle* ein Grundbaustein der *kontinuierlichen Softwareentwicklung* ist, werden die *Versionskontrolle* und die wichtigsten Begriffe dieser als nächstes erläutert.

[9] vgl. https://www.thoughtworks.com/continuous-integration, zuletzt besucht am 10.05.2018

[10] vgl. https://www.versionone.com/agile-101/, zuletzt besucht am 07.05.2018

[11] https : / / content . cdntwrk . com / files / aT05NjM2NTEmdj0yJmlzc3VlTmFtZT12ZXJza W 9ub25lLTEydGgtY W 5ud W FsLXN0YXRlL W 9mL W Fna W xlLXJlcG9ydCZjb W Q9ZCZza W c9YTFlY2RlYzYxYzAxNDFkZDVmMzc4MWQ2YzYxYjYxYTc%253D, zuletzt besucht am 10.05.2018

2.5 Versionskontrolle

Versionskontrollprogramme sind Programme, welche Entwicklern helfen, Veränderungen am Quelltext über eine bestimmte Zeit hinweg zu verwalten. Dieser Vorgang wird auch als **Versionskontrolle** bezeichnet.[12]

Ein **Commit** stellt eine Veränderung des Quelltextes durch einen Entwickler dar. Der Commit repräsentiert oft eine Veränderung, die sich auf einen spezifischen Teil der Software bezieht (vgl. (Nagele, o.D.)). So können beispielsweise das Hinzufügen einer Funktion, das Beheben eines Bugs oder die Lokalisierung einer App in eine andere Sprache in jeweils einem Commit festgehalten werden.

Wenn mehrere Entwickler an einem Programm zusammenarbeiten, wird die Versionskontrolle in der Regel über einen Server betrieben. Um zu garantieren, dass die aktuelle Version des Quelltextes auf diesem Server liegt, muss man Commits auf diesem Server hochladen. Dieser Vorgang wird als **Push** bezeichnet.[13] Ein typisches Beispiel für diese Art von Server ist Bitbucket[14].

Des Weiteren wird in einigen Versionskontrollprogrammen erlaubt, verschiedene *Branches* anzulegen. Ein **Branch** ist hierbei eine Folge von Commits. Verschiedene Branches können sowohl gleiche, als auch unterschiedliche Commits beinhalten. Der Branch, welcher den aktuellen Stand der Software repräsentiert, wird in dieser Arbeit als **Master Branch** bezeichnet.

Fügt man die Commitfolgen zweier Branches zusammen, wird dieser Vorgang als **Merge** bezeichnet (vgl. (Nagele, o.D.)). Dabei kann es zu verschiedenen Problemen kommen, den sogenannten **Merge Konflikten**. Für diese Arbeit ist es nur wichtig zu wissen, dass diese Merge Konflikte selten automatisch gelöst werden können und deshalb manuell von einem Entwickler behoben werden müssen.

Nachdem nun die wichtigsten Begriffe der Versionskontrolle genauer erläutert wurden, folgt nun die darauf aufbauende *kontinuierlichen Softwareentwicklung*.

2.6 Kontinuierliche Softwareentwicklung

Wie bereits in Abschnitt 2.4 geschrieben wurde, ist die **kontinuierliche Softwareentwicklung (KS)** ein Oberbegriff für eine Reihe von Praktiken, an deren Basis es steht, dass Entwickler häufig (oft mehrfach pro Tag) ihren geschriebenen Quelltext in ein Softwareprojekt pushen.[9] Je nachdem, was mit dem neuen Quelltext passiert, werden drei Kategorien der kontinuierlichen Softwareentwicklung unterschieden. Diese werden nun vorgestellt.

2.6.1 Kontinuierliche Integration

Bei der **kontinuierlichen Integration** (engl. Continuous Integration, kurz CI) wird der hochgeladenen Quelltext automatisch auf seine Richtigkeit verifiziert. Dies geschieht durch die Umwandung des Quelltextes in ein ausführbares Programm (**Bauen/Build**) und der anschließenden Ausführung der vorhandenen Testumgebung (vgl. (Fowler, 2006)).

Falls es beim Bauen oder Testen der Software zu einem Fehler kommen sollte, bietet der CI-Server dem Entwickler die Möglichkeit, diesen Fehler einzusehen. Zum Aufbau einer CI-Umgebung werden *Trigger*, *Steps* und *Pipelines* genutzt. Ein **Trigger** stellt eine bestimmte

[12] vgl. https://www.atlassian.com/git/tutorials/what-is-version-control, besucht am 08.05.2018

[13] vgl. http://docs.telerik.com/platform/appbuilder/development-tools/version-control/third-party-vc/push-changes, zuletzt besucht am 10.05.2018

[14] https://bitbucket.org/, zuletzt besucht am 10.05.2018

Bedingung dar, welche erfüllt sein kann oder nicht. So kann das Pushen von neuem Quelltext, das Beginnen einer bestimmten Tagesstunde oder auch das Drücken eines Buttons durch einen Entwickler einen solchen Trigger darstellen (vgl. (Fischer, 2018)).

Ein **Step** stellt eine bestimmte Aufgabe dar, die der CI-Server jedes Mal ausführen soll, wenn ein bestimmter Trigger erfüllt ist. Ein Step kann hierbei *gescheitert*, *instabil* oder *stabil* sein. Wenn ein Step **gescheitert** ist, dann gab es Probleme bei der Ausführung der Aufgabe. So konnte beispielsweise ein benötigtes Programm nicht gefunden werden. Wenn ein Step **instabil** ist, so ist er nicht gescheitert, aber die Aufgabe konnte nicht erfolgreich durchgeführt werden. Dies passiert zum Beispiel, wenn die Ausführung von Tests vollzogen werden konnte, aber nicht alle Tests erfolgreich waren. Zuletzt ist ein Step **stabil**, wenn er nicht gescheitert und nicht instabil ist. Die Ausführung von Tests, welche alle erfolgreich durchlaufen wurden, stellt ein Beispiel für einen stabilen Step dar.[15] Der aktuelle Zustand eines Steps kann sich durch die erneute Ausführung von diesem ändern.

Eine *Pipeline* ist eine Menge von Steps.[16] So kann ein CI-Server aus mehreren Pipelines bestehen, welche durch einen gewissen Trigger gestartet werden. Zur Veranschaulichung wird nun die Nutzung von CI unter der Entwicklung einer beispielhaften Software genauer betrachtet. Der CI-Server baut hier zunächst die Software. Nun führt er die Tests aus und überprüft, ob alle Tests erfolgreich ausgeführt worden sind. Falls es bei irgendeinem der Schritte zu einem Fehler kommt, zeigt der CI-Server dies an.
Eine Pipeline kann die Zustände *gescheitert*, *instabil* oder *stabil* annehmen. Sobald mindestens ein Step der Pipeline gescheitert ist, ist eine Pipeline **gescheitert**. Eine Pipeline ist **instabil**, wenn kein Step der Pipeline gescheitert ist, aber mindestens ein Step der Pipeline instabil ist. Wenn kein Step innerhalb der Pipeline gescheitert oder instabil ist, dann bezeichnet man eine Pipeline als **stabil**. Ähnlich wie bei den Steps kann sich der aktuelle Zustand einer Pipeline durch die erneute Ausführung dieser Pipeline ändern.[17]

2.6.2 Kontinuierliche Auslieferung

Die **kontinuierlichen Auslieferung** (engl. Continuous Delivery, kurz CDel) stellt eine Unterkategorie der CI dar. Es müssen alle Aufgaben der CI erfüllt und zusätzlich überprüft werden, ob der aktuelle Stand des Master Branches alle Kriterien erfüllt, um veröffentlicht werden zu können (vgl. (Fowler, 2013)). Die Software wird dabei allerdings nicht automatisch veröffentlicht (vgl. (Pauw, 2015)). Es wird noch ein manueller Eingriff eines Entwicklers benötigt, um die Veröffentlichung zu realisieren.

2.6.3 Kontinuierliche Bereitstellung

Die **kontinuierliche Bereitstellung** (engl. Continuous Deployment, kurz CDep) stellt eine Unterkategorie der CDel dar. Nachdem alle Aufgaben der CDel durchgeführt wurden, wird am Ende die Software automatisch veröffentlicht. Eine wichtige Schlussfolgerung daraus ist, dass ein CDep-Vorgang ohne manuelle Tests auskommen muss (vgl. (Pauw, 2015)). Als Beispiel schreibt (Friedenberg, 2017), dass Etsy[18] bis zu 50 Mal am Tag eine neue Version ihrer Webseite veröffentlicht.

Nachdem nun die wichtigsten Grundlagen für das Verständnis der Arbeit vorgestellt wurden, wird abschließend die Nutzung von KS unter Zuhilfenahme der anderen präsentierten Technologien anhand zweier Fallbeispiele genauer erläutert.

[15] vgl. https://jenkins.io/doc/book/pipeline/syntax/#scripted-steps, zuletzt besucht am 05.06.2018
[16] https://devops.com/continuous-delivery-pipeline/, zuletzt besucht am 10.05.2018
[17] vgl. https://wiki.jenkins.io/display/JENKINS/Terminology, zuletzt besucht am 26.05.2018
[18] https://www.etsy.com/de/, zuletzt besucht am 10.05.2018

2.7 Nutzung der kontinuierlichen Softwareentwicklung bei Apps

In diesem Abschnitt werden die beiden Firmen Etsy und Facebook vorgestellt, welche beide eine Form der KS nutzen, um die Programmierung ihrer App zu unterstützen.

2.7.1 Facebook

In (Rossi et al., 2016) wird beschrieben, wie der Veröffentlichungs-Zyklus von Facebooks Android- und iPhone-Apps aussieht. In dieser Bachelorarbeit wird nur auf den Android-App Zyklus eingegangen. Im Folgenden wird das Wort Facebook für den Teil der Mitarbeiter von Facebook genutzt, welcher sich um die Bereitstellung der Android-Apps kümmert.

Es wird beschrieben, dass Facebook soweit wie möglich CDep einsetzen möchte, um ihre App zu veröffentlichen. Jedoch ist dies nicht vollständig möglich, da beispielsweise manuelle Tests vonnöten ist, um eine App vollständig zu testen. Jedoch wird auch geschrieben, dass man das kontinuierliche Veröffentlichen der App auf dem Master Branch simuliert, ohne den Stand des Master Branches bei jedem Commit tatsächlich zu veröffentlichen. Tests werden auf echten Android-Geräten in einer eigenen Device Cloud ausgeführt. Es wird explizit geschrieben, dass dies einem IaaS Dienst gleichkommt. Zusätzlich werden auch Emulatoren eingesetzt.

Somit nutzt Facebook ein CDel-System, bei der noch einige Eingriffe von Entwicklern nötig sind, bevor eine neue Version der App veröffentlicht wird. Diese Eingriffe werden später genauer beschrieben.

Im Folgenden werden die wichtigsten Schritte vorgestellt, die laut (Rossi et al., 2016) von Facebook zur Entwicklung ihrer App genutzt werden.

Zunächst wird von der „Pre-Push Testing“ Phase gesprochen. Bevor der Entwickler seine Commits auf den Server pusht, führt er regelmäßig Unit-Tests aus. Wie in Abschnitt 2.1.1 gesagt wurde, ist die Ausführung von Unit-Tests schnell erledigt, so dass der Entwickler nicht zu lange aufgehalten wird.

Falls der Entwickler seine Commits auf den Master Branch pushen möchte, ist vorher noch eine *Code Review* nötig. Bei einer **Code Review** überprüft ein anderer Programmierer den Code der Commits auf Mängel.[19] Wenn auch dieser Programmierer davon überzeugt ist, dass der Code keine Mängel mehr aufweist, darf dieser in den Master gepusht werden.

Sobald der Push beginnt, beginnt auch die damit verbundene „On Push“-Phase. Bevor die Commits tatsächlich dem Master Branch hinzugefügt werden, werden eine Reihe von Tests auf den neuen Commits ausgeführt. Sollte dabei keiner fehlschlagen und zusätzlich kein Merge Konflikt bestehen, werden die Commits automatisch auf den Master Branch gepusht. Bei auftretenden Fehlern wird der Entwickler benachrichtigt. Es wird geschrieben, dass dabei nicht alle vorhandenen Tests ausgeführt werden, da dies laut eigenen Aussagen zu zeitaufwändig ist. Alle paar Stunden werden auf dem Master und *Release Branch* alle verfügbaren Tests ausgeführt.

Nun wird ein Veröffentlichungszyklus bei Facebook genauer betrachtet. Jede Woche wird der aktuelle Stand des Master Branches in einen **Release Branch** kopiert, welcher die nächste zu veröffentlichende Version der App darstellt. In der kommenden Woche beschäftigt sich das „Release-Engineering Team“ damit, die noch vorhandenen Bugs in der neuen App-Version zu definieren und zu entscheiden, welche dieser Bugs so kritisch sind, dass sie bis zur Veröffentlichung behoben werden müssen. Dazu kann das Release-Engineering Team auf eine Vielzahl von Hilfsmitteln, wie ein manuelles Testteam oder Metriken über Abstürze der App, zurückgreifen. Außerdem nutzen die Mitarbeiter bei Facebook die

[19] vgl. https://sdqweb.ipd.kit.edu/wiki/Codereview, zuletzt besucht am 10.05.2018

neue Version einer App im Alltag, um weitere Fehler zu finden. Sobald die kritischen Bugs behoben sind, kann die neue Version veröffentlicht werden.

Jedoch wird auch davon berichtet, welche Probleme die KS bei der Facebook-App erzeugt. Während Probleme wie die Fragmentierung der Android-Geräte schon in Abschnitt 2.1.3 genauer untersucht worden sind, soll es nun um die Fragmentierung der Android-App Versionen und das Fehlen einer *Rollback*-Funktion gehen. Ein sehr großes Problem bei der Einrichtung von CDep bei der App-Entwicklung ist, dass eine App nicht automatisch aktualisiert wird, wenn eine neue Version im Google Play Store verfügbar ist. Der Nutzer der App entscheidet selber, ob und wann er die App updaten möchte. Dazu soll das Beispiel aus Abschnitt 2.6.3 zur Veranschaulichung dienen, in welchem Etsy bis zu 50 Mal am Tag eine neue Version ihrer Webseite veröffentlicht. Sollte Facebook, ähnlich wie in diesem Beispiel, jeden Tag mehrere Versionen ihrer App veröffentlichen, kann es schwer werden, jede einzelne App-Version ohne Probleme zu unterstützen. Man müsste Buch darüber führen, welche App-Versionen noch welche Bugs beinhalten, wie die Tabellen der Datenbank aufgebaut sind oder ob neue Funktionen der App in dieser Version schon unterstützt werden oder nicht.

Bei einem Cloud Dienst ist dies nicht so, da der angebotene Dienst zentral auf einem Server läuft. Ein Update der Software in der Cloud aktualisiert somit den Dienst für alle Nutzer gleichzeitig. Im Gegensatz dazu ist eine App auf dem Android-Gerät selbst installiert, was eine zentrale Aktualisierung nicht möglich macht. Es lohnt sich, eine App vergleichsweise selten zu aktualisieren, um die Menge von genutzten App-Versionen klein zu halten.

Des Weiteren fehlt bei der Aktualisierung von Android-Apps eine *Rollback*-Funktion. Dabei bezeichnet **Rollback** die Möglichkeit, die aktuelle Version einer Software mit wenig Aufwand zurückzunehmen. Als Folge dessen wird die nächstälteste Version der Software als aktuelle Version ausgezeichnet.

Zur Veranschaulichung soll noch einmal das Beispiel angeführt werden, dass Etsy ihre Webseite etwa 50 Mal pro Tag neu veröffentlicht. Sollte in der aktuellen Version der Webseite eine Funktion eingeführt worden sein, welche eine Sicherheitslücke aufweist, kann man diese Funktion schnell durch einen Rollback entfernen. Nun kann man in aller Ruhe eine neue Version der Webseite entwickeln, welche diese Sicherheitslücke nicht mehr aufweist.

Im Google Play Store fehlt eine solche Rollback-Funktion. Kritische Sicherheitslücken können nicht so schnell aus der App entfernt werden, wie es nötig wäre. Man müsste eine neue Version der App erstellen, welche unter anderem eine neue Versionsnummer besitzt. Diese müsste vor der Veröffentlichung vom CDep-System getestet werden. Somit kann es sehr lange dauern, bis der Fehler beseitigt wurde. Außerdem muss ein Nutzer nicht zwingend seine Android-App auf die neuste Version ohne Sicherheitslücke aktualisieren, sondern kann die fehlerhafte Version weiterhin nutzen.

Um das Problem der fehlenden Rollback-Funktion zu lösen, nutzt Facebook laut (Rossi et al., 2016) sogenannte **Feature Flags**, welche von einem Server gesetzt werden können. So kann ohne Aktualisierung der App eine fehlerhafte Funktion für alle Nutzer deaktiviert werden. Jedoch muss man hierbei schon vor der Veröffentlichung an den Einbau der Feature Flags denken. Sollte man diese vergessen haben, muss man dennoch eine neue Version der App veröffentlichen.

Zusammenfassend kann man für Facebook sagen, dass sie einen Großteil ihrer Testumgebung automatisiert haben. Jedoch sind trotz alledem noch manuelle Tests nötig und der Release einer App muss über eine Woche hinweg von einem Release-Engineering Team vorbereitet werden.

2.7.2 Etsy (2014)

Bei der Untersuchung der Nutzung von KS bei Etsy fällt auf, dass Etsy ihren Ansatz für KS zwischen 2014 und 2017 stark gewandelt haben. Der in (Kammah, 2014) beschriebene Ansatz von 2014 ist dem von Facebook sehr ähnlich, unterschiedet sich jedoch auch in einigen Punkten, welche nun vorgestellt werden.

Auch bei diesem Beispiel ist erkennbar, dass es sich nicht um ein CDep-System handelt, da immer noch manuelle Tests vollzogen wurden. Es kann nur gemutmaßt werden, ob es sich um ein CDel- oder CI-System handelte. So wird beispielsweise nicht geschrieben, ob, wie bei Facebook, die Auslieferung der App auf dem Master Branch simuliert wurde oder nicht.

Fortführend wird in (Friedenberg, 2017) geschrieben, dass es vor der Umstellung des KS Systems in 2017 Release-Manager gab. Diese hatten ähnliche Aufgaben wie das Release-Engineering Team bei Facebook.

Außerdem versuchte Etsy, wie auch Facebook, eine eigene Device Cloud aufzubauen. Dies war allerdings laut Etsy ein „logistischer Albtraum" (Kammah, 2014). Aus diesem Grund wurde die Device Cloud des Anbieters AppThwack genutzt, um den eigenen Aufwand zu verkleinern. Es sei angemerkt, dass AppThwack später von Amazon Web Services übernommen wurde.[20]

Des Weiteren gab es auch bei Etsy verschiedene Phasen, wann welche Tests ausgeführt wurden. So wurde auch bei Etsy auf Code Reviews gesetzt, bevor Commits auf den Master Branch gepusht werden durften. Danach wurde der neue Code auf Emulatoren getestet, welche auf internen Servern liefen. Des Weiteren wurde der Push auf einigen wenigen Geräten der Device Cloud getestet. In der Nacht wurde der aktuelle Quelltext auf mehr als 200 Android-Geräten in der Device Cloud getestet. Um die App auf echten Geräten zu testen, standen des Weiteren Android-Geräte zum manuellen Testen für die Entwickler zur Verfügung. So wird geschrieben, dass täglich eine neue Version der Etsy App an alle Mitarbeiter verteilt wurde, welche das manuelle Testen übernahmen. Ein separates, manuelles Testteam wie bei Facebook gab es nicht.

Bei Etsy gab es auch das Problem, dass zu viele Android-App Versionen gleichzeitig genutzt wurden. Es wird von zehn gleichzeitig genutzten App-Versionen geschrieben.

Jedoch hat sich laut (Friedenberg, 2017) viel bei Etsy bezüglich der KS geändert. Auf den neuen Ansatz soll nun eingegangen werden.

2.7.3 Etsy (2017)

In (Friedenberg, 2017) wird zunächst erwähnt, dass es vor der Umstellung des KS-Systems zwei Release-Manager gab, welche ähnliche Aufgaben wie das Release-Engineering Team bei Facebook hatten. Jedoch bestand das Team nicht, wie bei Facebook, aus etwa zehn (vgl. (Rossi et al., 2016)), sondern nur aus zwei Personen. Einer der beiden verließ Etsy später, sodass der komplette Release von einer einzigen Person koordiniert werden musste. Jedoch wollte man bei Etsy vermeiden, dass ein einziger Release-Manager den kompletten Release einer App anleiten muss und der alleinige Entscheider bei wichtigen Entscheidungen war. Es zeigt sich, dass ein Ansatz mit Release-Managern nur dann funktioniert, wenn genügend Personen für diese Rolle eingesetzt werden.

Aus diesem Grund hat sich Etsy entschieden, ein Tool names Ship zu entwickeln. Ship ist hierbei als zentrale Anlaufstelle zur Verwaltung der App-Versionen gedacht.

Im Folgenden wird die Veröffentlichung einer neuen App-Version beispielhaft vorgestellt. Dabei wird auf dem Beispiel aus (Friedenberg, 2017) aufgebaut.

[20] https://www.crunchbase.com/organization/appthwack#section-overview, zuletzt besucht am 10.05.2018

Für jeden neue Release-Zyklus, welcher ungefähr drei Wochen dauert, wird zufällig einer der Entwickler als „Driver“ ausgesucht. Dieser kümmert sich um den reibungsfreien Ablauf des nächsten Releases und übernimmt somit die Rolle des Release-Managers. Der entscheidende Unterschied zwischen diesem und dem alten KS-System ist, dass die Entwickler selber bestätigen müssen, dass ihr Quelltext ordnungsgemäß funktioniert. Wenn ein Entwickler einen Commit auf den Master Branch pusht, bekommt er automatisch eine neu gebaute APK[21] zugeschickt. Diese muss er ausführlich manuell testen und per E-Mail bestätigen, dass er sicher ist, dass keine neuen Fehler vorhanden sind. Diese Phase dauert insgesamt zwei Wochen.

In der nun folgenden „finalen Test-Phase“ muss sich der Driver darum kümmern, dass alle noch vorhandenen kritischen Fehler aus der App entfernt werden. Sollte kein Fehler mehr gefunden werden, wird die neue App-Version automatisch veröffentlicht.

Obwohl Etsy nun tatsächlich ihre App automatisch veröffentlicht, sind die Kriterien für CDep dennoch nicht erfüllt. Es wird nicht jeder neue Stand des Master Branches gebaut und veröffentlicht. Außerdem wird viel manuell getestet. Es konnte jedoch nicht herausgefunden werden, wie bei Etsy aktuell die Umgebung aussieht, in der automatisierte Tests ausgeführt werden. Aus (Campbell, 2018) geht jedoch hervor, dass automatisierte Tests vorliegen, da laut dieser Quelle in einem CI-System die Testresultate überprüft werden. Des Weiteren wird in dieser Quelle auch davon berichtet, dass das KS-Programm *Jenkins*[22] genutzt wird.

Im folgenden Kapitel wird nun der aktuelle Stand des AMCS-Systems dargestellt. Dabei wird unter anderem darauf eingegangen, welche Form der KS zurzeit genutzt wird und welche Probleme dabei auftreten.

[21] Eine APK stellt das Installationspaket einer Android App dar.
[22] https://jenkins.io/, zuletzt besucht am 06.06.2018

3 Analyse des aktuellen Standes

In diesem Kapitel wird zunächst das AMCS-Projekt vorgestellt und welches Ziel es verfolgt. Im Weiteren wird der aktuelle Stand der AMCS Android-App betrachtet.

3.1 Das AMCS-Projekt

Das **Auditorium Mobile Classroom Service (AMCS)**[3] stellt ein Softwareprojekt zwischen Mitarbeitern der Fakultäten Psychologie und Informatik der Technischen Universität (**TU**) Dresden dar. In diesem Projekt möchte man das Problem lösen, dass Studenten in Vorlesungen oft nach einer Stunde die Aufmerksamkeit verlieren und dem Dozenten nicht mehr aktiv zuhören können.

