

Network Unit Testing

Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2016

Autoren: Andreas Stalder, David Meister
Betreuer: Prof. Beat Stettler, Urs Baumann
Projektpartner: INS Institute for Networked Solutions

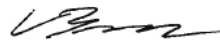
Aufgabenstellung

Nach Netzwerkumbauten wird die Funktionalität heute noch meist von Hand rasch getestet. Die Tests sind meistens einfache Befehle wie ping oder tracer. Anders sieht es in der Software Entwicklung aus. Hier werden zuerst Unit Tests geschrieben und die Entwicklung danach geprüft. Bei neuen Funktionen werden immer auch alle alten Tests durchgeführt. So eine Funktionalität möchten wir nun auch im Netzwerkbereich haben.

Aufgabe: Entwickeln Sie ein System, in dem mit einer einfachen Beschreibungssprache Netzwerk Unit Test geschrieben werden können. Diese sollen dann mit Hilfe von einem Configuration Management Tool wie Saltstack, Ansible oder Puppet auf verschiedenen Sonden ausgeführt werden. Die Tests sollen entweder manuell gesteuert werden oder in einem automatisierten Deployment Prozess integrierbar sein. Spannend ist sicherlich die Erarbeitung und Definition der Beschreibungssprache und die Verknüpfung mit Configuration Management Tools.



Prof. Beat Stettler



Urs Baumann

Abstract

Unit Testing ist in der Softwareentwicklung ein tägliches Werkzeug für automatisiertes White-Box Testing von einzelnen Softwaremodulen. Im Bereich der Computernetze ist ein solches Vorgehen nicht verbreitet. Häufig werden Devices «von Hand» auf Korrektheit und Funktionalität überprüft. Um zukünftig im Netzwerkkumfeld ein automatisiertes Testen zu ermöglichen, besteht die Nachfrage nach einem geeigneten Tool, welches in der Lage ist, die wichtigsten Funktionalitäten im Netzwerk zu überprüfen und auszuwerten.

In einem ersten Schritt wurden relevante, häufig benötigte Testfälle im Netzwerkkumfeld erarbeitet. Diese Testfälle sollen mit der entwickelten Software überprüft werden können. Damit geschriebene Unit Tests auf unterschiedlichen Devices ausgeführt werden können, wurde SaltStack als geeignetes Configuration Management Tool evaluiert. Mittels SaltStack werden Verbindungen auf unterschiedliche Netzwerkdevices über SSH oder RESTful HTTP hergestellt und gerätespezifische Kommandos ausgeführt. Um für Unit Tests notwendige Vergleiche zwischen angenommenen und tatsächlichen Resultaten durchführen zu können, müssen geeignete In- und Outputformate für ein reibungsloses Parsing gefunden werden.

Aus dieser Arbeit ist die Software «nuts» (Network Unit Testing System) auf Basis von Python hervorgegangen. Es konnten viele relevante Testfälle im Netzwerkkumfeld umgesetzt werden. Es hat sich gezeigt, dass die vielen unterschiedlichen Outputformate der Hersteller schwierig zu Parsen sind. Häufig bleibt nur der Weg über manuelles Parsing mittels regulären Ausdrücken. So bleibt zu hoffen, dass die Hersteller zukünftig auf die Ausführung der Kommandos über RESTful HTTP setzen und deren Outputs als strukturierte JSON Objekte zur Verfügung stehen werden.

Management Summary

Ausgangslage

Die Bedeutung des Testens in Netzwerken wächst mit zunehmender Grösse und Komplexität stetig. Oft arbeiten Netzwerk Ingenieure an Bereichen, welche bei Fehlern Auswirkungen auf das gesamte IT-System haben könnten. Im Gegensatz zur Softwareentwicklung existieren im Netzwerkkumfeld keine Tools, um Geräte und deren Funktionalitäten automatisiert zu Testen.

Mit Unit Tests für Netzwerkinfrastrukturen möchte man ein technisches Hilfsmittel für die Qualitätssicherung in der Informatik bereitstellen. Ähnlich wie in der Software Entwicklung soll eine Systematik entwickelt werden, um Komponenten im Netzwerk auf ihre Konfiguration und Funktionalität zu testen.

Vorgehen, Technologien

Anfangs wurde sich mit der Einarbeitung und Analyse von relevanten Testfällen im Netzwerkkumfeld befasst. Auf Basis der gewonnenen Erkenntnisse konnten die Anforderungen an das Softwareprodukt spezifiziert werden. Zusätzlich musste ein Configuration Management Tool analysiert werden, welches in der Lage ist, Befehle auf Netzwerkgeräten abzusetzen. Das Tool "SaltStack" wurde als geeignet angesehen und für die weitere Entwicklung verwendet. Eine geeignete Software Architektur auf Basis von SaltStack wurde ausgearbeitet und mit der Entwicklung der Software "nuts" begonnen.

Ergebnisse

Aus dieser Studienarbeit resultierte eine Python-basierte Software für Unit Tests im Netzwerkkumfeld. Es konnten alle spezifizierten Use Cases abgedeckt werden. Ein automatisiertes Testen von Netzwerken ist mit der entwickelten Software bereits möglich. Wie vorgängig geplant, mussten Abstriche in der Herstellerunterstützung in Kauf genommen werden. Es hat sich gezeigt, dass neue Geräte mit mitgelieferten REST API's wesentlich eleganter getestet werden könnten als ältere Geräte über SSH. Eine Weiterentwicklung der Software mit Unterstützung von neuen Geräten über REST API's wäre denkbar.

Inhaltsverzeichnis

Aufgabenstellung	i
Abstract	ii
Management Summary	iii
I. Technischer Bericht	1
1. Einleitung	1
1.1. Problemstellung	1
1.2. Aufgabenstellung	1
2. Testing in Netzwerken	2
2.1. Übersicht	2
2.2. Device Tests	3
2.2.1. Scope	3
2.2.2. Nutzen	3
2.2.3. Beispiele von Test Cases	3
2.3. Topology Tests	5
2.3.1. Scope	5
2.3.2. Nutzen	7
2.3.3. Beispiele von Test Cases	7
2.4. Traffic Tests	9
2.4.1. Scope	9
2.4.2. Nutzen	10
2.4.3. Beispiele von Test Cases	10
2.5. Network Service Tests	16
2.5.1. Scope	16
2.5.2. Nutzen	16
2.5.3. Beispiele von Test Cases	17
3. Anforderungen	19
3.1. Allgemeine Beschreibung	19
3.1.1. Produktperspektive	19
3.1.2. Produktfunktionen	19

3.1.3.	Benutzer Charakteristik	19
3.1.4.	Einschränkungen	19
3.2.	Use Cases	20
3.2.1.	Use Case Diagramm	20
3.2.2.	Aktoren	20
3.2.3.	Beschreibungen (Brief)	20
3.3.	Nichtfunktionale Anforderungen	22
3.3.1.	Qualität	22
3.3.2.	Schnittstellen	23
3.3.3.	Sicherheit	23
3.4.	Configuration Management Tool	24
3.4.1.	Einleitung	24
3.4.2.	Anforderungen	24
3.4.3.	Ansible	26
3.4.4.	SaltStack	27
3.4.5.	Auswahl	28
4.	Design	29
4.1.	Systemübersicht	29
4.2.	Klassenstruktur	30
4.2.1.	Klassendiagramm	30
4.2.2.	Klassenbeschreibungen	31
4.3.	Logische Architektur	36
4.3.1.	Application Layer	37
4.3.2.	Service Layer	38
4.3.3.	Data Layer	39
4.4.	Deployment Diagramm	40
4.5.	Sequenzdiagramm	41
4.6.	Datensicherung	42
4.6.1.	Testdateien	42
4.6.2.	Auswertungen	42
4.6.3.	Error Log	42