Der verfolgte Lösungsansatz besteht darin, dass man Vorlesungen durch die Nutzung von mobilen Geräten interaktiver macht. Die Software, welche im Rahmen des AMCS-Projekts entwickelt wird, bietet hierbei die Grundlage zum Austausch von Informationen zwischen dem Dozenten und den Studenten im Vorlesungssaal. Der Dozent kann so beispielsweise Aufgaben zum Stoff der Vorlesung an die Studenten stellen, welche die Studenten über ein mobiles Gerät beantworten können und Feedback erhalten. Des Weiteren bekommt der Dozent eine Übersicht, wie viele Studenten die Fragen richtig beantworten konnten. Diese Informationen kann der Dozent beispielsweise nutzen, um die Geschwindigkeit seiner Vorlesung anzupassen.[23]

Aus dem aktuellen Stand des AMCS-Projekts leitet sich eine weitere Anforderung ab für die Device Clouds ab. Diese ist, dass die Implementierung dieser keine Kosten verursachen darf (**A5**). Konkret wird dies danach bemessen, dass die untersuchten Device Clouds ein kostenloses Angebot oder eine Probezeit zum Ausprobieren der Device Clouds anbieten, in welcher auch echte Android-Geräte genutzt werden können.

3.2 Die AMCS Android-App

Zunächst muss erwähnt werden, dass es zwei verschiedene AMCS-Apps für Android gibt. Die erste wird zurzeit im Google Play Store angeboten und innerhalb der Universität genutzt. Die zweite AMCS-App wurde im Zuge von (Buchholz, 2017) erstellt. Sie wird zurzeit nicht im Google Play Store angeboten. Jedoch wird letztere App an die Device Clouds im Zuge dieser Arbeit angebunden. Wenn in dieser Arbeit von der AMCS-App geschrieben wird, ist dabei immer die App aus (Buchholz, 2017) gemeint. Soweit nicht anders angegeben, stammen die nun folgenden Informationen aus (Buchholz, 2017).

[23] vgl. https://amcs.website/about, zuletzt besucht am 20.05.2018

3.2.1 Erstellung der Integration Tests

Für die Erstellung von Unit Tests wird das Testframework JUnit 4[24] genutzt. Da es im Rahmen dieser Bachelorarbeit jedoch um die Ausführung von Integration Tests geht, wird dieser Teil der App nicht genauer betrachtet.

Für die Erstellung von Integration Tests wird das Testframework Espresso[25] genutzt. Mit Hilfe dieses Frameworks kann man Integration Tests erstellen, welche die graphische Oberfläche der App verändern und kontrollieren können. So kann in einem Espresso-Test beispielsweise zunächst ein Text in ein Textfeld eingegeben, danach ein Button gedrückt und zuletzt überprüft werden, ob eine erwartete Benachrichtigung erscheint.

Aus diesem Grund wird hier eine weitere Anforderung an die Device Clouds definiert. Dabei wird betrachtet, inwieweit die bereits geschrieben Tests der AMCS-App auf den Device Clouds ausgeführt werden können (**A2**). Insbesondere bedeutet dies, dass die Device Cloud in der Lage sein muss, Tests, welche mit Espresso geschrieben wurden, auszuführen.

3.2.2 Aktuelle Versionskontrolle

Es wird zurzeit zur Versionskontrolle der Dienst Bitbucket[14] genutzt. Des Weiteren wird eine Strategie für die Nutzung der Versionsverwaltung mit dem Namen *Gitflow Workflow*[26] verwendet.

Im Folgenden wird dieser Workflow so vorgestellt, wie er auch in (Buchholz, 2017) erklärt wurde. Es wird dabei keine vollständige Erläuterung stattfinden. Es werden nur die für diese Arbeit relevanten Punkte aufgezeigt.

Als Erstes werden die Branches der Versionsverwaltung in fünf verschiedene Kategorien eingeteilt. Diese sind Master, Develop, Release, Hotfix sowie Feature Branches. Im Rahmen dieser Arbeit werden die ersten vier Kategorien unter dem Namen Master Branch zusammengefasst, da sie bei der Nutzung im KS-System alle gleich behandelt werden und nur bei der eigentlichen Entwicklung der App von Bedeutung sind. Dies wird im nächsten Abschnitt genauer gezeigt.

Auf dem Master Branch werden lauffähige Versionen der App gespeichert. Neue Funktionen der App werden in sogenannten **Feature Branches** erstellt. Sobald eine Funktion fertig implementiert ist, findet ein Merge des entsprechenden Feature Branches in den Master Branch statt.

3.2.3 Kontinuierliche Softwareentwicklung der AMCS App

Es wird bereits ein Server der TU Dresden genutzt, welcher die kontinuierliche Softwareentwicklung der App ermöglicht. Dieser Server stellt eine Jenkins-Instanz bereit, mit deren Hilfe KS durchgeführt werden kann. Genauere Informationen sind im Abschnitt 3.4 zusammengefasst. Der Teil der Jenkins-Instanz, welcher sich mit der KS der Android App befasst, wird im Weiteren **Jenkins-Server** oder **AMCS-System** genannt. Das Programm Jenkins kann den Zustand einer Pipeline durch Farben anzeigen, was es einfacher für den Entwickler macht, ihren Zustand zu beurteilen. Dabei steht Grün für einen stabile, Gelb für eine instabile und Rot für eine gescheiterte Pipeline.

[24] https://junit.org/junit4/, zuletzt besucht am 06.06.2018
[25] https://developer.android.com/training/testing/espresso/, zuletzt besucht am 06.06.2018
[26] https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow, zuletzt besucht am 02.06.2018

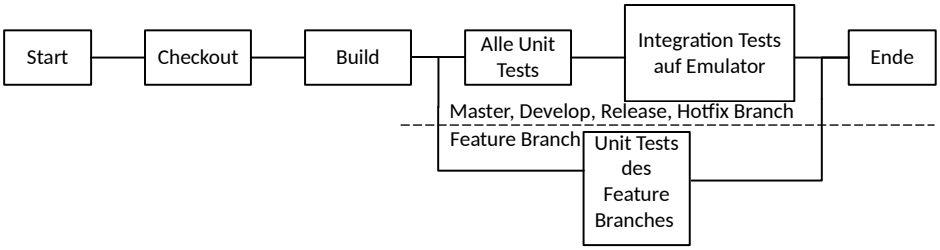


Abbildung 3.1: Testausführungsablauf in Jenkins nach (Buchholz, 2017)

In der Arbeit (Buchholz, 2017) wird eine ähnliche Abbildung wie die Abbildung 3.1 genutzt, um den Aufbau der Pipeline darzustellen. Der Jenkins-Server prüft aller 15 Minuten, ob auf dem genutzten Bitbucket-Server neue Commits vorhanden sind. Bitbucket selbst kann den Jenkins-Server nicht darüber benachrichtigen, da der Jenkins-Server nur innerhalb des Universitätsnetzwerkes angesprochen werden kann. Die vorliegende Pipeline in der oberen Abbildung ist in **Stages** eingeteilt, welche eine Menge von zusammengehörige Steps darstellen.[27]

Dabei wird, wie im oberen Abschnitt beschrieben, vereinfacht zwischen dem Master und Feature Branches unterschieden. Zunächst wird bei der Ausführung der entsprechenden Pipeline in der Checkout Stage der aktuelle Stand des Branches von dem Bitbucket Server heruntergeladen. Als nächstes wird die App in der Build Stage gebaut. Dabei wird mit Hilfe des Build-Programms Gradle[28] aus dem aktuellen Quelltext eine *APK*-Datei gebaut.

Nun unterscheidet sich die Ausführung der Pipeline nach dem Namen des Branches, der gerade betrachtet wird. Sollte es sich um den Master Branch handeln, so werden in den folgenden beiden Stages zunächst alle Unit Tests und danach alle Integration Tests des Projekts ausgeführt. Auf die Ausführung der Integration Tests wird im nächsten Abschnitt noch einmal genauer eingegangen, da diese Stage im Rahmen dieser Arbeit angepasst wird.

Sollte es sich jedoch um einen Feature Branch handeln, dann werden die beiden zuletzt genannten Stages nicht ausgeführt. Es werden nur die entsprechenden Unit Tests des neuen Features ausgeführt.

Zuletzt werden bei allen Branches die Ergebnisse der Tests für die Nutzer des Jenkins-Servers aufbereitet und ausgegeben.

Die Nutzung verschiedener Stages bei den Pipelines wurde eingesetzt, da eine Änderung auf einem Feature Branches laut (Buchholz, 2017) viel öfter stattfindet als auf dem Master Branch. Somit kann durch die Verteilung der Tests die Ausführungszeit minimiert werden, um dem Entwickler möglichst schnell Testresultate zu übermitteln.

Es handelt sich somit im Falle des AMCS-Systems um einen CI-Server. Es wird der aktuelle Quelltext gebaut und die Tests der App ausgeführt. Danach wird nicht versucht, die App darauf zu überprüfen, ob sie veröffentlichbar ist. Somit handelt es sich nicht um ein CDel- oder CDep-System.

Eine abzuleitende Anforderung an die Device Clouds während der Implementierung besteht darin, dass sie möglichst einfach in das vorhandene AMCS-System einzubinden sind (**A4**). Dies bedeutet konkret, dass es eine öffentlich einsehbare Dokumentation zur Verbindung eines Jenkins-Servers mit der Device Cloud gibt. Diese Anforderung ist sehr wichtig, da termingebundene Firmen schnell auf Probleme mit ihrer genutzten Infrastruktur eingehen können müssen. Um Strafzahlungen zu vermeiden, muss die Anbindung der Device

[27] vgl. https://jenkins.io/doc/book/pipeline/#stage, zuletzt besucht am 02.06.2018
[28] https://gradle.org/, zuletzt besucht am 20.05.2018

Cloud im Falle eines Problems dokumentiert sein, um Abgabetermine nicht verschieben zu müssen.

3.2.4 Ausführungsweisen von Jenkins

Zur Zeit werden zwei verschiedene Ansätze von Jenkins innerhalb des TU-Jenkins genutzt, um Pipelines zu entwickeln.

Pipeline-Projekt[29]

Jenkins bietet zur Ausführung von Pipelines das sogenannte Pipeline-Plugin[30] an. Mit Hilfe dieses Plugins können die Steps einer Pipeline in sogenannte *Jenkinsfiles* gespeichert werden. Ein **Jenkinsfile**[29] stellt eine Konfigurationsdatei für Jenkins dar, welche für jeden Branch des Softwareprojekts erstellt wird. Sobald eine Änderung des aktuellen Standes eines Branches von Jenkins festgestellt wird, führt dieser den Inhalt des Jenkinsfiles aus. Im Folgenden werden die wichtigsten Inhalte eines Jenkinsfiles betrachtet, welche auch in Kapitel 5 zur Implementierung der Device Clouds genutzt werden.

Ein Jenkinsfile definiert den Ablauf einer Pipeline. Diese wird zunächst in einzelne *Stages* untergliedert. **Stages** stellen logisch zusammenhängende Steps dar.[27] Ein Beispiel für eine Stage stellt die Ausführung aller benötigten Steps dar, um Integration Tests auf der Device Cloud auszuführen.

Ein Step stellt, wie schon in Abschnitt 2.6 erläutert, die Ausführung eines Befehls dar. Im Zuge dieser Arbeit werden zwei verschiedene Arten von Steps genutzt. Der erste ist der **sh-Step**, welcher mit dem Schlüsselwort `sh` beginnt und zur Ausführung von Shell-Skripten dient. Ein **Shell-Skript** stellt hierbei ein Programm dar, welches in einer Konsole von unixartigen Betriebssystemen ausgeführt werden kann (vgl. (Robbins & Beebe, 2006)).

Shell-Skripte werden im Verlauf dieser Arbeit verwendet, um die Anbindung von Device Clouds zu ermöglichen. In diesen Skripten kann man eine Menge von Konsolenbefehlen speichern. An den sh-Step wird ein konkreter Konsolenbefehl übergeben, welcher ausgeführt werden soll.

Eine zweite wichtige Art von Step ist der *build-Step*, welcher zur Ausführung von sogenannten *Freestyle-Projekten* dient. Auf diese wird im nächsten Abschnitt genauer eingegangen.

Freestyle-Projekte

Ein Freestyle-Projekt stellt einen Kontrast zu Pipeline-Projekten dar. Während in einem Pipeline-Projekt die einzelnen Steps in einer Konfigurationsdatei festgehalten werden, werden die Steps bei einem **Freestyle-Projekt** über die graphische Oberfläche von Jenkins konfiguriert bzw. hinzugefügt. Der Grund, warum Freestyle-Projekte auf dem Jenkins-Server genutzt werden, ist die Kompatibilität mit einigen Plugins für Jenkins. So ist es beispielsweise noch nicht möglich, mit Hilfe des Android Emulator Plugins, einen Android-Emulator über ein Jenkinsfile zu konfigurieren. Um dies zu ermöglichen, muss ein neues Freestyle-Projekt in Jenkins eingerichtet werden, in welchem der Emulator über die graphische Oberfläche[31] von Jenkins konfiguriert wird.

[29] https://jenkins.io/solutions/pipeline/, zuletzt besucht am 02.06.2018
[30] https://wiki.jenkins-ci.org/display/JENKINS/Pipeline+Plugin, zuletzt besucht am 02.06.2018
[31] https://wiki.jenkins.io/display/JENKINS/Android+Emulator+Plugin, zuletzt besucht am 02.06.2018

Da jedoch ein Hauptteil der CI für die AMCS-App innerhalb von Jenkins über ein Pipeline-Projekt gehandhabt wird, sollte ein Freestyle-Projekt innerhalb eines Pipeline-Projekts gestartet werden können. Dafür kann der im letzten Abschnitt erwähnte build-Step in einem Jenkinsfile genutzt werden. Dem build-Step wird hierbei der Name eines Freestyle-Projektes in Jenkins übergeben. Sobald dieser Schritt innerhalb der Pipeline ausgeführt wird, wird nach einem entsprechenden Projekt mit diesem Namen gesucht und dieses gestartet. In Kapitel 5 werden Device Clouds über ein Freestyle-Projekt in die Pipeline für die AMCS-App eingebunden.

3.3 Ausführung der Integration Tests

Es wird zurzeit mit Hilfe des Jenkins-Plugins Android Emulator Plugin[31] ein Android-Emulator erstellt, auf dem die Integration Tests ausgeführt werden. Hierbei wird ein Android-Gerät emuliert, welches die OS-Version 5.0, die Sprache Deutsch und die **Target ABI** armeabi-v7a nutzt. Letzteres bedeutet, dass ein simulierter ARM-Prozessor für den Android-Emulator genutzt wird. Nähere Informationen zur Emulation des Prozessors können den Leitfäden[32] zur Android-Entwicklung entnommen werden.

Eine Beschleunigung des Android-Emulators durch eine **Kernel-based Virtual Machine (KVM)**[33] ist derzeit nicht möglich. Dies wurde anhand der Anleitung auf der Webseite Android Developers[33] untersucht.

Die in (Buchholz, 2017) beschriebenen Timeouts bei der Ausführung der Tests auf dem Emulator traten während der kompletten Untersuchung im Rahmen dieser Bachelorarbeit mit dem aktuell genutzten Server nicht auf.

Ein Problem, welches innerhalb des AMCS-Systems auftritt, ist die Ausschreibung des Zustandes einer Pipeline. So wird derzeit ein Scheitern von Tests auf dem Emulator als stabiler Zustand der Pipeline ausgegeben, obwohl dieser instabil sein sollte. Nur über eine Auswertung der Datei „logcat.txt“ kann herausgefunden werden, dass ein Test fehlgeschlagen ist.[34] Diese Datei beinhaltet alle Statusinformationen, welche während der Ausführung des Emulators aufgezeichnet wurden. Sie wird im nachfolgenden **Device Log** genannt. Die manuelle Analyse des Device Logs ist jedoch nach der Anforderung A1 nicht wünschenswert, da der Entwickler möglichst automatisiert und schnell erfahren soll, dass ein Test fehlgeschlagen ist.

Die nächste Anforderung an eine Device Cloud, die an dieser Stelle gesetzt werden wird, ist, dass sie nicht auf einer on-premises Cloud basiert (**A3**). Wie bereits im Abschnitt 2.2.1 definiert wurde, sind die Nutzer und Betreiber einer on-premises Cloud die gleichen Personen. Aus diesem Grund müsste man zum Betreiben einer on-premises Device Cloud echte Android-Geräte kaufen, welche man mit dem entsprechenden Server verbindet. In dieser Arbeit wird jedoch der Vorteil von Device Clouds ausgenutzt, dass diese eine solche Infrastruktur bereitstellen und so Kosten vermieden werden können.

Nun soll erläutert werden, welche Schritte nötig sind, um die Integration Tests einer Android-App für die Ausführung auf einem Emulator vorzubereiten. Dies ist wichtig, da diese Schritte auch bei der Nutzung von Device Clouds durchgeführt werden müssen. Dabei wurden die aktuellen Schritte im AMCS-Projekt betrachtet.

In den nun folgenden Schritten wird der „Gradle Wrapper“[35] genutzt, um das Bauen der

[32] https://developer.android.com/ndk/guides/abis#v7a, zuletzt besucht am 30.05.2018

[33] vgl. https://developer.android.com/studio/run/emulator-acceleration#accel-check, zuletzt besucht am 05.06.2018

[34] siehe Jenkins - Android UI Integration #41

[35] https://docs.gradle.org/current/userguide/gradle_wrapper.html, zuletzt besucht am 02.06.2018

App zu koordinieren.

Im ersten Schritt muss eine *APK* gebaut werden. Ein **Application Package Kit (APK)** stellt hierbei das Paketformat dar, in welches der Android-App Quelltext gebaut wird (vgl. (Montegriffo, 2018)). Eine APK stellt somit eine installierbare Android-App dar. Eine solche APK-Datei kann mit Hilfe des Befehls `./gradlew assembleDebug` erzeugt werden.[36] Es wird hierbei eine sogenannte **Debug-APK** erstellt, welche nur innerhalb des Entwicklungsprozesses einer App genutzt und nicht in den Google Play Store hochgeladen werden kann[36].

Als Nächstes muss für die Ausführung der Espresso-Tests zusätzlich noch eine *Test-APK* gebaut werden. Diese **Test-APK**[37] wird zur Ausführung von Integration Tests auf einem Emulator bzw. echten Android-Gerät installiert. Sie wird mit dem Befehl `./gradlew assembleDebugAndroidTest` generiert.

Sowohl die APK als auch die Test-APK werden benötigt, um die Integration Tests auszuführen. Um diese beiden Dateien tatsächlich auf dem Emulator zu starten, sind noch weitere Schritte mit Hilfe der *Android Debug Bridge*[38] notwendig. Diese sind jedoch bereits im AMCS-System integriert und werden auch für die spätere Nutzung der Device Clouds nicht benötigt.

3.4 Rahmenbedingungen

In diesem Abschnitt wird zusammengefasst, welche Rahmenbedingungen zurzeit bei der AMCS-App und dem Jenkins-Server vorzufinden sind.

Der für die Implementierung genutzte Server zur Bereitstellung der kontinuierlichen Softwareentwicklung ist ein von der TU Dresden bereitgestellter Server. Dabei wird eine virtuelle Maschine auf einem vom „Zentrum für Informationsdienste und Hochleistungsrechnen“ bereitgestellten Server betrieben. Die virtuelle Maschine läuft mit dem Betriebssystem Debian 9 „Stretch“ (Stable) [39]. Auf dieser virtuellen Maschine läuft eine Jenkins Instanz in der Version 2.107.3 [40]. Folgende Plugins bzw. Programme, welche in (Buchholz, 2017) benannt wurden, werden zurzeit[41] in den folgenden Versionen genutzt:

- OpenJDK[42] Version 1.8.0_171
- Gradle Plugin 1.28
- BitBucket Plugin 1.1.8
- Android Emulator Plugin 3.0
- Matrix Project Plugin 1.13
- Pipeline Plugin 2.5
- Bitbucket Approve Plugin 1.0.3
- Bitbucket Build Status Notifier 1.3.3
- Green Balls 1.15
- Versionen aus der Datei „build.gradle“ der AMCS-App:

[36] vgl. https://developer.android.com/studio/build/building-cmdline, zuletzt besucht am 02.06.2018
[37] https://developer.android.com/studio/test/, zuletzt besucht am 02.06.2018
[38] https://developer.android.com/studio/command-line/adb, zuletzt besucht am 02.06.2018
[39] https://wiki.debian.org/DebianStretch , zuletzt besucht am 27.05.2018
[40] https://jenkins.io/changelog-stable/ , zuletzt besucht am 27.05.2018
[41] Stand: 07.06.2018
[42] http://openjdk.java.net/, zuletzt besucht am 06.06.2018

- minSdkVersion = 15
- targetSdkVersion = 26
- compileSdkVersion = 26
- buildToolsVersion = '25.0.3'
- espressoVersion = '3.0.1'
- firebaseVersion = '9.6.1'

3.5 Anforderungen

In diesem Abschnitt werden noch einmal die Anforderungen zusammengetragen, nach welchen die Device Clouds in den folgenden Kapiteln beurteilt werden. Die Anforderungen wurden bzw. werden dabei an den entsprechenden Stellen der Kapitel 2, 3 sowie 4 herausgearbeitet und begründet.

Anforderung	**Kurzbeschreibung**
A1	Schnelles Feedback der Testresultate an Entwickler
A2	Unterstützung der vorhandenen Testsoftware und Tests
A3	Keine on-premises Lösung
A4	Einfachheit der Integration in das vorhandene AMCS-System
A5	Kostenlose Nutzung der Device Cloud
A6	Abdeckung der am meisten genutzten Android-Geräte
A7	Geringe Netzwerkauslastung durch Nutzung der Device Cloud

Tabelle 3.1: Anforderungen

4 Konzept

In diesem Kapitel wird ein Konzept davon entwickelt, wie die einzelnen Device Clouds in das vorhandene AMCS-System integriert werden. Dabei wird erläutert, anhand welcher Kriterien und Forderungen die Device Clouds beurteilt werden. Des Weiteren wird im Detail darauf eingegangen, an welcher Stelle der vorhandenen Pipeline die entsprechende Ausführung der Tests mit Hilfe der Device Clouds stattfinden soll. Es wird auch benannt, welche Messdaten für die einzelnen Device Clouds erhoben werden.

4.1 Aufbau der Pipeline

4.1.1 Form der kontinuierlichen Softwareentwicklung

Wie in dem Abschnitt 2.6 erwähnt wurde, gibt es drei Formen der KS. Diese sind CI, CDel und CDep. Diese Arbeit wird sich darauf beschränken, das schon vorhandene CI-System der AMCS App weiter auszubauen. Wie anhand der App-Entwickler Etsy und Facebook in 2.7 gezeigt wurde, ist eine kontinuierliche Bereitstellung von Apps nicht ohne Probleme möglich. Aber auch ein CDel-System aufzubauen ist nicht das Thema dieser Arbeit, da zunächst das CI-System mit Hilfe der Device Clouds ausgeweitet wird.

4.1.2 Stages der Pipeline

Im Folgenden soll gezeigt werden, wie die Pipeline, welche in Abbildung 3.1 vorgestellt wurde, in dieser Arbeit weiter ausgebaut wird.

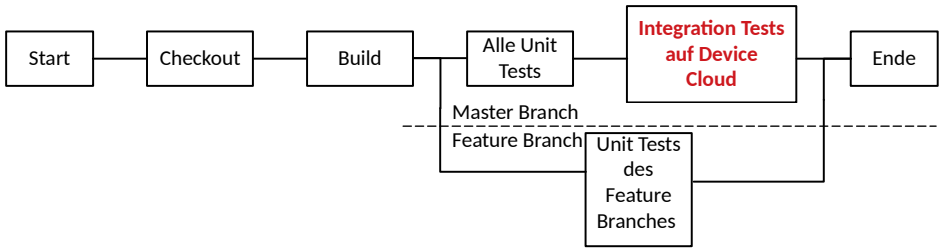


Abbildung 4.1: Konzept des Testausführungsablaufs (Abbildung nach (Buchholz, 2017))

Wie diese Abbildung verdeutlicht, wird eine Stage in der vorhandene Pipeline des AMCS-Systems gegen eine andere ausgetauscht. In dieser werden vorhandene Integration Tests

auf einer Device Cloud ausgeführt. Alle weiteren Stages werden nicht verändert.

4.1.3 Nutzung des Zustands einer Pipeline

Die automatisierte Nutzung des aktuellen Zustands einer Pipeline soll verbessert werden. Insbesondere soll das Pushen von fehlerhaftem Code in den Master Branch unterbunden werden. Um dies zu realisieren, soll der Push von Commits erst dann erlaubt werden, wenn ein fehlerfreier Pipelinedurchlauf dieses Codes vollzogen wurde. Falls währenddessen ein Fehler geworfen wird, soll dies durch einen automatischen Trigger an den Entwickler gemeldet und der Merge gestoppt werden. Somit kann die Planung der Testausführung vom Server übernommen und der Entwickler entlastet werden.