5. Realisierung	43
5.1. Input Files	43
5.1.1. Formate	43
5.1.2. Test	44
5.1.3. Device	48
5.1.4. Validierung	49
5.2. SaltStack	52
5.2.1. Minions	52
5.2.2. Salt-SSH	52
5.2.3. Custom Module	53
5.3. Device Output Formats	56
5.4. Output Parsing	57
5.4.1. Plain Text	57
5.4.2. XML	58
5.4.3. JSON	58
5.5. Ergebnisanzeige	60
5.6. Error Handling und Logging	62
5.6.1. Errorlog	62
5.6.2. Infolog	63
5.7. Unit Testing	64
6. Ergebnisse	66
6.1. nuts	66
6.2. Deviceunterstützung	66
6.3. Kommandos für Unit Tests	67
7. Ausblick	70
8. Glossar	71
II. Anhang	1
1. Projektplanung	1
1.1. Projektübersicht	1
1.2. Zweck und Ziel	1
1.3. Projektorganisation	2

1.4. Management Abläufe	2
1.4.1. Zeitbudget	2
1.4.2. Projektphasen	2
1.4.3. Meilensteine	3
1.4.4. Iterationen	4
1.4.5. Arbeitspakete (Tickets)	5
1.5. Risikomanagement	7
1.5.1. Risiken	7
1.5.2. Umgang mit Risiken	8
2. Zeitauswertung	9
3. Benutzeranleitung	11
3.1. Installation	11
3.1.1. Python	11
3.1.2. Arch Linux	11
3.1.3. Ubuntu	11
3.1.4. Nuts	12
3.2. SaltStack	12
3.3. SaltStack Modul	12
3.3.1. SSH Geräte in Salt erfassen	12
3.4. Cisco Spec-File	13
3.5. Benutzung	13

I. Technischer Bericht

1. Einleitung

1.1. Problemstellung

Die Bedeutung des Testens in Netzwerken wächst mit zunehmender Grösse und Komplexität stetig. Oft arbeitet man als Netzwerk Engineer an Bereichen, welche Auswirkungen auf das gesamte System haben könnten. Manuelles Testen von allen beteiligten Geräten und Systemen kann sehr aufwändig und lückenhaft sein. Ein grosses Netzwerk hat viele Teilnehmer und Komponenten, welche für ein reibungsloses Zusammenspiel notwendig sind. Nach dem Austausch oder nach Änderungen an einem Gerät sollten möglichst umfangreiche Tests durchgeführt werden, damit die Funktionalität im Ganzen sichergestellt werden kann. In der Softwareentwicklung ist ein solches Vorgehen schon lange zum Standard geworden. Unter dem Deckmantel "Continuous Integration" werden für einzelne Module und deren Zusammenspiel Tests geschrieben. Man unterscheidet zwischen Unit Tests (Testing einzelner Module) und Integrations- resp. Systemtests (Testing des Zusammenspiels der Module). Vor jeder Anpassung, Änderung oder Neueinführung werden diese Tests durchgeführt. Somit werden Unverträglichkeiten und schleichende Bugs vermieden. Schlussendlich dient das Testing dem Qualitätsmanagement. Durch gutes und umfangreiches Testing sollen unnötige Down Times vermieden werden. Je nach Branche stellen Down Times im Netzwerk einen erheblichen Business Impact dar und bedeuten erhebliche Kosten.

1.2. Aufgabenstellung

In dieser Arbeit sollen Testfälle für Unit Tests in Netzwerken ausgearbeitet werden. Die Aufgabe besteht darin, ein System, mit welchem Unit Tests für Netzwerke umgesetzt werden können, zu entwickeln. Es soll ein geeignetes Configuration Management Tool evaluiert werden, mit welchem diese Unit Tests auf den Devices ausgeführt werden können.

2. Testing in Netzwerken

2.1. Übersicht

Um ein Netzwerk möglichst vollständig und sinnvoll zu testen, gibt es verschiedenste Aspekte zu beachten. In Computernetzen gibt es eine Vielzahl Devices mit unterschiedlichsten Aufgaben. Diese Devices sind häufig von verschiedenen Herstellern, haben ein unterschiedliches Alter und unterschiedliche Technologien.

Als verantwortlicher Ingenieur eines Computernetzwerks beabsichtigt man mit der Installation und Konfiguration eines Systems ein Verhalten. Mit gezielten Tests möchte man Zustand und Verhalten des Systems testen. Beim Testing gibt es keine Grauzonen, entweder ist der Test erfolgreich, oder er ist es nicht.