4.2 Vorauswahl von Smartphones für Tests in der Device Cloud

Wie bereits in Kapitel 2 beschrieben wurde, werden viele Cloud Dienste mit dem Bezahlmodell On-Demand angeboten. Eine Nutzung aller bereitgestellten Android-Geräte in einer Device Cloud ist somit nicht praktikabel. Es würden zu viele Kosten entstehen, welche den geringen Preis von Device Clouds gegenüber einer eigenen Lösung wieder revidieren würden.

Aus diesem Grund wird sich diese Arbeit auch mit der Frage beschäftigen, anhand welcher Kriterien Android-Geräte ausgesucht werden sollten, um möglichst viele Fehler bei den Nutzern der App zu vermeiden. Als Ergebnis soll eine Menge von Android-Geräten definiert werden, welche zum Ausführen der Integration Test genutzt werden wird. Diese Menge soll dabei so gewählt sein, dass die Kosten bei der Nutzung der Device Cloud möglichst klein gehalten werden, aber die Menge an abgedeckten Nutzern möglichst groß ist.

Daraus leitet sich eine weitere Anforderung an die Device Clouds ab. Aufgrund der in Abschnitt 2.1.3 aufgezeigten Fragmentierung bei Android-Geräten ist es nicht möglich, alle Tests auf allen vorhandenen Android-Geräten auszuführen. Device Clouds sollen möglichst viele der am meisten genutzten Android-Geräte zum Ausführen der Tests anbieten (**A6**). Dazu wird überprüft, wie viele der zunächst ausgewählten Android-Geräte als echte Geräte in der Device Cloud angeboten werden.

4.3 Vorauswahl von Device Clouds

Die konkreten Device Clouds, welche prototypisch in das AMCS-System eingebunden werden sollen, werden aus (Braun et al., 2017) entnommen. Dabei wird zunächst anhand der Anforderungen A2 bis A5 mit Hilfe von öffentlich zugänglichen Informationen überprüft, ob die Device Cloud als Prototyp implementiert werden wird.

Dazu wird zunächst überprüft, ob die betrachtete Device Cloud die bereits vorhandenen Tests ausführen kann. Dies soll eine Nutzung im aktuellen CI-System erleichtern. Des Weiteren darf die Device Cloud keine on-premises Lösung darstellen. Außerdem muss eine Dokumentation vorliegen, anhand derer die Device Cloud in das derzeit genutzte CI-System eingebunden werden kann. Zuletzt wird überprüft, ob im Rahmen eines kostenloses Angebotes echte Android-Geräte zur Testausführung bereitgestellt werden.

4.4 Einbindung der Device Clouds

Um die Fehlerquellen zu Beginn der Implementierung möglichst gering zu halten, wird zunächst eine Verbindung zwischen dem lokalen Computer des Autors und der Device Cloud aufgebaut. In diesem Schritt sollen die richtigen Parameter gefunden werden, um die Kommunikation mit dem Device Cloud Anbieter zu ermöglichen.

Sobald eine Verbindung hergestellt werden konnte und eine Testausführung stattfand, wird die Device Cloud mit dem vorhandenen CI-Server verbunden.

Wenn auch dies eingerichtet ist, werden die gesendeten Testberichte der Device Cloud näher betrachtet. Zunächst wird untersucht, welche Auswirkung die Testberichte auf den CI-Server haben. So sollte beispielsweise ein fehlgeschlagener Test zu einem instabilen Zustand der Pipeline führen. Um dieses Kriterium zu testen, wird ein fehlerhaften Test, welcher immer fehlschlägt, bei der Untersuchung zur Testumgebung hinzugefügt. Je nachdem, ob dieser fehlerhafte Test in den aktuell ausgeführten Tests enthalten ist oder nicht, soll sich auch der Zustand der Pipeline verändern.

Des Weiteren soll die vorher definierte Menge an Android-Geräten zur Testausführung in der jeweiligen Device Cloud genutzt werden. Falls weitere Fehlschläge der Tests durch die Nutzung einer Device Cloud gefunden werden, so muss auch dies dem Entwickler gemeldet werden. Im Nachhinein werden im Rahmen dieser Arbeit die gefunden Fehler nicht behoben. Es wird sich damit beschäftigt, die Device Clouds in das aktuelle CI-System einzubinden.

4.5 Vergleich der Implementierungen

Nachdem die ausgewählten Device Clouds mit dem vorhandenen CI-System verbunden wurden, werden die einzelnen Prototypen untereinander weiter verglichen.

Als Erstes wird getestet, wie lange eine Ausführung aller Integration Tests auf der Device Cloud dauert. Dies ist vor allem für die Durchführung der in Kapitel 2 vorgestellten agilen Prinzipien wichtig. Der Entwickler muss kontinuierlich Informationen über seinen Quelltext erhalten, um auf Probleme und Fehler eingehen zu können.

Um diese Anforderung A1 zu überprüfen, wird die Zeit zwischen dem Push eines Commits und der Benachrichtigung, ob dieser Commit in den Master gepusht werden darf, gemessen. Dabei werden die Tests auf der vorher definierten Menge von Android-Geräten für jede Device Cloud ausgeführt. Zum Vergleich wird auch in Betracht gezogen, wie lange die Ausführung der Integration Tests auf der aktuell implementierten Lösung im CI-System dauert. Die Messwerte sollen im Bezug auf den Mittelwert sowie die Standardabweichung untereinander verglichen und beurteilt werden.

Als weitere Anforderung an die Device Clouds wird betrachtet, welche Netzwerkauslastung durch die Kommunikation mit der Device Cloud auftritt (**A7**). Dabei soll überprüft werden, welche Pakete bei der Kommunikation zwischen dem CI-Server und der Device Cloud ausgetauscht werden, welche Länge diese haben und welcher Inhalt in den Paketen steckt. Dies ist von großem Interesse, da nicht jeder Server einen Großteil seiner Bandbreite für die Kommunikation mit der Device Cloud nutzen möchte oder die Internetverbindung nach der Höhe des Datenverbrauchs bezahlt wird.

Um die Bewertung durchzuführen, soll die Kommunikation zwischen den Device Clouds und dem CI-Server überwacht und analysiert werden. Hierfür wird die Menge der meistverwendeten Android-Geräte genutzt.

Nachdem nun das Konzept zur Implementierung und Evaluation beschrieben wurde, wird im folgenden Verlauf der Arbeit dieses Konzept für das AMCS-System umgesetzt.

5 Implementierung

In diesem Kapitel wird die Integration der Device Clouds in das vorhandene AMCS-System ausgeführt. Zunächst wird eine Vorauswahl von Android-Geräten vorgenommen, welche zum Ausführen der Integration Tests auf den Device Clouds genutzt werden wird. Weiterhin werden die in Abbildung 2.1 gezeigten Device Clouds anhand der in Tabelle 3.1 zusammengefassten Anforderungen untersucht. Nachdem eine Vorauswahl der Device Clouds durchgeführt wurde, wird zuletzt die Integration der selektierten Device Clouds stattfinden. Dabei wird insbesondere auf Probleme eingegangen, welche zu einer Abweichung der im Konzept vorgesehenen Schritte führten.

5.1 Rahmenbedingungen der Implementierung

Für die Implementierung wird Jenkins mit den in Abschnitt 3.4 vorgestellten Plugins genutzt. Die Auswahl dieser Programme und Plugins liegt darin begründet, dass sie zurzeit im AMCS-System genutzt werden und diese Bachelorarbeit dieses System erweitern wird.

5.2 Auswahl der Geräte

Wie bereits in Abschnitt 2.1.3 angesprochen wurde, kann man die Tests einer App nicht auf allen verfügbaren Android-Geräten ausführen, da es mindestens 24.000 verschiedene genutzte Android-Modelle gibt. Man muss sich auf eine kleine Menge von Android-Geräten beschränken, welche einen möglichst großen Teil der Nutzer abdeckt.

5.2.1 Gegenüberstellung von Emulatoren und Echte Android-Geräte

In dem Abschnitt 2.1.3 wurde bereits erwähnt, dass Tests einer Android-App sowohl auf Emulatoren, als auch auf echten Android-Geräten ausgeführt werden können. Da es Device Clouds, wie etwa das Firebase Test Lab[5], gibt, welche sowohl echte Android-Geräte als auch Emulatoren zur Testausführung anbieten, wird folgend untersucht, ob man im Rahmen der Implementierung Emulatoren oder Android-Geräte zum Ausführen der Tests nutzen sollte.

Aus dem bereits oben genannten Abschnitt geht hervor, dass der größte Vorteil von Emulatoren ist, dass man diese vielfach nutzen kann, ohne diese kaufen zu müssen. Da man in einer Device Cloud auch die physischen Geräte nicht käuflich erwerben muss, um diese zu nutzen, ist dieser Vorteil von Emulatoren für die Betrachtungen nicht relevant.

Die Ausführung von Tests auf Emulatoren kann jedoch billiger als auf echten Android-Geräten sein. Dies ist z.B. im Firebase Blaze Plan[5] der Fall. Dies ist jedoch im Rahmen dieser Arbeit nicht zu beachten, da nach Anforderung A5 nur die kostenlose Angebot der Device Clouds betrachtet werden.

Ein großer Vorteil von echten Android-Geräten, welcher in Kapitel 2 genannt wurde, ist, dass diese bei der Ausführung von Tests ein Android-Gerät nicht nur emulieren, sondern tatsächlich darstellen. Somit sind die ausgegeben Testresultate von echten Geräten genauer als die von Emulatoren.

Da eine genaue Ausgabe der Testresultate vorteilhaft ist, um die AMCS-App weiter zu entwickeln, werden im Folgenden nur echte Android-Geräte betrachtet.

5.2.2 Statistiken der AMCS-App

Es wird eine erste Auswahl von Android-Geräten getroffen, die zur Ausführung von Tests in den Device Clouds genutzt werden kann. Dazu wird untersucht, welche Android-Geräte zum aktuellen Zeitpunkt[43] die veröffentlichte AMCS App aus dem Google Play Store nutzen.

Eine Vielzahl von Daten zur Nutzung einer App, welche im Google Play Store veröffentlicht wurde, können in der **Google Play Console (GPC)**[44] eingesehen werden. Über diese Webseite können unter anderem neue Versionen einer App veröffentlicht, aber auch Nutzerstatistiken zu bereits veröffentlichten Apps analysiert werden.

Zunächst bietet die Google Play Console eine Statistik über die „Installationen auf aktiven Geräten“ an. Es wird geschrieben, dass diese Statistik über alle Android-Smartphones erhoben wird, welche die AMCS-App installiert haben und innerhalb der letzten 30 Tage aktiv waren. Dabei zeigt sich, dass die AMCS-App zurzeit auf 88 aktiven Geräten installiert ist. Von diesen aktiven Android-Smartphones sind die drei am meisten genutzten OS-Versionen mit 31.8% Android 7.0, mit 23.9 % Android 6.0 und mit 11.4% Android 8.0 . Des Weiteren kann eine Auflistung aller Android-Modelle erhoben werden, welche die AMCS-App zurzeit installiert haben. Die am meisten genutzten Android-Modelle sind das Huawei P9, das Samsung Galaxy S7 sowie das Huawei P8 Lite mit jeweils vier Nutzern.

Auch in dieser Statistik ist die Fragmentierung bei den Android-Geräten wiederzufinden. Insgesamt gibt es zurzeit bei 88 Nutzern 59 verschiedene Android-Modelle, welche aktiv genutzt werden und die AMCS App nutzen.

Die genauen OS-Versionen der Modelle konnten nicht ermittelt werden, da die Google Play Console bei einer ausführlichen Anzeige der Daten diese mit der Fehlermeldung „Es ist ein unerwarteter Fehler aufgetreten. Bitte versuche es später noch einmal.“ quittierte. Spätere Versuche führten zu der gleichen Nachricht.

Aus der kleinen Menge an Nutzern ist es nicht möglich, aufschlussreiche Ergebnisse bezüglich der zu verwendenden Android-Geräte zu ermitteln. Den Daten kann jedoch entnommen werden, dass Android-Smartphones im Vergleich zu Android-Tablets eine übergeordnet Rolle spielen. Im Zuge dessen werden nur Smartphones weiter betrachtet. Eine Eingrenzung der Geräte ist auf Grund der Fragmentierung ohnehin nötig.

Im nächsten Abschnitt werden wissenschaftliche Arbeiten und Statistiken betrachtet, welche sich mit der Verteilung von Android-Geräten auf alle Nutzer beschäftigen.

[43] 27.05.2018

[44] https://developer.android.com/distribute/console/, zuletzt besucht am 05.06.2018

5.2.3 Daten aus externen Quellen

In der Arbeit (Vilkomir, 2018) wurde untersucht, welche Menge an Android-Geräten genutzt werden sollte, um Fehler von Apps zu finden, welche nur auf spezifischen Geräten entsteht. Dazu wurden 15 Android-Apps getestet, bei denen zuvor festgestellt wurde, dass sie insgesamt 24 Fehler beinhalten, welche nur auf speziellen Android-Geräten entstehen. Dabei wurde unter Nutzung von 30 verschiedenen Geräten überprüft, anhand welcher Kriterien diese Android-Geräten ausgewählt werden sollten, um möglichst viele der spezifischen Fehler zu finden.

Es stellte sich heraus, dass eine Nutzung von fünf Android-Geräten mit möglichst vielen verschiedenen OS-Versionen dabei die meisten Fehler entdecken konnte. Es konnten dabei 90 % der spezifischen Fehler gefunden werden, während alle andere Auswahlkriterien weniger Fehler finden konnten.

In dieser Quelle ist allerdings zu lesen, dass die aktuellste genutzte OS-Version Android 4.4.2 war. Diese Version ist jedoch bereits 2013 erschienen und ist damit schon veraltet. Außerdem werden zurzeit Android 4.4.2 und alle älteren Versionen von Android insgesamt nur noch von 15,3 % der Android-Nutzer genutzt.[45] Die drei aktuell am meisten genutzten Android OS-Versionen sind Android 6.0 mit 25.5 %, Android 7.0 mit 22.9% und Android 5.1 mit 17.6 % der Installationen auf Android-Smartphones.[45] Die ersten beiden Plätze decken sich somit mit den Daten der Statistiken der AMCS App aus der Google Play Console, während die am dritthäufigsten genutzte OS-Version bei der AMCS App Android 8.0 ist.

Aus der Quelle (Vilkomir, 2018) lässt sich somit ableiten, dass es wichtig ist, möglichst viele Android-Geräte mit unterschiedlichen OS-Versionen zur Testausführung zu nutzen, um viele Fehler innerhalb einer App zu finden. Jedoch sollten heutzutage Android-Geräte mit einer aktuellen OS-Version genutzt werden.

Zuletzt wird eine Statistik von AppBrain aufgeführt. [46] In dieser werden Statistiken über die Android-Modelle von weltweit genutzten Android-Smartphones erhoben. Dabei werden diejenigen Android-Smartphones betrachtet, welche eine App installiert haben, die die *AppBrain SDK* nutzt. Diese SDK wird in 70.000 verschiedenen Apps zur Monetarisierung dieser genutzt. [47]

Im folgenden wird die für Deutschland erhobene Statistik untersucht, da sich laut der Google Play Console 84 der insgesamt 88 aktiven Nutzer der AMCS-App in Deutschland befinden.

Die fünf in Deutschland am meisten genutzten Android-Modelle sind das Samsung Galaxy S7 mit 7,4 %, das Samsung Galaxy S8 mit 7,1% und das Samsung Galaxy S7 Edge mit 4,7 %, das Samsung Galaxy A5(2017) mit 3,7 % und das Samsung Galaxy S6 mit 3,7 % der Nutzer.

Im Folgenden soll aus den nun bestimmten Daten eine Menge von Android-Geräten ermittelt werden, welche potenziell möglichst viele Fehler bei der Testausführung der AMCS-App erzeugt. So kann man auftretende Bugs in der App finden und lösen.

Die von AppBrain bestimmten Daten werden doppelt so sehr gewichtet wie die Daten aus der Google Play Console. Um die Anzahl der genutzten Android-Geräte möglichst gering zu halten, wurde sich dafür entschieden, eines der am meisten genutzten Android-Modelle der AMCS-App und die zwei am meisten genutzten Android-Modelle zu nutzen, welche aus der von AppBrain veröffentlichten Statistik hervorgehen.

[45] vgl. https://developer.android.com/about/dashboards/#google-play-install-stats, zuletzt besucht am 27.05.2018

[46] https://www.appbrain.com/stats/top-android-phones-tablets-by-country?country=DE, zuletzt besucht am 27.05.2018

[47] https://www.appbrain.com/info/help/sdk/index.html, zuletzt besucht am 27.05.2018

Android-Modell	Grund der Nutzung
Samsung Galaxy S7	Eines der am meisten genutzten Android-Modelle in Deutschland.
Samsung Galaxy S8	Eines der am meisten genutzten Android-Modelle in Deutschland.
Huawei P9	Laut GPC eines der am meisten genutzten Android-Modelle.

Tabelle 5.1: Zu betrachtende Android-Geräte

Einige dieser Android-Modelle können mit verschiedenen OS-Version genutzt werden. So wurde beispielsweise das Samsung Galaxy S8 mit der Android Version 7.0 veröffentlicht, hat aber später ein Update auf Android 8.0 erhalten. Bei der Einrichtung der Device Clouds wird darauf geachtet, dass die genannten Android-Modelle in möglichst unterschiedlichen OS-Versionen genutzt werden, um nach (Vilkomir, 2018) möglichst viele Fehler innerhalb der App zu finden. Im Zuge dieser Arbeit wird in jeder Device Cloud die App nur in der deutschen Sprache getestet, um das Testkontingent der Device Clouds möglichst langsam zu verbrauchen.

5.3 Vorauswahl der Device Clouds

In diesem Abschnitt werden alle in Abbildung 2.1 vorgestellten Device Clouds nach den in Tabelle 3.1 zusammengefassten Anforderungen untersucht. Dabei werden die einzelnen Device Clouds noch nicht implementiert, sondern anhand der von den Firmen bereitgestellten Information verglichen. Es werden alle Device Clouds zunächst anhand der Anforderungen A2 bis A5 bewertet und so eine Vorauswahl für die Implementierung vorgenommen.

Zunächst ist anzumerken, dass (Braun et al., 2017) nicht für alle Device Clouds eine gültige Referenz verlinkt hat und somit teilweise unklar ist, welche Device Cloud betrachtet wurden. An den entsprechenden Stellen wird darauf hingewiesen.

Google Cloud Test Lab

Diese Device Cloud wurde mit der Firebase Test Lab Device Cloud vereint. Die Webseite von Google Cloud[48] verweist bei der Auswahl des Cloud Test Labs auf die Webseite des Firebase Test Labs.

Firebase Test Lab[49]

Die Firebase Test Lab Device Cloud (kurz: **Firebase**) stellt einen Dienst dar, welcher Espresso Tests unterstützt[49] (A2). Des Weiteren werden eigene Android-Geräte[5] angeboten und somit handelt es sich nicht um eine on-premises Device Cloud (A3). Eine Anleitung zur Integration mit Jenkins ist in (Martínez, 2017) zu finden (A4). Firebase bietet zudem den kostenlosen Spark-Plan[5] an, um die Tests auszuführen (A5).

Somit erfüllt Firebase die zunächst geprüften Anforderungen und wird in einem Prototypen implementiert.

[48] vgl. https://cloud.google.com/products/, zuletzt besucht am 01.06.2018
[49] https://firebase.google.com/docs/test-lab/, zuletzt besucht am 01.06.2018

AWS Device Farm[50]

Um das kostenlose Testkontingent[51] der AWS Device Farm (kurz: **AWS**) nutzen zu können, muss der Nutzer beim Einrichten eines Benutzerkontos eine Kreditkartennummer hinterlegen. Die Kreditkarte wird dabei automatisch belastet, sobald das kostenlose Kontingent erschöpft ist. Auf Nachfrage antwortete der Amazon-Support, dass es keine Möglichkeit gibt, die Kreditkarte dagegen zu sperren. Somit erfüllt AWS die Anforderung A5 nicht und wird deshalb nicht zur Implementierung in Erwägung gezogen.

Auch ein wissenschaftliches Benutzerkonto[52], welches für Forschungsarbeiten mit AWS angeboten wird, kann nicht in Betracht gezogen werden. Der Grund ist, dass die Vergabe dieser Konten bei einer Anfrage zu Beginn der Arbeit [53] bis Ende September[54] dauern würde. Zu dieser Zeit sind die Untersuchungen bereits abgeschlossen.

Experitest SeeTestCloud Online

Die Device Cloud mit diesem Namen konnte leider nicht gefunden werden. Der Autor geht davon aus, dass es sich um experitest SeeTest Continuous Testing Platform[55] handelt. Die oben genannte Device Cloud bietet keine öffentlich dokumentierte Möglichkeit[56], die Device Cloud mit Jenkins ohne das Build-Tool Ant[57] anzubinden. Da die AMCS-App jedoch das Build-Tool Gradle nutzt, wird die Anforderung A4 nicht erfüllt und die von experitest angebotene Device Cloud nicht implementiert.

SauceLabs Automated Testing Platform

Der Name der Device Cloud „SauceLabs Automated Testing Platform“, welcher in (Braun et al., 2017) für diese Device Cloud genutzt wurde, konnte leider nicht bestätigt werden. In dieser Quelle fehlt leider auch eine Referenz auf die entsprechend untersuchte Device Cloud. Es wird angenommen, dass es sich in (Braun et al., 2017) um die von Sauce Labs[58] angebotene **Continuous Testing Cloud (kurz: Sauce Labs)**[59] handelt.

Sauce Labs unterstützt Espresso Tests[60] (A2). Außerdem werden eigene Android-Geräte zum Ausführen der Tests angeboten[61] (A3). Es existiert eine Dokumentation zur Integration mit Jenkins[62] (A4) und es wird eine kostenlose Testphase[61] angeboten (A5). Somit erfüllt Sauce Labs alle der zunächst angelegten Anforderungen und wird in einem Prototypen implementiert.

[50] https://aws.amazon.com/de/device-farm/, zuletzt besucht am 01.06.2018
[51] https://aws.amazon.com/de/device-farm/pricing/, zuletzt besucht am 01.06.2018
[52] https://aws.amazon.com/de/grants/, zuletzt besucht am 01.06.2018
[53] April 2018
[54] https://aws.amazon.com/de/research-credits/faq/, zuletzt besucht am 01.06.2018
[55] https://experitest.com/mobile-testing/android-app-testing/, zuletzt besucht am 01.06.2018
[56] https://docs.experitest.com/display/TD/SeeTestAutomation-+Integration+With+Jenkins, zuletzt besucht am 01.06.2018
[57] https://ant.apache.org/bindownload.cgi, zuletzt besucht am 01.06.2018
[58] https://saucelabs.com/, zuletzt besucht am 28.05.2018
[59] https://az184419.vo.msecnd.net/sauce-labs/white-papers/core-elements-of-continuous-testing-a-sauce-labs-whitepaper.pdf, zuletzt besucht am 28.05.2018
[60] vgl. https://wiki.saucelabs.com/display/DOCS/Using+Espresso+for+Real+Device+Testing, zuletzt besucht am 01.06.2018
[61] https://signup.saucelabs.com/signup/trial, zuletzt besucht am 01.06.2018
[62] https://wiki.saucelabs.com/display/DOCS/Setting+Up+Sauce+Labs+with+Jenkins, zuletzt besucht am 01.06.2018

Xamarin Test Cloud

Die von Xamarin angebotene Device Cloud ist nun Teil[63]des Visual Studio App Centers[64]. Das Visual Studio App Centers bietet keine öffentliche Dokumentation zur Integration in Jenkins an, so dass die Anforderung A4 nicht erfüllt wird. Aus diesem Grund wird das Visual Studio App Center nicht zur Implementierung in das AMCS-System verwendet.

Mobile Labs deviceConnect[65]

Der Dienst Mobile Labs deviceConnect bietet sowohl eine on-premises, als auch eine public Device Cloud an. Im Folgenden wird nur die public Device Cloud betrachtet. Diese bietet keine Dokumentation zur Ausführung von Espresso Tests in der Device Cloud unter Zuhilfenahme von Jenkins an (A4). Somit wird eine Implementierung nicht in Betracht gezogen.