Effektives Testing ist mindestens so schwierig wie die eigentliche Entwicklung des Systems. Es erfordert genaueste Kenntnisse des Systems und dessen Zustände und Verhalten.

Seit vielen Jahren gibt es eine grosse Menge an Tools fürs Monitoring in Netzwerken. Testing und Monitoring sind jedoch nicht dasselbe. Beim Monitoring werden mittels Agenten die Zustände der überwachten Geräte (meist mittels SNMP) abgefragt. Die ermittelten Werte werden dann wiederum auf einem Dashboard dargestellt. Die Interpretation dieser Werte überliegt dann wiederum der zuständigen Person. Beim Testing werden vor der Ausführung die erwarteten Resultate definiert. Die Tests werden dann auf dem produktiven System in Echtzeit durchgeführt, die erzielten Resultate werden dann wiederum mit den erwarteten Resultaten verglichen. Dann kann entschieden werden, ob der Test bestanden wurde oder fehlgeschlagen ist.

In diesem Abschnitt wird versucht, möglichst viele Aspekte des Testens in Netzwerken in vier Gruppen unterzubringen. Selbstverständlich sind die Testfälle in diesen Gruppen bei weitem nicht vollständig. Die Vielfalt an Netzwerkprotokollen und Kommunikationsgeräten macht es unmöglich, sämtliche denkbaren Testfälle abzudecken. Der Fokus liegt deshalb auf weit verbreiteten und alltäglichen Tests.

2.2. Device Tests

2.2.1. Scope

Device Tests testen ein einzelnes Netzwerkgerät, wie zum Beispiel Switches, Router oder Firewalls. Häufig geht es bei den Device Tests um Vergleiche des Gerätezustands. Diese Zustände werden normalerweise durch manuelle Show-Befehle abgefragt. Mittels Vergleichen (sogenannten Show Diffs) kann man die gewünschten Zustände testen.

Meist werden die Informationen direkt auf dem Gerät angezeigt und man muss sich auf alle Geräte verbinden. Dies kann durch die Tests umgangen werden.

2.2.2. Nutzen

Der Nutzen durch diese Tests ist enorm. So könnte der Administrator beispielsweise eine Standardkonfiguration einrichten und passende Tests dazu schreiben. Diese Tests decken einzelne Einstellungen der Konfiguration ab und vergleichen diese. Bei jedem neuen Gerät kann der Test ausgeführt und die Resultate angezeigt werden. Oft werden gewisse Benutzer oder kleinere Einstellungen vergessen. Durch Testing werden nun diese Versäumnisse sofort sichtbar. Dies spart Zeit für die Suche von Konfigurationsfehlern und entsprechende Vorgänge werden effizienter. Auch wenn man bei bestehenden Netzen etwas Neues konfiguriert, können so die restlichen Einstellungen getestet werden. Wenn der Test erfolgreich ist, stimmt die Konfiguration und man muss sich nicht mehr direkt auf die jeweiligen Geräte verbinden.

2.2.3. Beispiele von Test Cases

Allgemeine Show Diffs

Mit dem Show Diff Test kann angegeben werden, welchen Befehl man auf dem Gerät ausführen möchte und man erhält die Ausgabe zurück. Diese kann danach mit dem gewünschten Wert verglichen werden und der Test ist bestanden oder nicht. So gibt es eine riesige Anzahl von Tests, welche man ausführen könnte. Hier haben wir ein paar gängige Tests aufgelistet.

Link Speed

Ein falsch ausgehandelter Link Speed kann eine Verbindung stark beeinträchtigen. Deshalb wäre es sehr praktisch, wenn dies automatisch überprüft werden könnte. So hätte man eine Fehlerquelle leicht entdeckt und schnell angepasst.

Users

Lokale Benutzer eines Netzwerk Devices haben meist privilegierte Rechte. Meistens sind eine bestimmte Anzahl und Usernamen vorgesehen. Mittels automatisiertem Testing ist schnell sichtbar, ob etwas an der Userkonfiguration nicht stimmt.