Perfecto Continuous Quality Lab[66]

Die von Perfecto angebotene Device Cloud bietet zwar eine Testphase[67] und eine Dokumentation[68] zur automatisieren Ausführung der Tests mit einem Gradle Plugin an. Jedoch ist eine automatisierte Ausführung von Tests auf echten Android-Geräten in der Testphase nicht möglich. Somit wird die Anforderung A5 nicht erfüllt, und die Device Cloud wird nicht implementiert.

Borland Silk Mobile Testing

Dieser Dienst wird nun unter dem Namen Micro Focus Silk Mobile angeboten.[69] Nachdem der Autor bei Micro Focus nach einem kostenlosen Forschungskonto nachgefragt hat, wurde dieses ihm auch zugesagt. Jedoch verkündete das Unternehmen dem Autor später, dass Micro Focus nur Softwarelösungen anbietet und keine eigene Device Cloud anbietet. Es handelt sich somit um eine on-premises Lösung, so dass Anforderung A3 nicht erfüllt und eine Implementierung nicht in Betracht gezogen wird.

HPE Mobile Center

HPE wurde mit Micro Focus fusioniert (vgl. (Ostler, 2017)). Somit wird der oben genannte Dienst nun unter dem Namen Micro Focus Mobile Center[70] betrieben. Wie bereits im vorherigen Abschnitt erläutert wurde, stellt Silk Mobile eine on-premises Lösung dar und verstößt somit gegen die Anforderung A3. Eine Implementierung wird nicht durchgeführt.

[63] https://testcloud.xamarin.com/, zuletzt besucht am 01.06.2018
[64] https://appcenter.ms/, zuletzt besucht am 09.06.2018
[65] https://mobilelabsinc.com/products/deviceconnect, zuletzt besucht am 01.06.2018
[66] https://www.perfecto.io/the-cloud-based-testing-lab/, zuletzt besucht am 01.06.2018
[67] https://www.perfecto.io/free-trial/, zuletzt besucht am 01.06.2018
[68] https://developers.perfectomobile.com/display/TT/Use+the+Gradle+Plugin+in+Android+Studio+Project, zuletzt besucht am 01.06.2018
[69] vgl. https://www.microfocus.com/de-de/products/silk-portfolio/silk-mobile/, zuletzt besucht am 01.06.2018
[70] https://software.microfocus.com/en-us/products/mobile-testing/overview, zuletzt besucht am 01.06.2018

Keynote Device Anywhere[71]

Der Link, welcher in (Braun et al., 2017) zu dieser Device Cloud angeführt wird, verweist nun auf Sigos App Experience[72] (kurz: **Sigos**). Sigos bietet eine kostenlose Testphase an.[73] Nach der Registrierung für ein solches Testkonto erhält der Nutzer eine E-Mail, in der man gebeten wird, einen Termin für ein telefonisches Gespräch anzugeben. Jedoch wurde die Antwort des Autors auf diese E-Mail zweimal nicht beantwortet. Somit erfüllt die Device Cloud von Sigos die Anforderung A5 nicht, da es nicht möglich ist, die Device Cloud kostenlos zu testen. Aus diesem Grund findet keine Implementierung statt.

SSTS pCloudy[74]

Diese Device Cloud bietet eine Testphase[75], sowie ein Jenkins Plugin[76] an. Jedoch stellte sich während einer ersten Implementierung heraus, dass eine automatisierte Nutzung von echten Android-Geräten nicht in der Testversion angeboten wird. Somit wird die Anforderung A5 nicht erfüllt und eine Implementierung wird nicht durchgeführt.

testdroid

testdroid stellt einen Teil von Bitbar dar. Dies wird daraus geschlussfolgert, dass bei einem Besuch der Webseite von testdroid[77] eine Anmeldeseite von Bitbar angezeigt wird.

Daraus wird geschlossen, dass es sich bei testdroid um Bitbar Testing[78] (kurz: **Bitbar**) unter Nutzung der Public Cloud[79] handelt.

Bitbar bietet ein kostenloses Testangebot an[78], jedoch wurde dem Autor zudem ein kostenloses Forschungskonto angeboten. Als Gegenleistung möchte Bitbar die ausgefertigte Bachelorarbeit erhalten. Von nun an wird nicht weiter das kostenloses Testangebot von Bitbar betrachtet, sondern der volle Umfang von Bitbar, welcher durch das Forschungskonto bereitgestellt wird. Dieses bietet eine kostenlose Ausführung von Tests auf allen Geräten der Device Cloud für die Zeit, in der diese Arbeit geschrieben wird.

Espresso Tests werden bei Bitbar unterstützt[79] (A2). Außerdem werden eigene Android-Geräte angeboten[78] (A3) und es gibt eine Dokumentation[80] zur Integration der Device Cloud mit Jenkins (A4). Zuletzt kann die Device Cloud durch das Forschungskonto kostenlos genutzt werden (A5). Bitbar erfüllt daher alle der zunächst angelegten Anforderungen und wird prototypisch implementiert.

SOASTA TouchTest[81]

Da SOASTA keine öffentlichen Aussagen darüber tätig, ob ihre Device Cloud mit dem in der AMCS-App genutzten Testframework Espresso genutzt werden kann, wird Anforderung A2 nicht erfüllt. Ein Prototyp wird somit nicht angefertigt.

---

[71] http://www.keynote.com/solutions/testing/mobile-testing, nach (Braun et al., 2017)
[72] https://appexperience.sigos.com/, zuletzt besucht am 01.06.2018
[73] http://info.sigos.com/trial, zuletzt besucht am 01.06.2018
[74] https://www.pcloudy.com/, zuletzt besucht am 01.06.2018
[75] https://device.pcloudy.com/signup, zuletzt besucht am 01.06.2018
[76] http://support.pcloudy.com/support/solutions/articles/9000074146-pcloudy-plugin-for-jenkins, zuletzt besucht am 01.06.2018
[77] https://cloud.testdroid.com/#, zuletzt besucht am 01.06.2018
[78] https://Bitbar.com/testing/, zuletzt besucht am 01.06.2018
[79] https://Bitbar.com/testing/solutions/public-cloud/, zuletzt besucht am 01.06.2018
[80] http://docs.Bitbar.com/testing/cloud-integrations/jenkins-plugin/jenkins-ui/index.html, zuletzt besucht am 01.06.2018
[81] https://www.soasta.com/videos/demo-of-mobile-app-testing-with-touchtest/, zuletzt besucht am 02.06.2018

TestPlant eggCloud[82]

Der in (Braun et al., 2017) gegebene Link zu dieser Device Cloud funktioniert nicht mehr. Es wird davon ausgegangen, dass diese Device Cloud in Eggplant Automation Cloud[83] umbenannt wurde.

Bei dieser Device Cloud bekommt man jedoch keinen Zugriff auf eine Menge von bereitgestellten Android-Geräten, sondern muss sich selbstständig um eine Menge von „systems under test“[84] kümmern. Somit wird die Anforderung A5 nicht erfüllt, da keine echten Android-Geräte kostenlos nutzbar sind. Eine Implementierung wird deshalb nicht durchgeführt.

Experitest SeeTestCloud Onsite[85]

Diese Device Cloud von experitest bietet, wie schon auf Abbildung 2.1 zu sehen ist, eine on-premises[86] Device Cloud an. Somit erfüllt diese Device Cloud die Anforderung A3 nicht und wird nicht implementiert.

Zusammengefasst werden im folgenden Abschnitt Prototypen für die drei Device Clouds Sauce Labs, Firebase und Bitbar umgesetzt. Diese Device Clouds erfüllen die Anforderungen A2 bis A5.

5.4 Entwicklung der Prototypen

5.4.1 Probleme zu Beginn

In der Anfangsphase der Implementierung entschied sich der Autor dafür, zunächst eine Anbindung an die Device Cloud Firebase herzustellen. Eine Ausführung der Tests führte dabei jedoch zu folgendem Fehler im Testbericht auf der Webseite von Firebase:

```
1 java.lang.NullPointerException: ...
2   at de.tudd.amcs.util.StorageUtil.savePrefString
3   at de.tudd.amcs.fcm.MyInstanceIDListenerService.onTokenRefresh
```

.

Auch bei Bitbar wurde dieser Fehler in Form eines „Issues“ auf der Webseite im Testbericht dargestellt.

Bei Sauce Labs hingegeben war die Identifizierung des Fehlers nicht auf einem ähnlich einfachen Weg möglich. Obwohl die Testausführung mit „SUCCESS“ auf der Webseite von Sauce Labs abgeschlossen waren, wurden die Tests nicht durchgeführt. Dies stellte sich erst dadurch heraus, als bei der Anzeige der „Test Cases“ auf der Webseite von Sauce Labs kein einziger Name eines Tests angeführt war. Dies bedeutet, dass kein Test ausgeführt wurde.

Nur durch eine gezielte Untersuchung des Device Logs des Android-Geräts konnte die bereits oben genannte Exception gefunden werden. Ohne die Informationen von Bitbar und Firebase wäre dieser Fehler wahrscheinlich erst zu einem späteren Zeitpunkt der Implementierung entdeckt worden.[87]

[82] http://www.testplant.com/eggplant/testing-tools/eggcloud/, nach (Braun et al., 2017)
[83] http://docs.testplant.com/eCL/eggcloud-documentation-home.htm, zuletzt besucht am 02.06.2018
[84] http://docs.testplant.com/eCL/ecl-getting-started-eggcloud.htm, zuletzt besucht am 02.06.2018
[85] https://experitest.com/mobile-cloud-testing/seetestcloud-onsite/, zuletzt besucht am 02.06.2018
[86] https://www.youtube.com/watch?time_continue=10&v=irL28ZfLhEs, zuletzt besucht am 02.06.2018
[87] Siehe Jenkins - nwuenscheSauceLabs #9

Beschreibung des Problems und Lösung

Anhand eines Beispieles soll das gefundene Problem verdeutlicht werden. Innerhalb der Klasse **Util** der AMCS-App wird der aktuelle **Context** der App gespeichert, welcher Zugriff auf globale Informationen der App gibt.[88] Solche globalen Information stellen auch die *Shared Preferences* dar. Sie stellen einen einfachen Schlüssel-Wert Speicher dar, welcher auch über den Lifecycle einer Android-App hinaus erhalten bleibt.[89] In der Methode **savePrefString**, welche als Auslöser des Fehlers von den Device Clouds von Firebase und Bitbar angeführt wurde, wird eine Zeichenkette in die Shared Preferences gespeichert. Jedoch wurde der Context, welcher zwischengespeichert wurde, vor der Ausführung von savePrefString intern von Android auf `null` gesetzt. Da die Methode nicht überprüft, ob der Context `null` ist oder nicht, wird somit eine NullPointerException geworfen.

Nun wird die erste Lösung beschrieben, welche den geworfenen Fehler behebt, ohne jedoch das darunterliegende Problem zu lösen. Diese Lösung besteht darin, zu überprüfen, ob der Context auf `null` gesetzt wurde, bevor er in der Methode savePrefString genutzt wird. Eine entsprechende Überprüfung wurde auch in andere Methoden eingefügt, welche auf den Context zugreifen.

Als diese Lösung implementiert wurde, konnten die Tests in den Device Clouds ohne den vorher beschriebenen Fehler ausgeführt werden. Jedoch wird durch diese Lösung die Zeichenkette, welche eigentlich in den Shared Preferences gespeichert werden sollte, nicht abgespeichert. In einer besseren Lösung des Problems sollte jedoch das Speichern der Zeichenkette bei einem auf `null` gesetzten Context nicht übersprungen werden, sondern sie sollte unter Zuhilfenahme des aktuellen Context dauerhaft abgespeichert werden.

Für eine solche Lösung stand der Entwickler der AMCS-App und Autor der Arbeit (Buchholz, 2017), Patrick Buchholz, dem Autor für Fragen zur Seite. Ein erstes Konzept für diese Lösung stammt auch von ihm. Dabei wurde vor der Ausführung der Methode savePrefString der gespeicherte Context in der Klasse Util aktualisiert, sodass dieser bei der Speicherung der Zeichenkette aktuell ist.

5.5 Sauce Labs

Im folgenden Abschnitt wird die Implementierung der von Sauce Labs bereitgestellten Device Cloud beschrieben.

Erwerb einer Testlizenz

Sauce Labs bietet kein dauerhaft nutzbares Testangebot ihrer Device Cloud an.[90] Jedoch wird eine Testphase angeboten. Nach der Erstellung eines Benutzerkontos bei Sauce Labs kann man innerhalb der nächsten 14 Tage die von Sauce Labs angebotenen Android-Geräte für insgesamt 100 Minuten zur automatisierten Ausführung von Tests nutzen.[61]

Hierbei muss der Nutzer unter anderem den Namen seiner Firma angeben. Im Zuge dieser Arbeit wurde jedoch festgestellt, dass ein Bindestrich als Name der Firma ausreicht, um für die Testphase der Device Cloud freigeschalten zu werden.

[88] vgl. https://developer.android.com/reference/android/content/Context, zuletzt besucht am 30.05.2018
[89] vgl. https://developer.android.com/training/data-storage/shared-preferences, zuletzt besucht am 30.05.2018
[90] https://saucelabs.com/pricing, zuletzt besucht am 28.05.2018

Lokale Einrichtung der Testumgebung

Zu Beginn der Implementierung (April 2018) funktionierte der in der Benutzeranleitung[60] gegebene Link zum Herunterladen der Datei „runner.jar“ nicht. Des Weiteren konnte auch der von Sauce Labs angebotene „Sauce Labs Espresso Runner“[91] nicht genutzt werden, um die Sauce Labs zu starten. Bei der Ausführung des angebotenen Releases v0.2.1 mit den in der README-Datei aufgezählten Parametern folgte der Fehler:

```
1 Exception in thread "main" com.beust.jcommander.ParameterException: Was passed
      main parameter '--apiKey' but no main parameter was defined in your arg class
```

.

Das eigene Bauen der Datei „Sauce Labs Espresso Runner“ auf dem Computer des Autors (Betriebssystem Ubuntu 18.04[92], OpenJDK 1.8.0_171, Maven[93] 3.5.2, Chromium[94] 66.0.3359.181) nach der Anleitung aus der README-Datei führte zu folgendem Fehler:

```
1 [ERROR] Failed to execute goal on project espressorunner: Could not resolve
      dependencies for project org.testobject:espressorunner:jar:0.3.0-SNAPSHOT:
      Failure to find org.testobject:testobject-java-api:jar:0.1.6 in https://repo.
      maven.apache.org/maven2 was cached in the local repository, resolution will
      not be reattempted until the update interval of central has elapsed or
      updates are forced -> [Help 1]
```

.

Ein Forcieren der Updates mit `mvn clean package -U` führte zu dem selben Fehler.

Erst auf eine Anfrage hin erhielt der Autor die in der Benutzeranleitung[60] genutzten Datei „runner.jar“ in der Version 1.1. Diese kann mittlerweile auch über den in der Benutzeranleitung vorhandenen Link heruntergeladen werden und wird im Folgenden **Runner-Datei** genannt.

Im folgenden wird anhand der offiziellen Anleitung[60] von Sauce Labs beschrieben, wie man die Sauce Labs Device Cloud aus einer Konsole heraus starten kann. Es wurde dabei nicht die Anleitung[62] für das Jenkins-Plugin genutzt. Dies hat den Grund, dass die in der ersten Anleitung beschriebene Integration auch ohne ein zusätzliches Freestyle-Projekt funktioniert. Der benötigte Konsolenbefehl mit der Runner-Datei kann in einem sh-Step des Jenkinsfiles ausgeführt werden.

Im folgenden wird beschrieben, welcher Weg zur erfolgreichen Integration der Device Cloud nötig war.

Wenn man sich mit seinem Benutzerkonto bei Sauce Labs anmeldet, muss man zunächst auf den Button mit der Aufschrift „Access Real Devices“ drücken, um zur tatsächlichen Einrichtung der Device Cloud zu gelangen. Es sei hierbei angemerkt, dass man auf eine Webseite der Firma TestObject [95] weitergeleitet wird. TestObject ist eine Firma, welche eine Device Cloud mit echten Android-Geräten zum Testen von Android-Apps anbietet und 2016 von SauceLabs übernommen wurde.[96] Es ist nicht möglich, sich mit dem bei Sauce Labs angelegten Benutzerkonto direkt über die Webseite von TestObject anzumelden, da der Benutzername nicht erkannt wird.

Nachdem man auf die Webseite von TestObject weitergeleitet wurde, muss man zunächst eine APK Datei seiner zu testenden App hochladen.

Auf der folgenden Webseite finden man in der Rubrik „Automated Testing → Espresso/Robotium → Setup Instructions “ einen vorgefertigten Konsolenbefehl, mit welchem die

[91] https://github.com/saucelabs/espresso-runner, zuletzt besucht am 20.04.2018
[92] http://releases.ubuntu.com/18.04/, zuletzt besucht am 06.06.2018
[93] https://maven.apache.org/, zuletzt besucht am 06.06.2018
[94] https://www.chromium.org/Home, zuletzt besucht am 11.06.2018
[95] https://testobject.com/de/home, zuletzt besucht am 28.05.2018
[96] vgl. https://saucelabs.com/news/sauce-labs-acquires-testobject-to-expand-real-device-mobile-app-testing-platform, zuletzt besucht 28.05.2018

Device Cloud angesprochen werden kann. Nachfolgend kann man auf der gleichen Webseite den `-device`-Parameter generieren lassen, welchen man benötigt, um das richtige Android-Gerät der Device Cloud anzusprechen.

Auswahl der Geräte

In der kostenlosen Testphase von Sauce Labs wird keines der in Tabelle 5.1 genannten Android-Geräte zur Ausführung der Tests angeboten. Es werden insgesamt fünf verschiedene Geräte angeboten, welche in der nachfolgenden Tabelle aufgelistet sind.

Android-Modell	**Datenzentrum**	**OS-Version**
LG Nexus 5X	EU	8.0
LG Nexus 5X	US	8.1
Motorola Moto E (2nd Gen)	EU	6.0
Motorola Moto E	US	5.1
Samsung Galaxy S6	US	7.0

Tabelle 5.2: Kostenlose Android-Geräte (Sauce Labs)

An dieser Menge an Android-Geräten ist dennoch positiv zu erwähnen, dass sie das Samsung Galaxy S6 enthält, welches nach Abschnitt 5.2.3 an fünfter Stelle der am meisten genutzte Android-Geräte in Deutschland steht. Des Weiteren benutzen die angebotenen Android-Geräte viele verschiedene OS-Versionen, was, wie im gleichen Abschnitt gesagt wurde, ein großer Vorteil zum Finden von Fehlern in der zu testenden App ist. Aus diesem Grund wird die Ausführung der Tests auf allen angebotenen Android-Geräten vorgenommen.

Einrichtung der Testumgebung auf dem Jenkins-Server

Nachdem nun der Konsolenbefehl und die genutzten Android-Geräte für die Sauce Lab Device Cloud herausgearbeitet wurden, wird nun die Anbindung an den vorhandenen Jenkins-Servers des AMCS-Projekts vorgestellt.

Wie bereits in Abschnitt 3.2.4 erwähnt wurde, basiert der vorhandene Jenkins-Server zum Großteil auf der Ausführung von Jenkinsfiles und nutzt nur bei der Ausführung der Integration Tests auf dem Emulator ein Freestyle-Projekt.

Zunächst wurde ein neuer Branch mit dem Namen **nwuenscheSauceLabs** erstellt, welcher den aktuellen Stand des Master Branches kopiert. Nun wurde die Stage „UI and Integration Tests“, welche zurzeit zur Ausführung der Integration Tests auf einem lokalen Emulator genutzt wurde, durch die folgende **Cloud Stage** ersetzt:

```
1          stage ('Cloud'){
2              ...
3              when {
4                  expression {BRANCH_NAME =~ "nwuenscheSauceLabs"}
5              }
6              steps {
7                  sh "./gradlew :amcs:assembleDebugAndroidTest"
8                  sh "java -jar runner.jar android ... 2>&1 | tee output.txt "
9                  sh "./post.sh"
10             }
11         }
```

.

Zunächst sei erwähnt, dass hier nur auf Anpassungen eingegangen wird, welche in (Buchholz, 2017) noch nicht vorgestellt wurden. Als Erstes wird im **when-Block** einer Stage ange-

geben, welche Voraussetzungen erfüllt sein müssen, damit die nachfolgenden Steps ausgeführt werden. In diesem Beispiel muss der Name des Branches, welchen Jenkins zurzeit betrachtet, mit dem des Branches des Sauce Labs Prototypen übereinstimmen, damit die Tests bei Sauce Labs ausgeführt werden.

In dem nun folgenden **steps-Block** werden die abzuarbeitenden Steps einer Stage festgehalten. Im ersten Step wird die Test-APK gebaut, welche für die Ausführung im nächsten Step benötigt wird. Das Bauen der normalen APK ist nicht nötig, da dies schon in der „Build"-Stage aus Abbildung 4.1 durchgeführt wurde. Das vorangestellte `:amcs:` gibt hierbei den Namen des zu bauenden Projektes an.

Im zweiten Step wird der im vorherigen Teil dieses Abschnitts aufgebaute Konsolenbefehl ausgeführt, um die Tests auf der Device Cloud zu starten. Um eine spätere Auswertung des gesendeten Testberichts von der Device Cloud vorzunehmen, werden diese in einer Datei gespeichert. Um das Testresultat sowohl in einer Datei zu speichern, als auch in der Konsolenausgabe von Jenkins zu archivieren, wird das Programm `tee` genutzt. Da die Runner-Datei unter anderem die Resultate von fehlgeschlagenen Tests nicht auf dem Standard-Datenstrom **stdout**, sondern auf dem Fehler-Datenstrom **stderr** ausgibt, muss mit Hilfe des Befehls `2>&1` zunächst der Fehler-Datenstrom in den Standard-Datenstrom geleitet werden, damit `tee` beide aufzeichnen kann.

Nun wird die Auswertung der Testergebnisse, welche in der Datei **post.sh** stattfindet, genauer vorgestellt.

Auswertung der Daten in Jenkins

Wenn man die beiden im letzten Absatz erläuterten Schritte ohne den dritten ausführt, bekommt der Nutzer des Jenkins-Servers nicht immer den richtigen Zustand der Pipeline ausgegeben. So kann eine Verweigerung der Testausführung durch die Device Cloud dennoch zu einem stabilen Pipeline Zustand führen.[97]

Wie in Abschnitt 2.6 bereits erwähnt, sollte dies jedoch zu einem instabilen Zustand führen. Es ist nicht ohne Probleme möglich, den Zustand eines Steps auf instabil zu setzen.[98] Um diesen Zustand zu erreichen, wird im dritten Schritt ein Skript mit dem Namen `post.sh` aufgerufen. Dieses Programm wird im weiteren **Post-Skript** genannt.

Das Post-Skript stellt hierbei ein Shell-Skript dar. Die Aufgabe des Post-Skriptes besteht darin, die Ausgabe der Konsole, welche mit Hilfe von `tee` in der Datei `output.txt` gespeichert wird, auszuwerten.

Wenn gewisse Schlüsselwörter in dieser Ausgabe gefunden werden, wird der Rückgabewert des Post-Skriptes verändert. Wenn der Rückgabewert des Skriptes nicht „0" beträgt, wird der Step von Jenkins als gescheitert markiert[99] und somit der Zustand der gesamten Pipeline auf gescheitert gesetzt.

Es wurde zunächst angenommen, dass man mit dem Rückgabewert „2" den Zustand eines Steps auf instabil setzen kann.[100] Jedoch funktioniert dies nur für Steps in Freestyle-Projekten und nicht in Pipeline-Projekten. Somit wird im Zuge der weiteren Implementierung nur zwischen den Zuständen stabil und gescheitert unterschieden. Dies stellt dennoch einen Fortschritt zum aktuellen Jenkins-System dar, da der Entwickler nun automatisch darüber informiert wird, dass der Zustand einer Pipeline nicht mehr stabil ist.

Eine Verfeinerung zur Nutzung von instabilen Zuständen könnte durch die Nutzung des Jenkins Plugins *Text-finder*[101] vollzogen werden. Im Ausblick dieser Arbeit wird noch einmal

[97] siehe Jenkins - nwuenscheSauceLabs #18

[98] https://groups.google.com/forum/#!topic/jenkinsci-users/pdIL2qB_0Eo, zuletzt besucht am 28.05.2018

[99] vgl. https://stackoverflow.com/questions/20845381/do-manual-build-fail-in-jenkins-using-shell-script/20845508, zuletzt besucht am 06.06.2018

[100] vgl. https://issues.jenkins-ci.org/browse/JENKINS-23786, zuletzt besucht am 05.06.2018

[101] https://wiki.jenkins.io/display/JENKINS/Text-finder+Plugin, zuletzt besucht am 30.05.2018

auf dieses Thema eingegangen.

```
1       FILE=$(cat output.txt)
2       TRIALOVER=$(echo "$FILE"| grep "HTTP 403 Forbidden")

4       if [ "$TRIALOVER" != "" ]; then
5           echo "Your Trial is over!"
6           exit 1
7       fi
```

.