HW-Modul

Manche Devices sind mit zusätzlichen Hardware Modulen ausgestattet. Beispielsweise könnten dies ein zusätzliches Power-Supply oder zusätzliche Netzwerk Ports sein. Mittels Show Befehlen kann man Module und deren Zustand anzeigen lassen

OS-/FW-Version

Die OS- und FW-Version ist für einen Administrator von Bedeutung. Alle Geräte sollten den gleichen Stand haben und alte Geräte müssen einfach zu finden sein. Der Test überprüft alle Geräte und vergleicht den OS-Stand mit der gewünschten Version. Falls ein Gerät veraltet ist, wird dieses angezeigt, damit man ein Update einspielen kann.

Weitere Tests

Man könnte noch viele weitere Tests schreiben, da der Konfigurationsvergleich viel Spielraum gibt. So greifen auch andere Teststufen auf den Show Diff Befehl zu, da er sehr hilfreich ist.

2.3. Topology Tests

2.3.1. Scope

Bei den Topology Tests werden die logischen Strukturen des Netzwerks getestet. In fast jedem Netzwerk existiert eine Vielzahl von Subnetzen. Wir möchten mit Topology Tests die logischen Strukturen auf Layer 2 und Layer 3 testen. Konkret bedeutet dies für Layer 2 vor allem Überprüfungen von Spanning-Tree und für Layer 3 Routing.

Um Loops im Netzwerk zu detektieren und verhindern wird das Spanning-Tree Protokoll verwendet. So stellt ein Algorithmus fest, wenn ein Zyklus in der Layer 2 Topologie gebaut wurde und behebt diesen mittels Blocken von Ports. Ansonsten würden sogenannte Broadcast Stürme für eine Überlast, und somit für einen Ausfall des Netzwerks sorgen. Um Redundanzen und Loadbalancing (für MSTP) auf Layer 2 umzusetzen, kann Spanning-Tree eingesetzt werden.

Damit eine Kommunikation von unterschiedlichen Layer 3 Netzwerken untereinander stattfinden kann, muss geroutet werden. Jeder Router muss sicherstellen, dass alle notwendigen Layer 3 Netze geroutet werden, seien sie direkt an ihm angeschlossen, oder über eine Weiterleitung an einen anderen Router. Falls ein Subnetz nicht geroutet wird, erhält der Client eine "Destination Network unreachable" Fehlermeldung. Die zuständigen Layer 3 Switches, Router und Firewalls tätigen bei einem einkommenden IP-Paket einen Lookup in ihre Routing Tabelle. Die Routing Tabelle kann manuell mittels statischen Routen, aber auch dynamisch mittels Routing Protokollen angepasst werden.

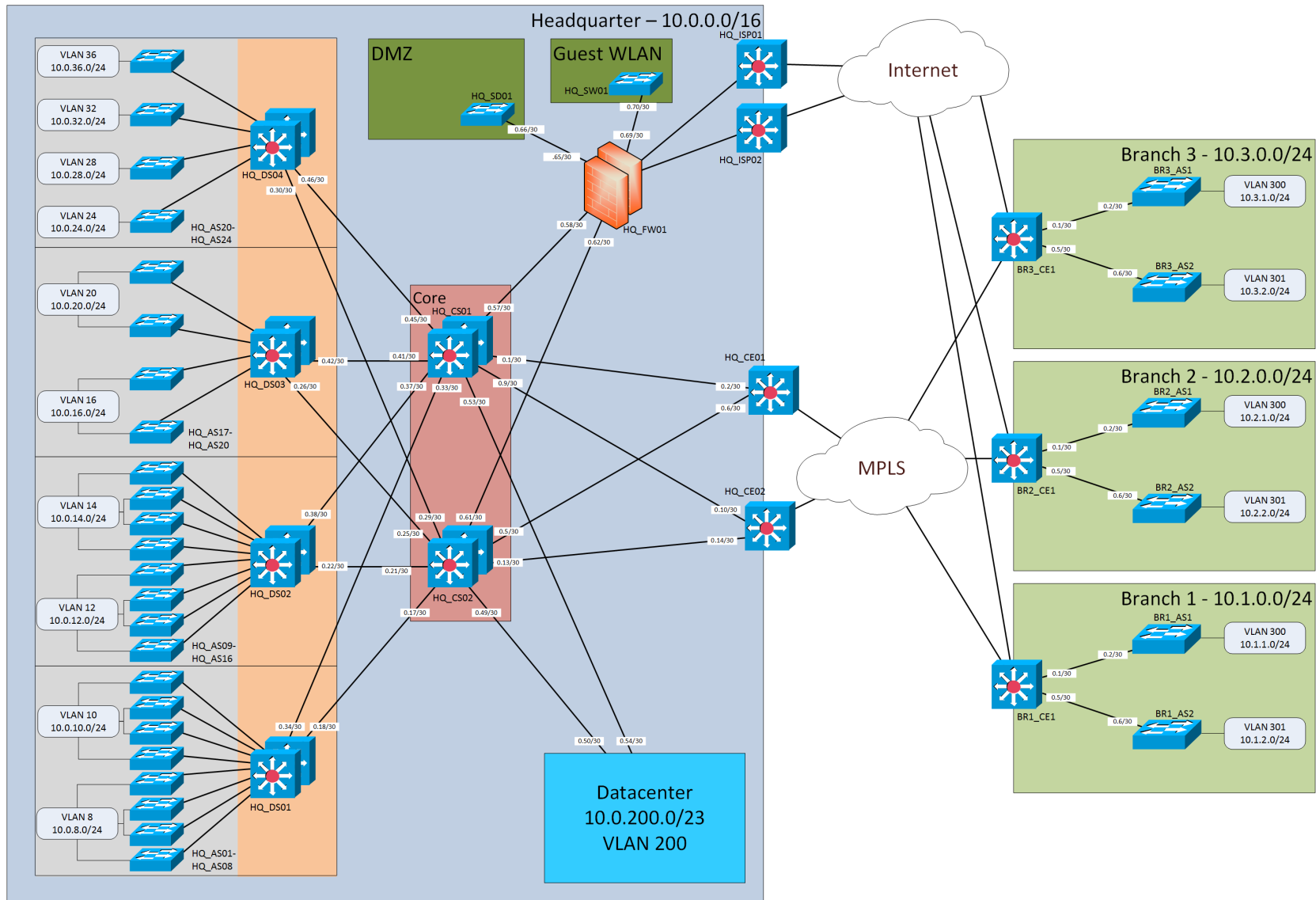


Abbildung 1: Logischer Netzwerkplan

Mit den Topology Tests möchten wir die Erreichbarkeit der Netze aus unterschiedlichen Subnetzen über Layer 3 überprüfen, aber auch das Verhalten der verwendeten Routing-Protokolle wie beispielsweise OSPF oder BGP. Auch Layer 2 Tests mit STP sind vorgesehen, beispielsweise das Ermitteln der Spanning-Tree Root oder wichtige designated- und blocked Ports.

2.3.2. Nutzen

In grossen Netzwerken kann das Routing und Switching schnell komplex und unübersichtlich werden. Viele Routing Einträge auf unterschiedlichsten Devices entstehen dynamisch mittels Routing Protokollen. Man ist kaum in der Lage, sämtliche Routing-Tabellen auf allen Devices zu überblicken. Routing ist ein sehr zentraler und wichtiger Aspekt im Netzwerk. Wenn Fehler im Routing auftreten, kann dies Teile oder das gesamte Netzwerk ausser Betrieb setzen. Mit systematischen Topology Tests kann man innert kürzester Zeit Sicherheit über die Funktionalität des Routing/Switching und die Erreichbarkeit der Netzwerke gewinnen. Ein manuelles durchprobieren der Netz-Erreichbarkeiten ist zwar grundsätzlich möglich, aber mit ansteigender Grösse der Netzwerke ist dieses Vorgehen extrem aufwändig.