Anhand eines Auszugs aus dem Post-Skript soll dargestellt werden, wie dieses funktioniert. Dieser Auszug hat die Aufgabe, den Zustand des Steps auf gescheitert zu setzen, falls die 14-tägige Testphase von Sauce Labs abgelaufen ist.[102]

Zunächst wird in der ersten Zeile der Inhalt der Datei `output.txt`, also die Konsolenausgabe, in eine Variable mit dem Namen `FILE` gespeichert. In der folgenden Zeile wird diese Konsolenausgabe mit Hilfe des Programms `grep` nach den Schlüsselwörtern „HTTP 403 Forbidden“ durchsucht. Diese Nachricht wurde ermittelt, indem die Runner-Datei mit einem bereits abgelaufenen API-Schlüssel genutzt wurde. Sollte diese Zeichenkette gefunden worden sein, wird diese in der Variable `TRIALOVER` gespeichert.

Somit kann in Zeile vier überprüft werden, ob der Inhalt dieser Variable leer ist. Ist sie leer, konnte die Zeichenkette nicht in der Ausgabe gefunden werden und die Testphase ist noch nicht abgelaufen. Die Bedingung in Zeile vier wird somit übersprungen, und das Post-Skript endet mit dem Standardrückgabewert „0“. Sollte die Variable jedoch nicht leer sein, und somit die Zeichenkette in der Konsole ausgegeben worden sein, wird der Rückgabewert in Zeile fünf auf „1“ gesetzt und das Skript beendet.

Wie bereits im Jenkinsfile gezeigt wurde, wird im dritten Step des Jenkinsfiles dieses Post-Skript ausgeführt. Sollte dieses den Wert „1“ zurückgeben, so sieht Jenkins diesen Step als gescheitert an. Somit wird auch der Zustand der Pipeline auf gescheitert gesetzt und der Entwickler bekommt dargestellt, dass die Ausführung der Pipeline nicht ordnungsgemäß durchlaufen werden konnte.

Im tatsächlich genutzten Post-Skript wird des Weiteren eine ähnliche Abfrage für die Ausgabe der Testresultate unternommen, welche jedoch analog zu dem oben genannten Beispiel mit dem Schlüsselwort „FAILURE“ funktioniert. Um dieses zu bestimmen, wurde ein Test mit dem Namen **alwaysFail** genutzt, welcher automatisch bei jeder Ausführung eine NullPointerException wirft und damit immer fehlschlägt. Außerdem wird mit Hilfe des Programms `awk` die URL der Webseite ausgelesen, auf welcher die detailierten Testresultate zu finden sind. Diese URL verweist allerdings auf die Webseite von TestObject. Bei dieser kann man sich jedoch, wie bereits oben dargestellt wurde, nicht mit seinen Sauce Labs Benutzerdaten anmelden. Man muss auch hier den Umweg über die Homepage von Sauce Labs gehen, um die detailierten Testresultate zu betrachten.

Um das Schlüsselwort zu finden, welches ausgegeben wird, wenn die Freiminuten abgelaufen sind, wurde zu Testzwecken die Ausführung der Runner-Datei im Jenkinsfile kopiert und insgesamt 50 Mal hintereinander ausgeführt. Wie sich dabei herausstellte, wurde keine der Testausführungen abgeschlossen und jede nach 60 Minuten mit der Fehlermeldung „unable to get test suite report result after 60 minutes“ abgebrochen. Wie in Abbildung 5.1 zu sehen ist, wird auf der Webseite von Sauce Labs nicht angezeigt, ob die Tests abgeschlossen wurden oder nicht. Der graue Kreis steht hierbei dafür, dass die Tests ausgeführt werden.

[102] siehe Jenkins - nwuenscheSauceLabs #19

Abbildung 5.1: Endlos laufende Tests (Sauce Labs)

Nach dieser Untersuchung konnte zunächst kein Test mehr bei Sauce Labs ausgeführt werden. Dies galt sowohl für das aktuell genutzte Benutzerkonto, sowie für ein neu angelegtes Konto. Dieser Fehler wurde erst am folgenden Tag behoben.

Jedoch wurden während der Implementierungsphase mit einem Konto bei Sauce Labs insgesamt über 40 Mal Tests ausgeführt. Bei einer durchschnittlichen Ausführungszeit von mindestens drei Minuten konnten mehr als die von Sauce Labs angegebenen[61] 100 freien Testminuten genutzt werden. Zudem wird die Anzeige der verbleibenden freien Zeit auf der Webseite von Sauce Labs nicht aktualisiert. Diese beträgt immer 1,7 Stunden, also rund 100 Minuten. Es kann sich hierbei auch nicht um die angebotenen manuellen Testminuten handeln. Diese Zahl beträgt nämlich laut Sauce Labs[61] nur 60 Minuten. Somit konnte kein Schlüsselwort für den Verbrauch der Testminuten in das Post-Skript eingepflegt werden.

Auswertung der detailierten Testresultate

Bei der Ausführung der Runner-Datei wird in der Konsole von Jenkins ausgegeben, ob die Ausführung der einzelnen Tests erfolgreich oder gescheitert ist und wie lange die Ausführung der Tests gedauert hat. Wenn man wissen möchte, warum ein Test gescheitert ist, kann man eine detailierte Ausgabe der Testresultate auf der Webseite von Sauce Labs wiederfinden.

Jedoch gibt es Probleme mit der Ausgabe der Testresultate. So wurde zwar der oben genannte alwaysFail ausgeführt[103], und es wurde auch in der Konsole ausgegeben, dass dieser fehlgeschlagen ist. Dennoch wird auf der Webseite in der Rubrik „Exceptions“ keine Nachricht ausgegeben, wie in Abbildung 5.2 zu sehen ist.

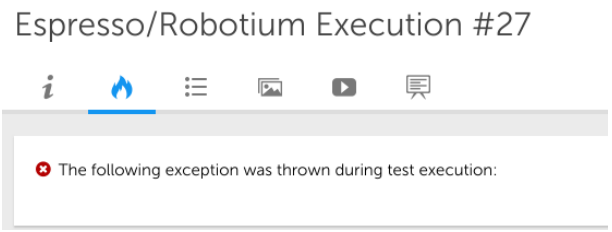


Abbildung 5.2: Fehlende Fehlermeldung bei gescheitertem Test (Sauce Labs)

Es ist jedoch möglich, im angegebenen Device Log nach der Exception zu suchen. Wenn

[103] siehe Jenkins - nwuenscheSauceLabs #46

man das Log-Level auf „VERBOSE“ einstellt, und den Namen alwaysFail sucht, wird die Zeilennummer des Quelltextes ausgegeben, welche den Test alwaysFail zum Scheitern brachte. Dies erfordert jedoch einen erhöhten Aufwand des Entwicklers. Dieser soll jedoch nach Anforderung A1 reduziert werden.

Ausführung aller ermittelten Android-Geräte

Im Folgenden soll gezeigt werden, wie die fünf in Tabelle 5.2 definierten Android-Geräte genutzt werden können, um die Tests der AMCS-App auszuführen. Zunächst wurde versucht, die Tests auf allen Android-Geräten gleichzeitig auszuführen. Dazu wurde nach der Anleitung von Sauce Labs [60] versucht, den Parameter `-device` an die Runner-Datei mehrfach zu übergeben. Jedoch trat dabei der Fehler „Can only specify option –device once.“ auf. Als nächstes sollte der Parameter in der Form *–device "Device1, Device2"* genutzt werden, jedoch wird dies in der Konsole mit der Ausgabe „HTTP 500 Internal Server Error“ quittiert.

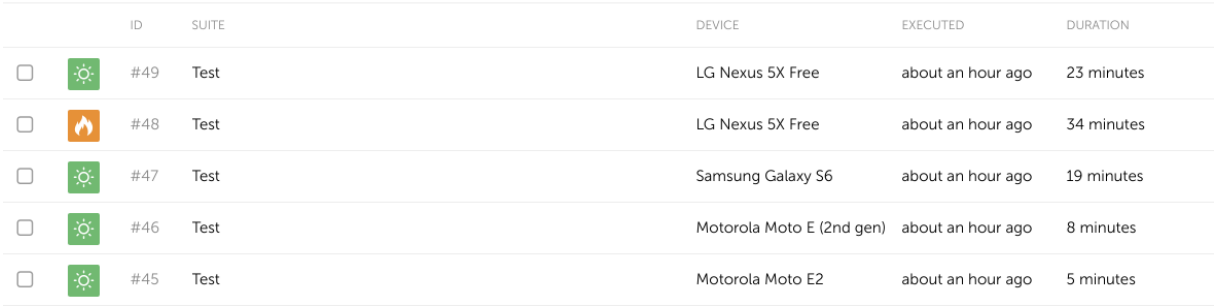
Als Folge dessen wurde ein weiteres Shell-Skript geschrieben, welches die Runner-Datei für jedes Android-Gerät parallel ausführt. Dieses Skript wurde **parallel.sh** genannt und hat folgende Struktur:

```
1    java –jar runner.jar ... ––device Device1 &
2    java –jar runner.jar ... ––device Device2 &
3    ...
4    java –jar runner.jar ... ––device Device5 &
5    wait
```

.

Der Befehl `&` hat den Zweck, die erste Ausführung der Runner-Datei im Hintergrund weiter auszuführen, während im Vordergrund der nächste Befehl ausgeführt wird. Somit laufen alle Ausführungen der Runner-Datei gleichzeitig. Um mit der weiteren Verarbeitung der Ausgaben zu warten, bis alle Ausführungen fertig sind, wird der Befehl `wait` verwendet. Dieses Skript wird nun im zweiten Step des Jenkinsfiles ausgeführt.

Tatsächlich werden zunächst fünf kostenlose Testausführungen gestartet. Es stellte sich allerdings heraus, dass die Ausführung der Tests auf den Geräten von Gerät zu Gerät zunahmen.[104] Es ist davon auszugehen, dass die Testausführungen nacheinander statt parallel ausgeführt werden.



	ID	SUITE	DEVICE	EXECUTED	DURATION
☐	#49	Test	LG Nexus 5X Free	about an hour ago	23 minutes
☐	#48	Test	LG Nexus 5X Free	about an hour ago	34 minutes
☐	#47	Test	Samsung Galaxy S6	about an hour ago	19 minutes
☐	#46	Test	Motorola Moto E (2nd gen)	about an hour ago	8 minutes
☐	#45	Test	Motorola Moto E2	about an hour ago	5 minutes

Abbildung 5.3: Parallel laufende Testausführungen (Sauce Labs)

Ergebnisse der Tests

Während der Implementierung des Prototypen konnten mehrere gescheiterte Testausführungen beobachtet werden. So stellt sich beispielsweise bei der Nutzung des Motorola

[104] siehe Jenkins - nwuenscheSauceLabs #54

Moto E (2nd gen)[105]heraus, dass der Test „testUsernameWithWhitespace“ fehlgeschlagen ist. Dieser Test überprüft, dass man bei der Anmeldung innerhalb der AMCS-App kein Leerzeichen in seinem Namen nutzen darf. Jedoch wurde, ähnlich wie in Abbildung 5.2 bereits gezeigt, keine Exception angezeigt. Eine Untersuchung des Device Logs wurde im Rahmen dieser Arbeit nicht durchgeführt.

5.6 Firebase

Erwerb einer Testlizenz

Firebase bietet ein dauerhaft nutzbares Testangebot an. Der Spark-Plan[5] ermöglicht dem Nutzer insgesamt fünf Testausführungen pro Tag auf echten Android-Geräten.

Lokale Einrichtung der Testumgebung

Im Folgenden wird die Anleitung nach (Martínez, 2017) genutzt. Es wird diese Anleitung genutzt, da sie bereits in der Arbeit (Buchholz, 2017) angeführt wird und sich diese Bachelorarbeit als Erweiterung von dieser Masterarbeit versteht. In der Anleitung aus (Martínez, 2017) wird der von Firebase angebotene Blaze-Plan genutzt, jedoch funktioniert sie auch mit dem in dieser Arbeit genutzten Spark-Plan.

Zunächst muss die zu testende App mit Firebase verbunden werden. Für die AMCS-App wurde dies bereits in (Buchholz, 2017) unternommen.

Im nächsten Schritt muss das Programm **gcloud** installiert werden, welches die Ausführungen von Tests in Firebase über die Konsole ermöglicht. Jedoch hat sich der in Schritt vier der Anleitung angegebene Befehl von `gcloud beta test android devices list` zu `gcloud beta firebase test android models list` geändert.

In dieser Arbeit wird keine „.yml“ Datei für die Auflistung der genutzten Android-Geräte genutzt, sondern die Konsolen-Parameter, welche in der offiziellen Anleitung zu gcloud[106] wiederzufinden sind. gcloud wurde hierbei der Parameter `-device-ids` übergeben. Dieser ist in der Dokumentation als veraltet markiert. Jedoch wurde zunächst vom Autor angenommen, dass sich dieser Parameter dazu eignet, ein genaues Android-Gerät anzuführen, auf dem die Tests ausgeführt werden sollen. Die dafür genutzten IDs der Android-Geräte wurden über den Befehl `gcloud beta firebase test android models list` bestimmt.

Auswahl der Geräte

Das in Tabelle 5.1 definierte Gerät Samsung Galaxy S7 kann auf Firebase genutzt werden, jedoch sind das Samsung Galaxy S8 sowie das Huawei P9 nicht als echte Android-Geräte wieder zu finden.

Aus diesem Grund wurde zunächst versucht, die Tests nur auf dem Samsung Galaxy S7 auszuführen. Jedoch führte dies mit der in der „build.gradle“-Datei des AMCS-Projektes angegebenen `targetSdkVersion=26` sowie `compileSdkVersion=26`, welche für die OS-Version 8.0 stehen, zu dem folgendem Fehler in Firebase:

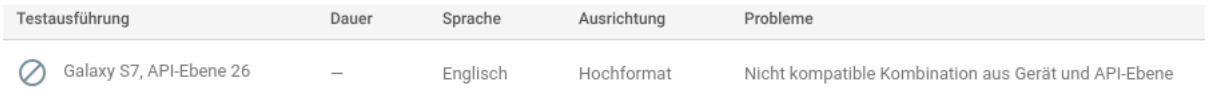


Abbildung 5.4: Inkompatibilität (Firebase)

[105]siehe Sauce Labs - muelleru #33

[106]https://cloud.google.com/sdk/gcloud/reference/firebase/test/android/run, zuletzt besucht am 01.06.2018

.

In der Android-Geräte Liste von gcloud wird ausgeschrieben, dass das Samsung Galaxy S7 nur mit der OS-Version 23 bzw. 24 unterstützt wird, also mit Android 6.0 bzw. 7.0. Aus diesem Grund wurde zunächst die targetSdkVersion und die compileSdkVersion der AMCS-App auf 24 geändert. Dies führte jedoch zu demselben Fehler in Firebase.

Es stellte sich erst später heraus, dass mit Hilfe des Parameters `-os-version-ids` die OS-Version der genutzten Android-Modelle eingestellt werden kann. Sollte dieser Parameter nicht explizit angegeben werden, wird dieser laut der Dokumentation von gcloud anhand des Gerätekatalogs bestimmt. Dies war, wie in Abbildung 5.4 gezeigt, das API-Level 26. Wenn man nun explizit angibt, dass die Version 24 genutzt werden soll, so wurde auch der Testdurchlauf gestartet.[107] Dies funktionierte auch, nachdem die targetSdkVersion und compileSdkVersion wieder auf 26 gestellt wurden.

Als Ersatz für das nicht vorhandene Samsung Galaxy S8 wird zunächst das Samsung Galaxy S9 mit der OS-Version 8.0 festgelegt. Das Huawei P9 wurde durch das vorhandene Huawei P8 Lite mit der OS-Version 5.0 ausgetauscht. Dieses Android-Modell wird laut Abschnitt 5.2.2 genauso oft von Nutzern der AMCS-App genutzt wie das Huawei P9 und stellt so eine Alternative zur Ausführung der Tests dar. Die entsprechenden OS-Versionen müssen auch hier übergeben werden.

Einrichtung der Testumgebung auf dem Jenkins-Server

Es gab keine Probleme bei der Einbindung von gcloud auf den Jenkins-Server. Die Installation und Einrichtung auf dem Jenkins-Server wurde von Samuel Knobloch[108] vorgenommen, da er einen Adminstratorzugriff auf den Jenkins-Server besitzt. Es wird gcloud in der Version „core 2018.06.04“ genutzt.

Nutzung mehrerer Android-Geräte

Im ersten Schritt wurde versucht, mit Hilfe der Parameter `-device-ids` und `-os-version` die Android-Geräte zu definieren. Konkret wurde dafür die Kombination `-device-ids=herolte, hwALE-H,starqlteue -os-version-ids=24,21,26` genutzt, welche die drei oben genannten Android-Modelle mit deren OS-Version darstellt. Dabei wurde immer die OS-Version angegeben, welche die aktuellste für das entsprechende Android-Modell war. Insbesondere ist zur Kenntnis zu nehmen, dass jedes der genutzten Android-Modelle nur genau eine der definierten OS-Version anbietet und umgekehrt. Dennoch führe die Ausführung von gcloud mit diesen Parametern[109] zu dem Fehler:

```
1 ERROR: (gcloud.beta.firebase.test.android.run) Http error while creating test
      matrix: ResponseError 429: Insufficient test quota. See http://firebase.
      google.com/pricing/ for details.
```

.

Diese Nachricht deutet darauf hin, dass bereits das kostenlose Kontingent von fünf Testausführungen pro Tag verbraucht wurde. Jedoch wurden an diesem Tag noch keine Test ausgeführt. Außerdem wurde mit den genutzten Parametern erwartet, dass drei Testausführungen stattfinden. Erwähnenswert ist weiterhin, dass diese Fehlermeldung erst nach 221 Sekunden ausgegeben wurde. Im Bezug mit den später in Tabelle 6.1 erhobenen Messdaten stellte sich heraus, dass dies in etwa die gleiche Zeit ist, die Firebase benötigt, um

[107] siehe Firebase - matrix-2dfphvlief80w

[108] https://tu-dresden.de/ing/informatik/sya/se/die-professur/beschaeftigte/samuel_knobloch, zuletzt besucht am 01.06.2018

[109] siehe Jenkins - nwuenscheFirebase #87

eine komplette Testausführung durchzuführen. [110]

Erst durch die Betrachtung des Beispiels in der Dokumentation[106] stellte sich heraus, dass das Kreuzprodukt der genutzten Parameter gebildet wird. Somit wurden die Tests auf den drei Android-Modellen, jedoch mit jeweils drei verschiedenen OS-Versionen ausgeführt. Dies macht in der Summe neun Testausführungen. Dies erklärt die genannte Fehlermeldung, dass zu viele Tests ausgeführt wurden. Jedoch ist an den Testausführungen zu beachten, dass von den neun (Android-Modell, OS-Version)-Kombinationen insgesamt nur drei von Firebase angeboten werden.

Zur Lösung dieses Problems wurde der Parameter `-device` genutzt. Dieser kann mehrfach genutzt werden und definiert genau ein Android-Gerät. Somit wurde im Beispiel aus `-device-ids=herolte -os-version-ids=24` der Parameter `-device model=herolte, version=24`.

Jedoch konnten die Tests nicht auf dem Samsung Galaxy S9 ausgeführt werden. Auf der Abbildung 5.5 ist zu sehen, dass selbst nach 23 Minuten die Tests noch nicht fertiggestellt worden waren.

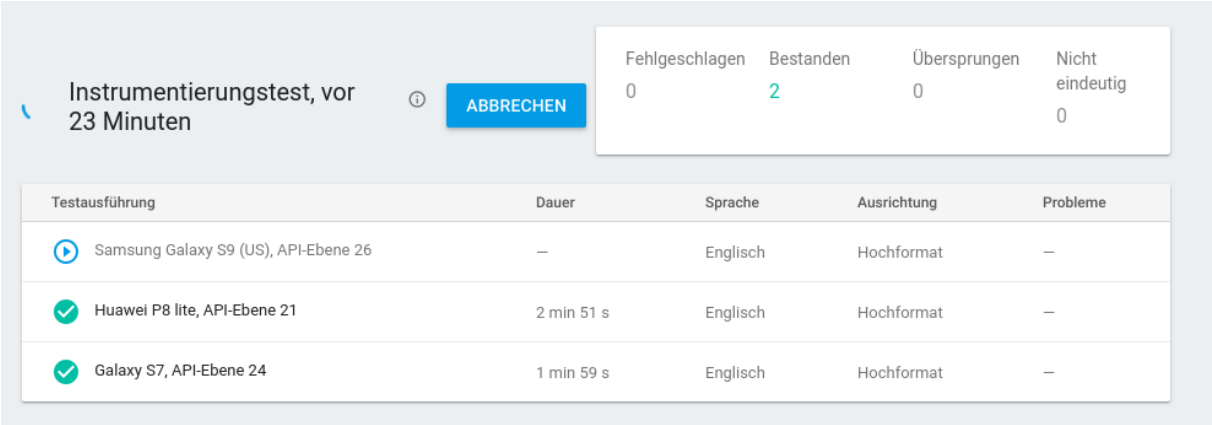


Abbildung 5.5: Samsung Galaxy S9 startet nicht (Firebase)

Aus diesem Grund wurde das S9 durch das S6 mit der OS-Version 6.0 ersetzt. Das S6 ist, wie in Abschnitt 5.2.3 gezeigt, ein weiteres Android-Modell, welches oft von Deutschen genutzt wird. Es eignet sich somit auch zur Ausführung der Tests.

Auswertung der Daten in Jenkins

Zunächst wurde eine Kopie des Master Branches mit dem Namen **nwuenscheFirebase** angelegt. Ähnlich wie bei der Implementierung der Device Cloud von Sauce Labs wurde eine Cloud Stage in das Jenkinsfile eingeführt. Auch hier wird im ersten Step zunächst die Test-APK gebaut. Im nächsten Schritt wird gcloud mit den entsprechenden Parametern ausgeführt und die Ausgabe in `output.txt` gespeichert. Im letzten Step wird ein Post-Skript ausgeführt. Dieses hat die gleiche Funktion wie bei Sauce Labs. Im Falle von Firebase wurden die Schlüsselwörter für das Fehlgeschlagen eines Tests und der Erschöpfung des Testkontingents ermittelt. Im ersten Fall wurde das Schlüsselwort „Failed“ ermittelt. Im zweiten Fall wird nach fünf Testausführungen an einem Tag die Nachricht „ResponseError 429“[111] ausgegeben.

[110] Der Autor mutmaßt, dass Firebase zunächst die Tests ausführt, bevor überprüft wird, ob die Testresultate an den Nutzer geschickt werden dürfen.

[111] siehe Jenkins - nwuenscheFirebase #101

Auswertung der detailierten Testresultate

Firebase bietet auf ihrer Webseite detailierte Testresultate zu jeder Testausführung an. Insbesondere kann für jeden fehlgeschlagenen Test direkt die geworfene Exception betrachtet werden.

Ausführung aller ermittelten Android-Geräte

Die Ausführung von gcloud mit drei *–device* Parametern führte dazu, dass die Ausführung des gcloud-Befehls[112] mit rund 14 Minuten ungefähr dreimal so lange dauerte wie mit einem Gerät. Dies kann den später ermittelten Messwerten aus Tabelle 6.1 entnommen werden. Es wird geschlussfolgert, dass die Testausführungen nacheinander und nicht parallel abgearbeitet wurden.

Ergebnisse der Tests

Während der Implementierung des Prototypen konnten mehrere gescheiterte Testausführungen beobachtet werden. So stellte sich konkret bei der Nutzung des Android-Gerätes Pixel 2 mit der OS-Version 8.0 heraus, dass der Test „addCourseAndDeleteIt" fehlgeschlagen ist.[113] Dieser Test überprüft, ob es möglich ist, einen neuen Kurs innerhalb des AMCS-Projekts anzulegen und danach zu löschen. Im Rahmen dieser Arbeit wurde das Testresultat nicht näher analysiert.

5.7 Bitbar

Im Folgenden wird die Implementierung der Device Cloud von Bitbar in das AMCS-System untersucht. Dies wird mit Hilfe des Jenkins-Plugins und der entsprechend angebotenen Dokumentation[80] durchgeführt. Es sei angemerkt, dass seit Mitte Mai 2018 auch eine Dokumentation[114] für die direkte Einbindung in einem Jenkinsfile vorhanden ist. Diese Dokumentation wurde erst nach Beginn der Implementierungsphase veröffentlicht, sodass die Implementierung mit dem Jenkins-Plugin stattfindet.

Erwerb einer Testlizenz

Wie bereits erläutert, wurde statt einer Testlizenz ein kostenloses Forschungskonto mit vollem Funktionsumfang genutzt. Insofern fällt die Betrachtung zum Erwerb der Testlizenz weg.

Lokale Einrichtung der Testumgebung

Eine lokale Einrichtung war nicht möglich, da das Bitbar-Plugin nicht ohne eine Installation von Jenkins genutzt werden kann. Somit wurde direkt eine Implementierung mit dem Jenkins-Servers vorgenommen.

[112] siehe Jenkins - nwuenscheFirebase #102
[113] siehe Jenkins - nwuenscheFirebase #61
[114] http://docs.Bitbar.com/testing/cloud-integrations/jenkins-plugin/pipeline/index.html, zuletzt besucht am 04.06.2018

Auswahl der Geräte

Das in Tabelle 5.1 benannte Samsung Galaxy S7 steht bei Bitbar nicht zur Verfügung. Dies kann auf der Webseite von Bitbar unter der Rubrik „Devices“ überprüft werden.

Es wurde daher auf das Samsung Galaxy S7 Edge ausgewichen, da, wie in Abschnitt 5.2.3 benannt wurde, dieses Android-Modell auch von vielen Smartphone-Nutzer in Deutschland genutzt wird. Das Samsung Galaxy S8 sowie das Huawei P9 sind indes vorhanden. Man erfährt die OS-Versionen der einzelnen Android-Geräte nur dann, wenn man alle verfügbaren Geräte nach den OS-Versionen filtert und dadurch entnimmt, welches Android-Gerät welche OS-Version nutzt. Dafür wurden für das Samsung Galaxy S7 Edge die Version 6.0.1, für das S8 die Version 8.0.0 sowie für das Huawei P9 die OS-Version 6.0 ermittelt.

Einrichtung der Testumgebung auf dem Jenkins-Server

Die Nutzung des Bitbar-Plugins verläuft über ein Freestyle-Projekt. Dieses heißt im folgenden **nwuenscheBitBarPlugin**. Das Plugin trägt den Namen **Testdroid Plugin for CI**, wird in der Version 1.0.22 installiert und es wird die offizielle Anleitung von Bitbar[80] genutzt.

Im nächsten Schritt wurde über die Webseite von Bitbar ein neues Test-Projekt für die AMCS-App angelegt.

In der Konfiguration des Freestyle-Projektes wurde als Erstes in der Kategorie „Source-Code-Management“ eine Verbindung mit dem Bitbucket Server hergestellt. Um den Branch festzulegen, der gebaut werden soll, musste zunächst ein entsprechender neuer Branch im AMCS-Projekt mit dem Namen **nwuenscheBitBar** angelegt und dieser Name hier eingegeben werden.

In der Kategorie „Buildverfahren“ wurden drei Steps hinzugefügt. Diese sind stark an die Steps in den anderen beiden genutzten Device Clouds angelehnt. Zunächst wird im ersten Step ein „Shell ausführen“-Befehl hinzugefügt. Dieser dient der Ausführung von Shell-Skripten und baut in diesem Fall die Test-APK. Als zweiter Step wird ein „Testdroid: Run tests in Testdroid Cloud“-Befehl hinzugefügt. Dabei wurde im Besonderen bei der Eigenschaft „Scheduling“ der Wert „Run on all devices simultaneously“ ausgewählt. Eine parallele Ausführung der Tests auf verschiedenen Geräten ist somit möglich. Des Weiteren wurde die Eigenschaft „Wait for test finished in Testdroid Cloud and download results“ ausgewählt und im folgenden Fenster der „API_CALL“ ausgewählt. Eine „HOOK_URL“ kann nicht genutzt werden, da der Jenkins-Server nicht öffentlich zugänglich ist. Somit muss eine Bitbar-API vom Jenkins-Server aufgerufen werden, um zu überprüfen, ob die Tests bereits vollendet worden sind. Der letzte Step ist die Ausführung des Post-Skriptes, welches im nächsten Abschnitt genauer betrachtet wird. Abschließend wird dem Jenkinsfile des Branches nwuenscheBitBar eine neue Stage mit einem einzigen Build-Step hinzugefügt.

```
1         stage ('Cloud'){
2             ...
3             steps {
4                     build 'nwuenscheBitBarPlugin'
5             }
6         }
```

Auswertung der Daten in Jenkins

Bei der Ausführung des Bitbar Plugins werden, anders als bei Sauce Labs und Firebase, die Testresultate nicht in der Konsole ausgegeben. Stattdessen muss nach der Ausführung der Tests eine API angesprochen werden. Zur Nutzung dieser wird eine Dokumentation[115] von

[115] http://docs.Bitbar.com/testing/api/, zuletzt besucht am 04.06.2018

Bitbar unter Zuhilfenahme des Programmes **curl**[116] angeboten. Es ist zu beachten, dass der API-Schlüssel mit einem Doppelpunkt beendet werden muss, da sonst eine Passwort-Abfrage gestartet wird. Die genutzte curl Version ist 7.52.1 .

```
1 DCResults=$(curl ...) #Lade alle Testausfuehrung eines Projektes
2 numberOfFailedTests=$(echo $DCResults | python3 -c "import sys, json; print(json
      .load(sys.stdin)['data'][0]['failedTestCaseCount'])") #Anzahl der
      fehlgeschlagenen Tests

4 if [ $numberOfFailedTests != 0 ]; then
5     echo "Failed Tests!"
6     exit 1
7 fi
```

In der ersten Zeile werden alle Testresultate des Projektes aus Bitbar in der Variable `DCResults` gespeichert. Die Form dieser Ausgabe ist JSON[117]. Im nun folgenden Schritt wird dieses JSON-Objekt mit Hilfe eines Skriptes ausgewertet. Dieses wurde in der Programmiersprache Python[118] geschrieben. Das Python-Programm gibt dabei aus, wie viele Tests bei der letzten(entspricht `[0]`) Testausführung des Projektes fehlgeschlagen(entspricht `['failedTestCaseCount']`) sind. Diese Zahl wird in der Variable `numberOfFailedTests` gespeichert. Auf dem Jenkins-Server wird die Python Version 3.5.3 genutzt.

In Zeile 4 wird untersucht, ob die Zahl gescheiterter Tests ungleich „0“ ist. Sollte sie ungleich „0“ sein, ist ein Test fehlgeschlagen und es wird der Wert „1“ vom Skript zurückgegeben. Auch diese Abfrage wurde mit dem Test alwaysFail überprüft und verifiziert.

Auswertung der detailierten Testresultate

Zunächst bietet Bitbar die Funktion an, dass neben den Builds der Jenkins-Pipeline das Logo von Bitbar erscheint. Wird dieses ausgewählt, soll man zu den detailierten Testresultaten auf der Webseite von Bitbar weitergeleitet werden. Es wird jedoch bei jedem Build auf eine Webseite verwiesen, welche nicht gefunden werden kann und die zu einem HTTP 404 Fehler führt. Um die Testresultate einzusehen, muss man also den Weg über die Bitbar Seite gehen.

Die detailierten Testberichte bieten sich, ähnlich wie bei Firebase, zur Auswertung gescheiterter Tests an, da die Exceptions direkt bei den entsprechenden Tests angezeigt werden.

Ausführung aller ermittelten Android-Geräte

Die gleichzeitige Ausführung der Tests auf allen ausgewählten Android-Geräten ist bei Bitbar möglich und kann bei der Konfiguration des Freestyle-Projektes aktiviert werden.

Ergebnisse der Tests

Während der Implementierung des Prototypen konnten mehrere gescheiterte Testausführungen beobachtet werden. So stellte sich beispielsweise bei der Nutzung des Android-Gerätes Samsung Galaxy S8 mit der OS-Version 8.0.0 heraus, dass über mehrere Testausführungen hinweg insgesamt 15 der Integration Tests gescheitert sind.[119] Wie jedoch bereits im Konzept geschrieben wurde, setzt sich diese Arbeit nicht mit dem Lösen der Bugs

[116] https://curl.haxx.se/, zuletzt besucht am 04.06.2018
[117] https://www.json.org/json-de.html, zuletzt besucht am 04.06.2018
[118] https://www.python.org/, zuletzt besucht am 04.06.2018
[119] siehe Jenkins - nwuenscheBitBar #33

auseinander, sondern ausschließlich mit der Integration der Device Clouds in das AMCS-System.

Nachdem nun alle selektierten Device Clouds prototypisch in das AMCS-System eingebunden wurden, wird im nächsten Kapitel eine detailierte Auswertung weiterer Anforderungen stattfinden.

6 Evaluation

Das folgende Kapitel beurteilt die in dieser Arbeit ausgearbeiteten Ergebnisse. Dabei werden die Anforderungen A1, A5, A6 und A7 genauer betrachtet und die ermittelten Daten der Device Clouds untereinander verglichen.

6.1 Messung der Zeit zum Ausführen von Tests

Zunächst werden die implementierten Device Clouds anhand der Anforderung A1 beurteilt.

6.1.1 Erhebung der Messdaten

Zur Bewertung der Anforderung A1 wurde bereits ein Bewertungskriterium in Abschnitt 2.4.1 definiert. Es soll die Zeit gemessen werden, welche zwischen der Testausführung und der Übermittlung der Testresultate vergeht.

Dementsprechend wurden nach dem Konzept aus Kapitel 4 die gleichen Rahmenbedingungen geschaffen. Zunächst wurde darauf geachtet, dass alle Device Clouds den gleichen Stand des Quelltextes untersuchen. Dies ist bei dieser Untersuchung der aktuelle Stand des Masters[120]inklusive der in Abschnitt 5.4.1 vorgestellten Änderung des Context-Objektes. Das Jenkinsfile und Post-Skript des entsprechenden Branches sind jedoch auf die entsprechende Device Clouds angepasst.

Es wurde sich dafür entschieden, vom Konzept abzuweichen. Da, wie im Abschnitt 3.2.3 erläutert, Jenkins nicht direkt benachrichtigt wird, wenn ein neuer Commit auf Bitbucket vorliegt, ist die Zeit zwischen dem Push und der entsprechenden Benachrichtigung sehr stark von Jenkins abhängig, und nicht von den Device Clouds. So können zwischen einem Commit auf Bitbucket und dem Starten der Pipeline bis zu 15 Minuten vergehen. Es wurde zudem auf eine Implementierung des in Abschnitt 4.1.3 vorgestellten Systems verzichtet, nachdem man Code erst pushen dürfte, nachdem der Jenkins-Server die Tests über diesem Code ausgeführt hat. Diese Entscheidung wird damit begründet, dass zurzeit nur eine Person an der unveröffentlichten AMCS-App arbeitet. Es wäre hinderlich, jeden Commit vor dem Push zunächst von Jenkins überprüfen zu lassen. Der aktuellen Zustand der entsprechenden Pipeline kann auch schon jetzt mit Hilfe des Jenkins Plugins Bitbucket Build Status Notifier eingesehen werden.

Die Erhebung der jeweils fünf Messdaten erfolgt durch das wiederholte Bauen der entsprechenden Pipeline. Dazu wird der Button „Jetzt bauen“ in Jenkins genutzt. Erst nachdem

[120]Commitnummer: 282cf07d3a8798a5049bc488b54d5b3862812e3b

ein Durchlauf der Pipeline vollendet ist, wird der nächste gestartet. Nach der Ausführung der Pipeline wird in Jenkins betrachtet, wie lange der Step zur Ausführung der entsprechenden Device Cloud dauerte. Dies wäre bei Sauce Labs die Ausführung der Runner-Datei, bei Firebase die Ausführung von gcloud und bei Bitbar der Build-Step zum Ausführen des Freestyle-Projektes.

Im Besonderen sei darauf hingewiesen, dass bei Bitbar die Ausführungszeit des Post-Skriptes in die entsprechenden Messwerte einbezogen wird, während dieses bei Firebase und Bitbar nicht betrachtet wird. Dieser Schritt wurde gewählt, da bei Firebase und Sauce Labs die Testresultate bereits bei der Ausführung des entsprechenden Konsolenbefehls dargestellt werden, während diese bei Bitbar erst durch die Kommunikation mit der Bitbar-API innerhalb des Post-Skriptes heruntergeladen werden.

Des Weiteren werden Messdaten für die aktuelle Ausführung von Integration Tests erhoben. Wie bereits in Abschnitt 3.3 betrachtet wurde, handelt es sich dabei um die Ausführung der Tests auf einem Emulator, welcher auf dem Jenkins-Server läuft.

Um die Messdaten zu erheben, wurde zunächst das Freestyle-Projekt zum Starten des Emulators unter dem Namen **nwuenscheAndroidUIIntegration** kopiert. Des Weiteren wurde eine weitere Kopie des Master Branches mit dem Namen **nwuenscheEmulator-Test** erstellt. In diesem Branch wurde auch das in Abschnitt 5.4.1 benannte Context-Objekt abgeändert. Im Zuge dessen musste der zu bauende Branch im Freestyle-Projekt auf den Namen des neuen Branch angepasst werden. Analog wurde der Name des Freestyle-Projektes im Build-Step des Branches angepasst.

6.1.2 Rahmenbedingungen der Erhebung

Bei der Erhebung der Messdaten wurde darauf geachtet, dass eine möglichst gleiche Umgebung bei allen Device Clouds vorhanden ist. Es wurde darauf geachtet, dass die Rahmenbedingungen zwischen den Device Clouds so ähnlich wie möglich sind. Zunächst wurde ein Android-Gerät identifiziert, welches von allen Device Clouds angeboten wird. Da keines der in Tabelle 5.1 definierten Android-Modelle in allen Device Clouds vorhanden war, wurde ein neues Modell gesucht, welches von allen Device Clouds angeboten wird. Die Untersuchung wurde bei Sauce Labs gestartet, da diese mit fünf angebotenen Android-Geräten die wenigsten Geräte im Rahmen der drei Device Clouds anbietet. Abschließend wurde sich für das Android-Modell Samsung Galaxy S6 entschieden. Dieses Modell wird sowohl von Sauce Labs, als auch von Firebase und Bitbar als echtes Android-Modell angeboten. Ferner ist es, wie in Abschnitt 5.2.3 betrachtet, eines der am meisten genutzten Android-Modelle in Deutschland. Auf eine parallele Ausführung von Tests auf verschiedenen Android-Geräten wurde verzichtet, da Sauce Labs und Firebase diese Funktion nicht anbieten.

Während bei Bitbar und Sauce Labs das Samsung Galaxy S6 mit der OS-Version 7.0 angeboten wird, wird bei Firebase das gleiche Modell mit der OS-Version 5.1 oder 6.0 zur Verfügung gestellt. Es wurde sich bei Firebase für die Version 6.0 entschieden, da diese im Bezug auf die Aktualität näher an der von Bitbar bzw. Sauce Labs angebotenen OS-Version ist.

Die Parameter des Emulators, welche in Abschnitt 3.3 vorgestellt wurden, werden dabei so belassen, wie sie zurzeit tatsächlich genutzt werden. Dieser Schritt wurde gewählt, da die Ausführungszeiten der Device Clouds mit der tatsächlich zurzeit genutzten Methode im AMCS-Projekt verglichen werden sollen. Eine Anpassung des Emulators würde so den Vergleich der Messdaten zwischen der aktuellen Implementierung und den Device Clouds verfälschen.

6.1.3 Nachbereitung der Messdaten

Es stellte sich nach der Erhebung der Testdaten heraus, dass bei der Betrachtung des Freestyle-Projektes von Bitbar sowie des Emulators vergessen wurde, die Steps zum Bauen der APK und Test-APK aus dem Freestyle-Projekt zu entfernen und als Step in den Jenkinsfile zu verschieben. In Jenkins kann man leider nur die Ausführungszeiten einzelner Steps eines Pipeline-Projektes einsehen, nicht jedoch die eines Freestyle-Projektes.

Um die Messdaten zwischen den Device Clouds möglichst vergleichbar zu halten, werden dementsprechend die Zeiten zum Bauen der APK und Test-APK aus den Messdaten entfernt.

Um zu ermitteln, wie viel Zeit das Bauen der Test-APK in Anspruch nahm, wird die Konsolenausgabe der korrespondierenden Ausführung des Freestyle-Projektes betrachtet. Hierbei wird nach dem Bauen der APKs beispielsweise die Meldung `BUILD SUCCESSFUL in 1s` dargestellt, wenn das Bauen der APK eine Sekunde benötigt hat. Es ist darauf zu achten, dass man insgesamt zwei dieser Nachrichten für das Bauen der APK und Test-APK erhält. Diese Zeiten wurden von den betreffenden Messdaten von Bitbar und des Emulators abgezogen.

Des Weiteren wurde bei den Device Clouds Firebase und Sauce Labs vor Erhebung der Daten vergessen, das Programm `tee` aus den sh-Steps zu entfernen. Die Ausführungszeit für dieses Programm kann nicht mehr aus den Testdaten herausgerechnet werden. Es wird jedoch als vernachlässigbar angesehen.

6.1.4 Auswertung der Messdaten

Device Cloud	**V1 in s**	**V2 in s**	**V3 in s**	**V4 in s**	**V5 in s**	$\overline{v}$ **in s**	δ_v **in s**
Sauce Labs [121]	191	221	221	223	222	215	12
Firebase[122]	451	274	276	310	259	314	71
Bitbar[123]	265	203	202	265	264	240	31
Emulator Jenkins[124]	864	932	848	864	878	877	29

Tabelle 6.1: Auswertung der Messdaten zur Testausführungszeit

In der Tabelle 6.1 stellt beispielsweise die Spalte V1 den ersten Messwert jeder Versuchsreihe dar. Außerdem bezeichnet $\overline{v}$ das arithmetische Mittel der Stichprobe und δ_v die Standardabweichung (mit n Werten) der Messwerte einer Device Cloud.

Wie man aus der obigen Abbildung zunächst ablesen kann, dauert die Ausführung der Tests auf dem Emulator bis zu viermal länger als auf den Device Clouds. Die Standardabweichung von Firebase ist mit 71 Sekunden fast sechsmal höher als die von Sauce Labs. Die hohe Standardabweichung bei Firebase kommt vor allem durch den hohen Messwert im ersten Versuch zustande. Im Verlauf des nächsten Abschnittes werden die Standardabweichungen der Device Clouds noch einmal genauer betrachtet und untersucht.

Die Device Cloud, die die Anforderung A1 somit am besten erfüllt, ist die Sauce Labs Cloud mit 215 Sekunden. Jedoch ist jede der betrachteten Device Clouds deutlich schneller als die Ausführung der Tests auf einem Emulator. Dies kann dadurch begründet werden, dass, wie bereits in Abschnitt 2.1.3 formuliert wurde, Emulatoren langsamer als entsprechende Android-Geräte sind. Weiterhin ist auch die Tatsache, dass KVM auf dem Jenkins-

[121] aus Jenkins - nwuenscheSauceLabs #40-#44
[122] aus Jenkins - nwuenscheFirebase #71, #72, #74-#76
[123] aus Jenkins - nwuenscheBitBar #19, #21-#24
[124] aus Jenkins - nwuenscheEmulatorTest #8, #12-#15

Server nicht genutzt wird (siehe Abschnitt 3.3), dafür verantwortlich, dass der Emulator langsam läuft.

6.2 Verschickte Pakete

Im Folgenden werden die Prototypen der Device Clouds anhand der Anforderung A7 beurteilt. Alle genannten Log-Dateien können auf Bitbucket[125] eingesehen werden.

6.2.1 Erhebung der Daten

Zur Bewertung der Anforderung A7 wurde das Bewertungskriterium definiert, dass die Pakete, welche zwischen den Device Clouds und dem Jenkins-Server ausgetauscht werden, genauer untersucht werden.

Um das in Kapitel 4 ausgearbeitete Konzept umzusetzen, wurde während der Erhebung der Messdaten im vorherigen Abschnitt bei jeder Device Cloud einmal die Kommunikation aufgezeichnet.

Um die Kommunikation zu erfassen, wurde zunächst über eine VPN- und SSH-Verbindung auf den Jenkins-Server zugegriffen. Die Kommunikation wurde mit dem Netzwerk-Analysetool **tshark**[126], in der Version 2.2.6 ausgeführt. Dabei wird mit Hilfe des Parameters `-w` die anfallende Kommunikation in eine Datei(**Log**) gespeichert. Die Untersuchung dieser Datei wurde mit dem Netzwerk-Analysetool **Wireshark**[127] vollzogen.

Die Rahmenbedingungen sind äquivalent zu denen im vorherigen Abschnitt beschriebenen. Demzufolge wurde auch für diese Betrachtung auf das Samsung Galaxy S6 ausgewichen.

6.2.2 Anpassung der Bewertungskriterien

Nachdem die Logs der einzelnen Device Clouds aufgezeichnet wurden, wurde der Inhalt der einzelnen Pakete analysiert. Bevor die eigentliche Analyse erläutert wird, sei darauf hingewiesen, dass bei den drei untersuchten Logs zunächst eine Verschlüsselung der Pakete mit TLS in der Version 1.2 (vgl. (Rescorla & Dierks, 2008)) stattfand.

Um eine Entschlüsselung der Pakete zu ermöglichen und so den Inhalt der Pakete genauer zu analysieren, wurde nach (Iveson, 2013) versucht, den privaten RSA-Schlüssel in Wireshark zu importieren. Jedoch führte dies nicht zur erhofften Entschlüsselung der Pakete.

Später stellte sich heraus, dass eine weitere Verschlüsselung zwischen dem Jenkins-Server und den Device Clouds stattfindet. Dies ist eine Verschlüsselung der Art Diffie-Hellman (**DH**). Jedoch wird auf der Webseite von Wireshark folgendes geschrieben: „Decoding an SSL connection requires either knowledge of the (asymmetric) secret server key and a handshake that does not use DH or the (base of) the symmetric keys used to run the actual encryption.“[128]. Die gleiche Information wird von (Solnica, 2016) sowie Citrix[129] angeführt.

Da DH genutzt wird, kann der Inhalt der Pakete nicht betrachtet werden. Bei der Analyse der entsprechenden Logs der Device Clouds wird genau angegeben, wo die Information, dass DH genutzt wird, zu finden ist.

[125] https://bitbucket.org/NWuensche/ba-enclosures/src, zuletzt besucht am 07.06.2018
[126] https://www.wireshark.org/docs/wsug_html_chunked/AppToolstshark.html, zuletzt besucht am 05.06.2018
[127] https://www.wireshark.org/, zuletzt besucht am 04.06.2018
[128] https://wiki.wireshark.org/SSL, zuletzt besucht am 31.05.2018
[129] https://support.citrix.com/article/CTX116557, zuletzt besucht am 31.05.2018

Zunächst wurde versucht, mehr Informationen zu der auftretenden Kommunikation aus dem von Jenkins bereitgestellten Systemlogs zu erhalten. Jedoch konnte dort für alle drei Device Clouds nur entnommen werden, dass das Bitbar Plugin aller 60 Sekunden den Status der Testausführung mit der Nachricht „Check for testRun(150,802,022) state." abfragt. Alle drei Device Cloud Anbieter veröffentlichen weiterhin keine technische Dokumentation zu der entsprechenden Kommunikation.

Aus diesem Grund wurde sich mit der Frage beschäftigt, wie viele Pakete mit der Beschreibung „Application Data(AD)" über das Protokoll TLS bei der Testausführung verschickt wurden, wie groß diese Pakete waren und welche Zeiten bei der Kommunikation von Jenkins-Server und Device Cloud vergingen. Dabei wurden die AD-Pakete gewählt, da diese den verschlüsselten Inhalt der Kommunikation beinhalten (vgl. (Rescorla & Dierks, 2008)). Um den Verlauf der Kommunikation zu betrachten, wurden alle Pakete betrachtet, welche das Protokoll TLS in der Version 1.2 nutzten. Davor mussten jedoch für die Kommunikation zunächst die IPs identifiziert werden, welche die Device Clouds repräsentierten. Dabei wurde der Netzwerkverlauf auf wiederkehrende IPs und deren Subnetz überprüft. Die IP des Jenkins-Server ist über alle Logs hinweg 141.76.19.69 .

Im weiteren Verlauf dieses Abschnittes werden diverse Mutmaßungen über gewisse Eigenschaften der Kommunikation vom Autor benannt. Diese Mutmaßungen sind als solche angeführt und wurden so gut wie möglich mit Fakten bekräftigt. Jedoch kann ohne eine Entschlüsselung der Pakete nicht eindeutig bestätigt werden, ob all diese Mutmaßungen zutreffend sind.

Es wurde sich dagegen entschieden, die Netzwerkauslastung des Emulators zu betrachten. Dies hat den Grund, dass die Kommunikation bei der Nutzung von Device Clouds und Emulatoren zu unterschiedlich ist, als das man sie vergleichen könnte. Beispielsweise muss für das erstmalige Erstellen des Emulators ein „System Image"[130] herunterladen werden. Man müsste nun sowohl die Kommunikation des Jenkins-Server bei der Ausführung des Emulators betrachten, wenn das System Image heruntergeladen wird, aber auch, wenn es bereits heruntergeladen wurde. Es müsste weiterhin überprüft werden, welche geladenen Dateien explizit für die Bereitstellung des Emulators benötigt werden und welche nur für das Bauen der APK genutzt werden. Diese Analyse ist jedoch im Rahmen dieser Arbeit nicht vorgesehen, so dass darauf verzichtet wird.

Die untersuchte Log-Datei für Sauce Labs hat den Namen logSauceSam6New, für Firebase den Namen logFire2Sam6 und für Bitbar den Namen logBit3Sam6. Sie können auf Bitbucket[125] gefunden werden.

6.2.3 Sauce Labs

Ermittelung der IPs[131]

Die IP, nach der die Pakete gefiltert werden, ist 185.94.24.39. Diese IP wurde ermittelt, indem überprüft wurde, welche IP nach der Ausführung der Runner-Datei angesprochen wurde. Dazu wurde ein Paket mit der Info „Client Hello" gesucht. Dies war im Paket 1392 der Fall. In diesem Paket konnte zudem die Zeichenkette „app.testobject.com" gefunden werden. Wie bereits in Abschnitt 5.5 erläutert, wurde die Firma TestObject von Sauce Labs übernommen und ist somit nun ein Teil von Sauce Labs.

Somit wurde der Log in Wireshark mit dem Befehl `ip.addr == 185.94.24.39` gefiltert, um nur Pakete zwischen der Kommunikation des Jenkins-Servers und Sauce Labs zu betrachten. Jedoch wird im Paket 1396 deutlich, dass eine Verschlüsselung zwischen Sauce

[130] vgl. https://developer.android.com/studio/run/managing-avds, zuletzt besucht am 07.06.2018
[131] vgl. Jenkins - nwuenscheSauceLabs #48

Labs und Jenkins mit der Verschlüsselungsart Diffie-Hellman (DH) genutzt wird.

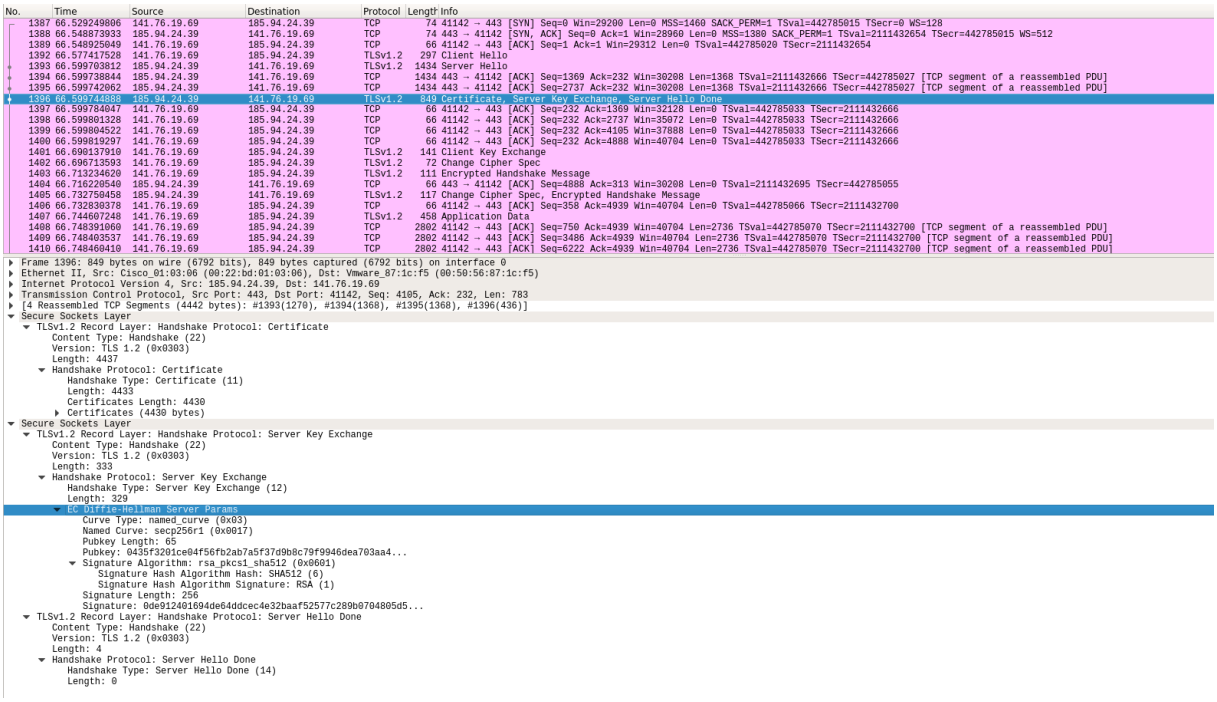


Abbildung 6.1: Diffie-Hellman Verschlüsselung (Sauce Labs)

Auswertung der Pakete

Im Folgenden wurden zunächst die Längen der AD-Pakete ausgewertet. Dazu wurden unter Wireshark unter „Statistiken → Paketlängen“ mit dem Filter `ip.addr == 185.94.24.39 and ssl.record.content_type == 23` nur die AD-Pakete der Kommunikation betrachtet. Insgesamt wurden dabei 912 AD-Pakete mit einer durchschnittlichen Länge von 9508,71 Byte gemessen. Aus diesen beiden Zahlen ergibt sich somit eine Gesamtlänge aller AD-Pakete von rund 8671944 Byte.

Verlauf der Kommunikation

Um alle Pakete zu sehen, welche das Protokoll TLS in der Version 1.2 nutzen, wurde der Filter durch `ip.addr == 185.94.24.39 and ssl.record.version == 0x0303` ersetzt. Hierbei stellt 0x0303 die Version 1.2 von TLS dar, welche bei der Verschlüsselung der Pakete bei der aufgezeichneten Kommunikation genutzt wurde.

Die Kommunikation beginnt hierbei zum Zeitpunkt (gerundet auf ganze Sekunden) von 67 Sekunden mit dem Paket 1392 mit einem Client Hello. Bis zum Zeitpunkt von 71 Sekunden werden ohne große Pausen Pakete ausgetauscht. Diese Kommunikation endet mit dem Paket 4965. Da auch in der Konsolenausgabe der ausgeführten Runner-Datei zwischen der Kontaktaufnahme des Jenkins-Servers („InstrumentationTestRunner 1.1 initialized.“) und dem Ausführen der Tests („Starting test...“) in etwa vier Sekunden vergehen, ist davon auszugehen, dass in dieser Zeit nur die APK und Test-APK vom Jenkins auf den Server von Sauce Labs hochgeladen werden. Diese Phase wird von nun an als **Upload-Phase** bezeichnet.

Zum Zeitpunkt von 101 Sekunden, also 30 Sekunden nachdem die Upload-Phase beendet wurde, wird ein weiteres Client Hello in Paket 5712 verschickt. Es werden innerhalb

der nächsten 0,2 Sekunden zehn Pakete verschickt, wobei zwei davon AD-Pakete sind. Die Phase endet mit einem Encrypted Alert - Paket mit der Nummer 5733.

Der Autor mutmaßt, dass der Jenkins-Server in dieser Phase die Device Cloud kontaktiert, um zu überprüfen, ob bereits alle Tests auf der Device Cloud ausgeführt wurden. Jedoch findet in dieser Zeit keine Konsolenausgabe der Runner-Datei statt. Diese Mutmaßung wird dadurch bekräftigt, dass in der Dokumentation[60] der Runner-Datei geschrieben wird, dass wenn der Parameter `checkFrequency` nicht gesetzt ist, aller 30 Sekunden überprüft wird, ob die Testausführung beendet ist. Das letzte Paket, was in dieser Phase verschickt wird, ist das Paket 5733. Diese Phase wird auf Grund der Mutmaßungen des Autors als **Pull-Phase** bezeichnet.

Zwischen den Zeitpunkten 101 Sekunden und 282 Sekunden findet aller 30 Sekunden eine solche Pull-Phase statt. Die letzte Phase, welche zum Zeitpunkt 282 Sekunden mit dem Paket 11013 beginnt, teilt sich in zwei Unterphasen. Nachdem die erste Unterphase, welche wie eine Pull-Phase strukturiert ist, in Paket 11039 mit einem Encrypted Alert endet, wird bereits 0,02 Sekunden später ein erneutes Client Hello mit der Nummer 11044 verschickt. Der Autor mutmaßt, dass während der ersten Unterphase dem Jenkins-Server signalisiert wurde, dass alle Tests ausgeführt wurden. In der zweiten Unterphase wurden wahrscheinlich die Testresultate von der Device Cloud an den Jenkins-Server gesendet. Für diese Mutmaßung spricht die Tatsache, dass zwischen dem Zeitpunkt der ersten Kontaktaufnahme (67 Sekunden) und dem letzten gesendeten Paket (283 Sekunden) insgesamt 216 Sekunden vergangen sind. Zwischen den Konsolenausgaben „Instrumentation-TestRunner 1.1 initialized.“ und „Test completed.“ der Runner-Datei in Jenkins vergehen 217 Sekunden. Somit stimmen diese Zeiten fast überein. Diese letzte Phase wird deshalb als mutmaßliche **Resultat-Phase** bezeichnet.

6.2.4 Firebase

Ermittelung der IPs[132]

Zunächst werden mehrere verschiedene IPs zur Kommunikation mit Firebase genutzt. Der Autor mutmaßt, dass die APK und Test-APK an die IP 172.217.16.202 versendet werden. Dies wurde anhand der großen Menge an AD-Paketen festgemacht, welche zu Beginn der Kommunikation an diese IP gesendet wird.

Die Überprüfung des Stands der Testausführung wird mit der IP 216.58.206.10 kommuniziert. Diese IP wird, wie später dargestellt wird, in der mutmaßlichen Pull-Phase kontaktiert. Der Autor mutmaßt, dass die eigentlichen Testresultate von der IP 216.58.208.42 gesendet werden, da diese in der mutmaßlichen Release-Phase angesprochen wird. Somit wird zur Auswertung der Teil des Filters von Sauce Labs, welcher die IPs filtern, in `ip.addr == 172.217.16.202 or ip.addr == 216.58.206.10 or ip.addr == 216.58.208.42` verändert. Als Belege für die IPs sei hier angeführt, dass im Client Hello zur IP 172.217.16.202 im Paket 1476 die Zeichenkette www.googleapis.com gefunden wurde, bei dem entsprechenden Client Hello an die IP 216.58.206.10 die Zeichenkette testing.googleapis.com und bei der dritten IP wieder die Zeichenkette www.googleapis.com.

Zunächst gilt auch bei der Auswertung der Daten bei Firebase, dass eine Entschlüsselung der Daten nicht möglich ist, da im Paket 1499 wieder ein Schlüssel nach Diffie-Hellman ausgetauscht wurde.

[132] vgl. Jenkins - nwuenscheFirebase #74

Auswertung der Pakete

Es wurde insgesamt 391 AD-Pakete mit einer durchschnittlichen Länge von 22178,38 Byte verschickt. Somit beträgt die Gesamtpaketlänge rund 8671739 Byte. Die einzelnen Arten von Phasen von Firebase scheinen, laut Vermutung des Autors, stark mit denen von Sauce Labs übereinzustimmen. Jedoch wird das erste Client Hello in Paket 1476 zunächst mit TLS Version 1.0 gesendet, sodass der Filter für dieses Paket auf `... and ssl.record.version == 0x0301` angepasst werden muss. Spätere Client Hello Pakete nutzen auch TLS 1.0, jedoch wird immer ab dem Server Hello, beispielsweise das erste gesendet Server Hello in Paket 1478, TLS in der Version 1.2 genutzt.

Verlauf der Kommunikation

Zunächst wird die Upload-Phase bei Firebase laut Meinung des Autors von den Paketen 1477 und 5269 eingegrenzt. Sie findet zwischen den Zeitpunkten 58 Sekunden und 61 Sekunden statt. Der Autor kommt zu diesem Schluss, da in dieser Zeit viele AD-Pakete an die IP 172.217.16.202 gesendet werden. In der Konsolenausgabe von gcloud in Jenkins werden keine Zeitstempel an das Hochladen der APK und Test-APK gesetzt.

Weniger als 0,1 Sekunden später wird mit dem Paket 5278 ein Client Hello an die IP 216.58.206.10 versendet. Dies ist auch das erste und einzige Mal, dass ein Client Hello im Bezug auf diese IP versendet wird. Der Autor vermutet, dass die Verbindung über die komplette Zeit der Testausführung aufrechterhalten wird.

Zwischen den Zeitpunkten 61 Sekunden und 74 Sekunden werden verschiedene AD-Pakete mit dieser IP ausgetauscht. Der Autor vermutet, dass dies unter anderem die URLs für die anstehenden Testresultate sind, welche von gcloud ausgegeben werden. Dies kann aufgrund fehlender Zeitstempel in der Konsolenausgabe jedoch nicht bestätigt werden.

Ab dem Zeitpunkt 86 Sekunden, beginnend mit dem Paket 7291, bis zu dem Paket 13288 zum Zeitpunkt 317 Sekunden, werden aller zwölf Sekunden vier AD-Pakete ausgetauscht. Der Autor vermutet, dass es sich hier um die Pull-Phasen handelt, da die Kommunikation mit zwölf Sekunden Pause sehr regelmäßig ausfällt und stark an die Pull-Phase bei Sauce Labs erinnert. In der vermeintlich letzten Pullphase (zwischen den Paketen 13443 und 13469) werden jedoch statt vier AD-Paketen insgesamt acht solche Pakete ausgetauscht. Sie endet zum Zeitpunkt 330 Sekunden. Nachdem die letzte Pull-Phase abgeschlossen ist, wird rund 0,1 Sekunden später eine Verbindung mit der IP 216.58.208.42 in Paket 13482 aufgebaut. Es werden zwischen den Zeitpunkten 330 Sekunden und 332 Sekunden insgesamt acht AD-Pakete ausgetauscht. Der Autor vermutet, dass es sich hierbei um die Resultat-Phase handelt. Die Vermutung soll damit gestützt werden, dass dies die letzten Pakete sind, die zwischen Firebase und dem Jenkins-Server ausgetauscht werden.

Die komplette Dauer zwischen den Zeitpunkten 332 Sekunden und 58 Sekunden beträgt somit 274 Sekunden, was auch zu der Ausführung des gcloud-Kommandos in Jenkins von 276 Sekunden passt.

6.2.5 Bitbar

Die Kommunikation zwischen dem Jenkins-Server und Bitbar läuft anders ab als bei den anderen beiden Device Clouds. Dies liegt daran, dass die Testresultate wie bereits in Abschnitt 5.7 erläutert, nicht über die Konsole ausgegeben werden, sondern über eine API abgefragt werden müssen. Somit muss der Jenkins-Server eigenständig diese IP ansprechen, um die Ergebnisse zu erhalten.

Problem bei der Auswertung der Daten[133]

Zunächst wurde ein Log evaluiert, welcher mit tshark aufgezeichnet wurde. Leider stellte sich heraus, dass die Gesamtlänge aller AD-Pakete kleiner war als die Größe der APK und Test-APK, welche an Bitbar geschickt werden müssen (6 MiB gegenüber 8,1 MiB, siehe Log logBit3Sam6). Aus diesem Grund wurde ein neuer Log mit dem Programm tcpdump[134] in der Version 4.9.2, mit den Parametern `-s 65535` und `-w` und den gleichen Rahmenbedingungen aufgezeichnet. In diesem Log ist die Länge aller Application Data größer als die Größe der APK und Test-APK. Der Name dieses Logs ist „bit.log4“. Demzufolge ist dieser Mitschnitt nicht auf einen Messwert aus Tabelle 6.1 bezogen.

Eine erneute ausführliche Ermittlung und Auswertung der Netzwerkdaten unter Firebase und Sauce Labs mit dem Programm tcpdump wurde nicht durchgeführt. Es wurde nach der Evaluation ein tcpdump-Log mit den in diesem Abschnitt genannten Rahmenbedingungen für Sauce Labs aufgezeichnet. Die Gesamtlänge der AD-Pakete wurde dabei untersucht. Diese war mit 8642753 Byte nicht sehr verschieden von den bereits durch tshark aufgezeichnet Log mit 8671944 Byte. Dasselbe gilt für Firebase, bei dem die bereits betrachteten 8671739 Byte nicht zu sehr von den durch tcpdump ermittelten 8643409 Byte abweichen. Diese Zahlen wurden den beiden Logs „sauce.log“ bzw. „fire.log“ entnommen, welche im weiteren Verlauf der Arbeit jedoch nicht weiter betrachtet werden. Aus dem Grund, dass sich die Größe aller AD-Pakete bei diesen beiden Device Clouds bzgl. der jeweiligen Logs von tshark und tcpdump nicht stark unterscheiden, wird vermutet, dass sich die bereits ermittelten Phasen auch in den tcpdump-Logs wiederfinden lassen.

Ermittelung der IPs

Die IP, welche während der mutmaßlichen Upload- und Pull-Phase angesprochen wird, ist 50.18.111.143 . Diese IP wurde im Client Hello Paket 769 ermittelt. Dieses Paket beinhaltet in dessen Metadaten die Zeichenkette cloud.testdroid.com. Wie in Abschnitt 5.3 bereits gesagt wurde, gehört Testdroid zu Bitbar. Somit lässt sich darauf schließen, dass diese IP zu Bitbar gehört. Eine IP, welche später für die vermeintliche Resultat-Phase angesprochen wurde, ist 184.72.52.157 . Im Client Hello Paket 8111 kann hierbei die Zeichenkette cloud.Bitbar.com gefunden werden. Es sei auch hier erwähnt, dass die Kommunikation mit Diffie-Hellman verschlüsselt wurde, was man im Paket 780 herauslesen kann.

Auswertung der Pakete

Es wurden insgesamt 1133 AD-Pakete mit einer Durchschnittslänge von 7570,06 Byte gemessen. Somit stellt sich eine Gesamtpaketlänge von rund 8576878 Byte heraus.

Verlauf der Kommunikation

Die Kommunikation findet zwischen den Zeitpunkten 46 Sekunden und 245 Sekunden statt. Das erste Paket ist das Client Hello Paket 769 und das Letzte ist das Paket 8297.

Die mutmaßliche Upload-Phase ist zwischen den Paketen mit den Nummern 769 und 5427 und somit zwischen den Zeitpunkten 46 und 54 feststellbar. Diese Mutmaßung beruht darauf, dass in dieser Zeit besonders viele AD-Pakete versendet werden. Jedoch werden in der Konsoleausgabe des Bitbar-Plugins keine Zeitstempel in dieser Phase ausgegeben, so dass diese Mutmaßung nicht bewiesen werden kann.

[133] vgl. Jenkins - nwuenscheBitBar #32

[134] https://www.tcpdump.org/tcpdump_man.html, zuletzt besucht am 09.06.2018

Besonders zu beachten sind die drei Encrypted Alert Pakete 5494, 5498 und 5981. Diese treten erst zu den Zeitpunkten 62, 62 bzw. 112 Sekunden auf. Über die Konsolenausgabe können keine entsprechenden Zeitstempel gefunden werden. Somit kann nicht herausgefunden werden, warum diese drei Pakete in solch einen zeitlichen Abstand zu den anderen Paketen der Upload-Phase verschickt wurden.

Wie bereits aus dem Systemlog von Jenkins heraus gelesen werden konnte, beträgt die Zeit zwischen den Pull-Phasen 60 Sekunden. Dies kann auch in dem Quelltext[135] des Bitbar Jenkins Plugins eingesehen[136] werden. Dieser Abstand ist auch in der ausgewerteten Kommunikation wieder zu finden. Wenn man das letzte Application Data Paket 5427 der Upload-Phase und das nächste Client Hello Paket 6003 beachtet, dann liegen zwischen dem Senden der beiden Pakete genau 60 Sekunden Zeitunterschied. Weitere mutmaßliche Pull-Phasen beginnen zu den Zeitpunkten 114, 174 und 234 Sekunden. Aber auch zwischen diesen einzelnen Phasen werden Encrypted Alert Pakete gesendet, die sich keiner konkreten Phase zuordnen lassen. Der Autor mutmaßt, dass es sich hier um Alert Pakete mit der Nachricht „close_notify" handeln könnte. Diese Nachricht könnte anzeigen, dass Bitbar keine weiteren Pakete mehr schicken wird (vgl. (Rescorla & Dierks, 2008)).

Nachdem in den ersten Pull-Phasen jeweils sechs Application Data ausgetauscht werden, werden in der letzten vermeintlichen Pull-Phase bis zum Zeitpunkt 237 Sekunden insgesamt 52 AD-Pakete ausgetauscht, das Letzte ist 8099. Der Autor nimmt an, dass hierbei dem Jenkins-Server übermittelt wurde, dass die Tests alle ausgeführt wurden. Dafür spricht, dass der Jenkins-Server direkt danach eine andere IP von Bitbar anspricht.

Rund 0,3 Sekunden später schickt der Jenkins-Server ein Client Hello im Paket 8111 an die zum Erhalt der Testresultate mutmaßliche IP 184.72.52.157. Für diese Mutmaßung spricht, dass es sich bei den nun ausgetauschten AD-Paketen mit dieser IP um die letzten AD-Pakete handelt, welche zwischen dem Jenkins-Server und Bitbar ausgetauscht werden. Die Verbindung endet mit zwei Encrypted Alert Paketen mit den Nummern 8143 und 8145 an die Testresultat-IP, sowie nachfolgenden drei Encrypted Alert Paketen mit den Nummern 8240, 8294, sowie 8297 zu den Zeitpunkten 241, 241, sowie 245 Sekunden an die in dieser Kommunikation zuerst angesprochene Bitbar-IP. Aus den vorliegenden Messdaten kann nicht geschlossen werden, warum diese letzten Pakete in einem Abstand von vier Sekunden verschickt werden.

Somit läuft die Kommunikation zwischen der Bitbar Device Cloud und dem Jenkins-Server innerhalb von 199 Sekunden ab. Das Bauen der Test-APK und der APK, welche im Bitbar-Projekt dem Ausführen des Bitbar-Plugin vorangestellt werden, beträgt in der Summe sieben Sekunden. Die Zeit zur Ausführung des Post-Skripts wird vernachlässigt. Somit beträgt die Zeit zur Ausführung des Bitbar-Plugins und Post-Skripts laut der Berechnung 200 Sekunden. Dies passt zu der Zeit der betrachteten Netzwerkkommunikation.

6.2.6 Auswertung der Daten

Bei der Betrachtung der Werte aus Tabelle 6.2 fällt auf, dass der Großteil der transferierten Daten aus den APKs besteht. So wurde ermittelt, dass die Größe der APK und Test-APK summiert 8,11 MiB beträgt. Des Weiteren werden bei Bitbar die meisten AD-Pakete verschickt, fast viermal so viel wie bei Firebase, aber dennoch ist die Gesamtlänge der betrachteten Pakete bei Bitbar am kleinsten. [137]

[135] https://github.com/jenkinsci/testdroid-run-in-cloud-plugin, zuletzt besucht am 31.05.2018

[136] vgl. https://github.com/jenkinsci/testdroid-run-in-cloud-plugin/blob/93957a33e6b89f72c98c7a218dabea946e51ab2e/src/main/java/com/testdroid/jenkins/scheduler/APIDrivenTestFinishCheckScheduler.java, zuletzt besucht am 31.05.2018

[137] Es wird vermutet, dass dies damit zusammenhängt, dass bei Bitbar die meiste Zeit zwischen den Pull-Phasen vergeht und somit weniger häufig Pakete ausgetauscht werden.

Device Cloud	OS-Version	Anzahl AD-Pakete	Gesamtlänge AD-Pakete in MiB	Abstand Pull-Phasen in s
Sauce Labs	7.0	912	8,27	30
Firebase	6.0	391	8,27	12
Bitbar	7.0	1129	8,18	60

Tabelle 6.2: Auswertung der Datenströme

Des Weiteren können die in Tabelle 6.1 berechneten Standardabweichungen der Device Clouds von Bitbar und Sauce Labs dadurch erklärt werden, dass zwischen zwei vermeintlichen Pull-Phasen bei Sauce Labs 30 Sekunden und bei Bitbar 60 Sekunden vergehen. Die Standardabweichungen der beiden Device Clouds sind kleiner als die entsprechende Zeit, die zwischen zwei Pull-Phasen vergeht. Somit muss die Ausführung der Tests auf einer Device Cloud nicht tatsächlich die ermittelte Standardabweichung besitzen, sondern man kann davon ausgehen, dass sie daher kommt, dass der Jenkins-Server nicht weiß, wann die Tests genau beendet sind.

Bei Firebase hingegen ist die Standardabweichung höher als die Zeit zwischen den Pull-Phasen. Leider lässt sich auf den Grundlagen der hier erhobenen Daten diese hohe Standardabweichung nicht begründen,da Firebase zudem die kleinste Zeit zwischen den Pull-Phasen vergehen lässt.

Zusammenfassend kann gesagt werden, dass Bitbar im Verhältnis zu den anderen Device Clouds am wenigsten Netzwerkdaten verschickt. Im Bezug auf Anforderung A7 ist somit Bitbar die objektiv beste Device Cloud, welche im Rahmen dieser Arbeit getestet wurde.

6.3 Abdeckung der am meisten genutzten Android-Geräte

Nachdem in Kapitel 5 zunächst die Android-Modelle Samsung Galaxy S7, Samsung Galaxy S8 sowie das Huawei P9 zur Ausführung der Tests auf den Device Clouds definiert wurden, wird nun ausgewertet, in wie weit diese Android-Modelle tatsächlich innerhalb der Prototypen genutzt werden konnten.

Wie in Abschnitt 5.5 beschrieben wurde, ist keines dieser Android-Modell im kostenlosen Testumfang von Sauce Labs enthalten. Es wurde auf alle angeboten Android-Geräte von Sauce Labs ausgewichen.

Bei Firebase zeigte sich im Abschnitt 5.6, dass das S7 genutzt werden kann. Jedoch sind das S8 sowie das P9 nicht vorhanden, weshalb auf das S6 sowie das Huawei P8 Lite ausgewichen wurde.

Für Bitbar konnte im Abschnitt 5.7 gezeigt werden, dass das S8 sowie das P9 genutzt werden, jedoch musste das S7 durch das Samsung Galaxy S7 Edge ausgetauscht werden.

Somit werden von den zunächst definierten Geräten von Sauce Labs keines, von Firebase eins und von Bitbar zwei angeboten. Somit ist Bitbar im Bezug auf die Anforderung A6 die beste Device Cloud.

6.4 Weitere Verwendbarkeit

Zunächst wurde die Anforderung A5 aus Kapitel 3 nur mit dem Bewertungskriterium definiert, dass überprüft werden soll, ob die Device Cloud ein kostenloses Testangebot anbietet. Im Rahmen der durchgeführten Implementierung wurde erkannt, dass es zur genaueren Bewertung der Implementierungen sinnvoll ist, diese Kriterien auszuweiten. Die erweiterten Bewertungskriterien beinhalten nun die folgenden Betrachtungen:

- Parallele Ausführung der Tests auf mehreren Android-Geräten
- Anzahl möglicher Testausführungen pro Tag
- Zeitliche Begrenzung der Testphase

Bitbar bietet an, auf mehreren Android-Geräten gleichzeitig die Tests auszuführen. Dies konnte in Abschnitt 5.7 bestätigt werden. Insgesamt können unbegrenzt viele Tests pro Tag durchgeführt werden. Jedoch ist von Bitbar nicht die Frage beantwortet worden, ob die Device Cloud für das AMCS-Projekt nach Abgabe der Bachelorarbeit weiterhin kostenlos genutzt werden darf.

Sauce Labs bietet, wie in Abschnitt 5.5 gezeigt, keine Parallelität bei der Testausführung an. Weiterhin konnten innerhalb der Testphase statt der besagten 100 Testminuten mehr Minuten genutzt werden, wie in der Implementierung gezeigt wurde. Jedoch ist die Testphase auf 14 Tage begrenzt.

Firebase bietet keine Parallelität der Testausführungen an, wie in 5.6 gezeigt wurde. Des Weiteren können pro Tag insgesamt fünf Testausführungen auf echten Android-Geräten betrachtet werden. Außerdem ist die Testphase zeitlich unbegrenzt.

Sollte das Forschungskonto bei Bitbar weiterhin nutzbar sein, erfüllt Bitbar anhand der neuen Bewertungskriterien die Anforderung A5 am besten.

Sollte Bitbar jedoch sein kostenloses Angebot nicht weiter für das AMCS-Projekt ausweiten, so wird hier nicht mehr auf das kostenlose Testangebot von Bitbar, welches man weiterhin nutzen könnte, eingegangen. Dieses wurde in der Arbeit nicht betrachtet und soll deshalb nicht in diese Evaluation einfließen. Jedoch kann man nicht genau sagen, ob in diesem Fall Sauce Labs oder Firebase nach der Anforderung A5 besser ist.

Sollte man den API-Schlüssel im Falle von Sauce Labs nicht aller 14 Tage aktualisieren wollen bzw. wenn einem die Zahl der angebotenen Android-Modelle zur Ausführung der Tests wichtig ist, muss man die Firebase Device Cloud nutzen. Möchte man jedoch aller 14 Tage den API-Schlüssel der Runner-Datei abändern und das Risiko eingehen, dass dies nicht dauerhaft möglich ist, so bekommt man bei Sauce Labs potenziell mehr Testausführung pro Tag als bei Firebase geboten.

7 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde das Ziel verfolgt, aus den in (Braun et al., 2017) vorgestellten 16 Device Clouds zunächst anhand gesetzter Anforderung, welche für das AMCS-Projekt wichtig sind, herauszuarbeiten, welche Device Clouds prototypisch in das AMCS-System integriert werden.

Es wurden die von Bitbar, Sauce Labs und Firebase angebotenen Device Clouds integriert. Des Weiteren wurde herausgearbeitet, welche Android-Geräte zur Ausführung der Tests genutzt werden sollten, um möglichst viele Fehler innerhalb der AMCS-App zu finden. Die Implementierung der entsprechenden Prototypen wurde anhand von Beispielen erläutert, auftretende Probleme benannt und versucht diese zu lösen. Ein Fehler in der Nutzung eines Context-Objektes innerhalb der AMCS-App konnte bereits mit Hilfe der Device Clouds gefunden und gelöst werden.

Es folgte ein Vergleich der Testausführungszeiten der Integration Tests auf den Device Clouds mit einem Vergleich der aktuell genutzten Lösung. Im Anschluss wurde die Kommunikation zwischen dem Jenkins-Server und den entsprechenden Device Clouds analysiert und verglichen. Weiterhin fand eine Auswertung statt, welche Device Cloud die meisten Android-Geräte aus der zuvor definierten Menge an Android-Geräten zur Testausführung bereitstellt. Zuletzt wurde die weitere Verwendbarkeit der kostenlosen Angebote der Device Clouds analysiert.

Es konnte gezeigt werden, dass mit Hilfe von Device Clouds Tests kontinuierlich auf echten Android-Geräten ausgeführt werden können. Des Weiteren wurden Fehler bei der Ausführung von Tests gefunden, welche nur auf speziellen Android-Geräten auftreten und vom bisher genutzten Emulator noch nicht gefunden wurden. Es kann somit gesagt werden, dass die Nutzung einer Device Cloud im Rahmen des AMCS-Projektes als sinnvoll betrachtet werden kann.

7.1 Integration in das AMCS-Projekt

Der Autor hat sich zum Abschluss dieser Arbeit dafür entschieden, vorerst keine der vorgestellten Device Clouds dauerhaft mit dem AMCS-Projekt zu verbinden. Dass jedoch eine Device Cloud genutzt werden sollte, zeigt sich bereits daran, dass während der Implementierung der Device Clouds viele verschiedene gescheiterte Tests gefunden werden konnten.

Der Grund für eine nicht durchgeführte Integration liegt in dem verfolgten Ziel des Autors. Er wollte die Device Cloud in das AMCS-Projekt integrieren, welche die Anforderung A5 mit den neu definierten Bewertungskriterien am besten erfüllt. Es wurde gezeigt, dass

dies die Device Cloud von Bitbar ist. Jedoch hat sich Bitbar bis zur Abgabe der Bachelorarbeit nicht dazu geäußert, ob das kostenlos angebotene Benutzerkonto mit dem vollen Funktionsumfang auch noch nach dem Abschluss der Arbeit weiterhin innerhalb des AMCS-Projekts genutzt werden darf.

Sollte dies nicht der Fall sein, so würde der Autor die Device Cloud von Firebase weiter nutzen, da die Testphase unbegrenzt lang ist. Dies ist bei Sauce Labs nicht der Fall.

Um die Bitbar Device Cloud in das Projekt einzubinden, müsste bei den in Abschnitt 3.2.2 benannten Branches Master, Develop, Release sowie Hotfix im Jenkinsfile die Stage zum Ausführen der Integration Tests auf dem Jenkins-Emulator durch die Cloud-Stage des Bitbar-Branches ersetzen. Außerdem müsste man die Namen der Branches in der jeweiligen `when`-Klausel der Cloud-Stage einfügen. Weiterhin müssen im Freestyle-Projekt von Bitbar die zu bauenden Branchnamen angepasst werden. Zuletzt müsste das Post-Skript in alle Branches kopiert und der Quelltext zur Lösung des Problems mit dem Context-Objekt aus Abschnitt 5.4.1 eingebunden werden.

Damit Firebase genutzt werden kann, muss bei den oben genannten vier Arten von Branches wieder die Emulator-Stage durch die Cloud-Stage des Firebase-Branches ersetzt und analog zu Bitbar die `when`-Klausel angepasst, sowie das Post-Skript und die Lösung des Context-Problemes kopiert werden.

7.2 Weiterführende Projekte

Abschließend soll ein Ausblick darüber gegeben werden, welche Themen auf dieser Bachelorarbeit aufbauen und ihren Inhalt erweitern könnten.

Im Abschnitt 6.2 wurde bereits geschildert, dass eine Entschlüsselung zwischen dem Jenkins-Server und der Device Cloud mit Hilfe von Wireshark nicht möglich ist. Auf Grund der verschlüsselten Pakete kann beispielsweise bei der Auswertung des Logs von Bitbar nicht beantwortet werden, warum diverse Alert Pakete verschickt werden.

Eine weitere Möglichkeit, die Pakete, die während der Kommunikation ausgetauscht werden, möglicherweise zu entschlüsseln, ist, diese mit Hilfe eines Paketfilters zu untersuchen, bevor sie mit TLS verschlüsselt werden. So könnte unter Linux das Programm **iptables**[138] genutzt werden. Dieses Programm wird zur Konfiguration von **netfilter**[139] genutzt, was die Filterung von Netzwerkpaketen vornimmt. Es kann versucht werden, die Pakete der Kommunikation zu speichern. Dazu könnte man versuchen, Netzwerk-Pakete des Jenkins-Servers zu speichern, bevor diese durch TLS verschlüsselt werden. Analog kann man versuchen, die Pakete der Device Cloud zu speichern, nachdem diese entschlüsselt wurden. Ein Einstieg in dieses Thema bieten Kapitel fünf und sechs der offiziellen netfilter-Dokumentation.[140]

Außerdem kann eine Erweiterung der Pipeline vorgenommen werden. Wie in Kapitel 3 erläutert wurde, handelt es sich bei der aktuell genutzten KS-Lösung für die AMCS Android-App um ein CI-System. Man könnte dieses zu einem CDel-System ausweiten. Ähnliche Fallbeispiele wurden bereits in Abschnitt 2.7 vorgestellt.

Um die AMCS App in einem CDel-System dauerhaft zur Veröffentlichung bereit zu halten, müsste man beispielsweise überprüfen, ob die Versionsnummer der APK seit der letzten Veröffentlichung der App erhöht wurde. Eine weitere, wichtige Aufgabe des CDel-Systems

[138] https://linux.die.net/man/8/iptables, zuletzt besucht am 30.05.2018
[139] https://www.netfilter.org/, zuletzt besucht am 30.05.2018
[140] https://www.netfilter.org/documentation/HOWTO/NAT-HOWTO.html#toc6, zuletzt besucht am 30.05.2018

könnte es sein, automatisiert eine signierte Version der APK zu bauen. Eine solche Signatur ist notwendig, bevor eine APK im Google Play Store veröffentlicht werden darf.[141] Im „User Guide“[141] von Android Studio kann eine Anleitung dafür gefunden werden, wie man mit Hilfe des Programms Gradle eine APK auf der Kommandozeile signiert.

Des Weiteren kann man die in Abschnitt 2.6 vorgestellten Zustände von Steps und Pipelines präziser gestalten. In Abschnitt 5.5 wurde bereits darauf hingewiesen, dass Pipelines, welche erfolgreich gebaut wurden, aber bei der Testausführung fehlschlugen, als gescheitert angezeigt werden. Jedoch sollten sie nach der Definition aus Abschnitt 2.6 als instabile Pipelines dargestellt werden.

Eine Möglichkeit, diese Zustände zu verfeinern, ist die Nutzung des Jenkins-Plugins **Textfinder**[101]. Unter Zuhilfenahme dieses Plugins kann der Zustand einer Pipeline dadurch geändert werden, dass Schlüsselwörter in Textdateien gesucht werden. So könnte die Ausgabe, welche von den Post-Skripts der in Kapitel 5 implementierten Prototypen erzeugt wird, mit Hilfe des Programms `tee` in einer Textdatei abgespeichert werden. In dieser werden, je nach Art des Fehlers, verschiedene Nachrichten abgelegt. Mit Hilfe von Text-finder kann die Datei nach diesen Nachrichten durchsucht werden und so der entsprechende Zustand der Pipeline angepasst werden.

Es sei darauf hingewiesen, dass vom Text-finder Plugin im Januar 2014 das letzte Mal eine neue Version veröffentlicht wurde. Jedoch wurde das Plugin laut einer anonymen Auswertung von Nutzerdaten der Jenkins-Instanzen[142] im April 2018 noch von 7465[143] Jenkins-Servern genutzt. Somit scheint dieses Plugin auch noch mit aktuellen Versionen von Jenkins kompatibel zu sein. Eine beispielhafte Einbindung des Text-finder Plugins kann in (Lamb, 2016) nachgelesen werden.

Zuletzt können die mit Hilfe der Device Cloud gefunden Fehler bei der Testausführung innerhalb der AMCS-App behoben werden. Nachdem mit Hilfe der drei Device Clouds in Kapitel 5 herausgefunden wurde, dass diverse Tests auf speziellen Android-Geräten nicht ausgeführt werden können, sollten diese Fehler auch behoben werden. Hierbei kann mit den entsprechend geworfenen Fehlern bzw. dem Device Log gearbeitet werden.

[141] vgl. https://developer.android.com/studio/publish/app-signing, zuletzt besucht am 30.05.2018
[142] vgl. https://wiki.jenkins.io/display/JENKINS/Plugin+Installation+Statistics, zuletzt besucht am 30.05.2018
[143] vgl. https://plugins.jenkins.io/text-finder, zuletzt besucht am 30.05.2018

Literaturverzeichnis

Baun, C., Kunze, M. & Ludwig, T. (2009). Servervirtualisierung. *Informatik-Spektrum*, *32*(3), 197–205. doi:10.1007/s00287-008-0321-6

Braun, S., Elberzhager, F. & Holl, K. (2017). Automation Support for Mobile App Quality Assurance – A Tool Landscape. In *The 14th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2017)*.

Buchholz, P. (2017). *Entwicklung Automatisierter Tests zur Überwachung der Integration und Performanz von Mobilen Applikationen im Rahmen von AMCS* (Magisterarb., TU Dresden).

Campbell, J. (2018). Culture of Quality: Measuring Code Coverage at Etsy. https://codeascraft.com/2018/02/15/culture-of-quality-measuring-code-coverage-at-etsy/. besucht am: 06.06.2018.

Eckstein, J. (2012). *Agile Softwareentwicklung mit verteilten Teams*. dpunkt.verlag.

Fischer, M. (2018). *A DSL for Configuring Continuous Deployment Pipelines for Android Apps* (Magisterarb., Friedrich-Alexander-Universität Erlangen-Nürnberg).

Fowler, M. (2006). Continuous Integration. https://www.martinfowler.com/articles/continuousIntegration.html. besucht am: 10.05.2018.

Fowler, M. (2013). ContinuousDelivery. https://martinfowler.com/bliki/ContinuousDelivery.html. besucht am: 10.05.2018.

Fowler, M. (2014). UnitTest. https://martinfowler.com/bliki/UnitTest.html. besucht am: 02.05.2018.

Fowler, M. (2018). IntegrationTest. https://martinfowler.com/bliki/IntegrationTest.html. besucht am: 02.05.2018.

Fowler, M. & Highsmith, J. (2001). The Agile Manifesto. http://users.jyu.fi/~mieijala/kandimateriaali/Agile-Manifesto.pdf. besucht am: 07.06.2018.

Friedenberg, S. (2017). How Etsy Ships Apps. https://codeascraft.com/2017/05/15/how-etsy-ships-apps/. besucht am: 10.05.2018.

Gao, J., Tsai, W. T., Paul, R., Bai, X. & Uehara, T. (2014). Mobile Testing-as-a-Service (MTaaS) – Infrastructures, Issues, Solutions and Needs. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering* (S. 158–167). doi:10.1109/HASE.2014.30

Geroch, M. & Zabieglinska–Lupa, M. (2017). In-House or Cloud? Where is More Secure? https://www.infosecurity-magazine.com/blogs/inhouse-or-cloud-where-is-more/. besucht am: 09.05.2018.

Grohmann, W. (2010). Cloud, Iaas, PaaS, SaaS, XaaS, S+S - was ist das? https://www.computerwoche.de/a/cloud-iaas-paas-saas-xaas-s-s-was-ist-das,1933248. besucht am: 09.05.2018.

Haas, M. (2018). Smartphone-Markt: Konjunktur und Trends. https://www.bitkom.org/Presse/Anhaenge-an-PIs/2018/Bitkom-Pressekonferenz-Smartphone-Markt-22-02-2018-Praesentation-final.pdf. besucht am: 24.04.2018.

Hechtel, E. (2016). Mobile Device Emulator and Simulator vs Real Device. https://saucelabs.com/blog/mobile-device-emulator-and-simulator-vs-real-device. besucht am: 02.05.2018.

Hoffmann, A. (2016). Cloud Computing: Public, Private, Hybrid - Was ist was? https://blog.unbelievable-machine.com/cloud-computing-unterschiede-public-private-hybrid. besucht am: 09.05.2018.

ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:Concepts and definitions. (2013). *ISO/IEC/IEEE 29119-1:2013(E)*, 1–64. doi:10.1109/IEEESTD.2013.6588537

ISO/IEC/IEEE International Standard - Systems and software engineering – Requirements for testers and reviewers of information for users. (2017). *ISO/IEC/IEEE 26513:2017(E)*, 1–49. doi:10.1109/IEEESTD.2017.8085435

Iveson, S. (2013). Using Wireshark to Decode SSL/TLS Packets. http://packetpushers.net/using-wireshark-to-decode-ssltls-packets/. besucht am: 01.06.2018.

Kammah, N. (2014). Etsy's Journey to Continuous Integration for Mobile Apps. https://codeascraft.com/2014/02/28/etsys-journey-to-continuous-integration-for-mobile-apps/. besucht am: 09.05.2018.

Karlstetter, F. (2017). Was ist On-Premises? https://www.cloudcomputing-insider.de/was-ist-on-premises-a-623402/. besucht am: 09.05.2018.

Kelly, W. (2016). Mobile app testing: When to use real devices versus emulators. https://techbeacon.com/mobile-app-testing-benefits-when-use-real-devices-vs-emulators. besucht am: 25.04.2018.

Kochhar, P. S., Thung, F., Lo, D. & Lawall, J. (2014). An Empirical Study on the Adequacy of Testing in Open Source Projects. In *2014 21st Asia-Pacific Software Engineering Conference* (Bd. 1, S. 215–222). doi:10.1109/APSEC.2014.42

Kochhar, P. S., Thung, F., Nagappan, N., Zimmermann, T. & Lo, D. (2015). Understanding the Test Automation Culture of App Developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* (S. 1–10). doi:10.1109/ICST.2015.7102609

Kuo, J. Y., Liu, C. H. & Yu, W. T. (2015). The Study of Cloud-Based Testing Platform for Android. In *2015 IEEE International Conference on Mobile Services* (S. 197–201). doi:10.1109/MobServ.2015.36

Lamb, C. (2016). Parsing Jenkins log output to determine job status. https://chris-lamb.co.uk/posts/parsing-jenkins-log-output-determine-job-status. besucht am: 05.06.2018.

Linares-Vásquez, M., Moran, K. & Poshyvanyk, D. (2017). Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (S. 399–410). doi:10.1109/ICSME.2017.27

Linthicum, D. S. (2009). *Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide*. Addison-Wesley Professional.

Martínez, P. A. (2017). Running Android Tests on cloud devices using a Jenkins CI Server. https://pamartinezandres.com/running-android-tests-on-cloud-devices-using-a-jenkins-ci-server-firebase-test-lab-amazon-device-b67cb4b16c40. besucht am: 13.04.2018.

Mell, P. M. & Grance, T. (2011). *SP 800-145. The NIST Definition of Cloud Computing*. Gaithersburg, MD, United States: National Institute of Standards & Technology.

Montegriffo, N. (2018). What is an APK file and how do you install one? https://www.androidpit.com/android-for-beginners-what-is-an-apk-file. besucht am: 06.06.2018.

Mtibaa, A., Fahim, A., Harras, K. A. & Ammar, M. H. (2013). Towards Resource Sharing in Mobile Device Clouds: Power Balancing Across Mobile Devices. *SIGCOMM Comput. Commun. Rev. 43*(4), 51–56. doi:10.1145/2534169.2491276

Nagele, C. (o.D.). An introduction to version control. http://guides.beanstalkapp.com/version-control/intro-to-version-control.html. besucht am: 10.05.2018.

Ostler, U. (2017). Ab 1. September gehört das HPE-Software-Geschäft zu Micro Focus. https://www.datacenter-insider.de/ab-1-september-gehoert-das-hpe-software-geschaeft-zu-micro-focus-a-631072/. besucht am: 01.06.2018.

Pauw, H. (2015). Unterschiede zwischen Continuous Integration, Continuous Delivery und Continuous Deployment. https://www.scrum.de/unterschiede-zwischen-continuous-integration-continuous-delivery-und-continuous-deployment/. besucht am: 10.05.2018.

Rescorla, E. & Dierks, T. (2008). The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. doi:10.17487/RFC5246

Robbins, A. & Beebe, N. H. F. (2006). *Klassische Shell-Programmierung*. O'Reilly Verlag.

Rohrman, J. (2017). Emulator, Simulator, or Real Device - What To Use When? https://saucelabs.com/resources/articles/emulator-simulator-or-real-device-what-to-use-when. besucht am: 02.05.2018.

Rossi, C., Shibley, E., Su, S., Beck, K., Savor, T. & Stumm, M. (2016). Continuous Deployment of Mobile Software at Facebook (Showcase). In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (S. 12–23). FSE 2016. doi:10.1145/2950290.2994157

Shrivatri, A. (2016). *Selection and implementation of test framework for automated system test of mobile application* (Magisterarb., TU Chemnitz, http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-202553).

Silva, D. B., Endo, A. T., Eler, M. M. & Durelli, V. H. S. (2016). An analysis of automated tests for mobile Android applications. In *2016 XLII Latin American Computing Conference (CLEI)* (S. 1–9). doi:10.1109/CLEI.2016.7833334

Solnica, S. (2016). Manually decrypting an HTTPS request. https://lowleveldesign.org/2016/03/09/manually-decrypting-https-request/. besucht am: 01.06.2018.

Vasylyna, N. (2011). The basic definitions of automated software testing. http://blog.qatestlab.com/2011/05/05/the-basic-definitions-of-automated-software-testing/. besucht am: 03.05.2018.

Vilkomir, S. (2018). Multi-device coverage testing of mobile applications. *Software Quality Journal*, *26*(2), 197–215. doi:10.1007/s11219-017-9357-7