2.3.3. Beispiele von Test Cases

Spanning-Tree

Sofern Spanning-Tree im Netzwerk eingesetzt wird, ist es unerlässlich zu überprüfen, ob es korrekt funktioniert. Als Netzwerk Engineer verfolgt man mit dem Einsatz von Spanning-Tree möglicherweise die Absicht, Redundanzen und Ausfallsicherheiten zu bauen oder um unbeabsichtigte Broadcast Storms durch falsches Patching zu vermeiden. Das Testing folgt fast ausschliesslich aufgrund von Show Diffs. So müssen wir auf bestimmten Ports überprüfen, ob er designated-, bzw. blocked ist resp. welche Bridge als Root auserkoren wurde.

Reachability

In diesem Bereich soll die Erreichbarkeit, respektive das Routing, getestet werden indem Zugriffe von Subnetzen in andere Subnetze getätigt werden. Der Verbindungsweg zweier Stationen soll hier erst einmal als Black-Box betrachtet werden. Die Verbindungsqualität ist in diesem Schritt auch nicht von Relevanz, es soll lediglich im Netzwerk getestet werden, ob das Routing zwischen den Netzen funktioniert.

Der einfachste Weg um die Reachability zu testen ist ping. Bei den Netzwerkgeräten ist ein ping-Tool vorhanden. Somit kann ein Router von allen seinen direkt angeschlossenen Netzen einen Ping an unterschiedlichste Destinationen absetzen.

Routing Protokolle

Für den Austausch der Routing-Informationen und den dynamischen Aufbau der Routing Tabellen werden Routing Protokolle eingesetzt. Dabei wird unterschieden, ob innerhalb eines Autonomen Systems (AS) oder zwischen unterschiedlichen AS geroutet wird. Beispielprotokolle fürs Routing innerhalb eines AS wären OSPF, RIP, EIGRP, IS-IS und das heute einzig eingesetzte Protokoll fürs AS Routing ist BGP. Diese Routing-Protokolle verwenden unterschiedliche Algorithmen und besitzen andere Eigenschaften und Informationen über das Netzwerk.

Mögliche Test Cases für Routing Protokolle schauen wir uns am Beispiel OSPF an. So möchte man beispielsweise die Adjacencies benachbarter Router testen, indem überprüft wird, ob, und wie häufig Link State Advertisements ausgetauscht werden. Ebenfalls interessant ist das Verhalten beim Ausfall eines Links, was eine Neuberechnung des SPF Trees bedeutet. Wie lange dauert es, bis das Netzwerk wieder in einem konsistenten Zustand ist (Konvergenz)? Solche Tests könnten mit Hilfe von show Outputs (z.B. Topologie DB's) direkt aus den Devices herausgelesen werden. So kann man vergleichen ob das erwartete Resultat eingetroffen ist.

Routing Paths

Häufig ist es wichtig, welchen Weg Pakete im Netzwerk nehmen. Manchmal ist es notwendig, unterschiedliche Pfade für dasselbe Ziel zu haben. Sei dies aus Gründen der Redundanz für die Ausfallsicherheit oder Load-Sharing für mehr Throughput. Wichtige Verbindungen, beispielsweise an den Core Switches, sind redundant ausgelegt. Fällt ein Link aus, so muss ein anderer Link existieren, um ans Ziel zu gelangen. Bei unterschiedlichen Routen auf dieselbe Destination mit gleichen Kosten (equal cost paths) muss ein Load-Sharing stattfinden. So müssten die Flows gleichmässig auf die zur Verfügung stehenden Routen verteilt werden.

Grundsätzlich muss getestet werden, ob Pakete den vorgesehenen Weg nehmen. Oft ist es wichtig, dass ein bestimmter Verkehr über eine gewisse Verbindung läuft, während ein anderer Verkehr einen unterschiedlichen Weg nehmen sollte. Tools, um den Weg aufzuzeigen, wären beispielsweise traceroute oder pathping.

2.4. Traffic Tests

2.4.1. Scope

Durch die Traffic Tests kann man den End-to-End Traffic, auf dem Netzwerk überprüfen. Das heisst, es werden wichtige Punkte getestet, die das Verhalten der Pakete im Netzwerk beeinflussen. Wichtig sind vor allem die speziellen Flags, welche die Pakete auf dem Netzwerk erhalten. Es ist dabei durchaus erwünscht, dass der VoIP-Verkehr schneller durch das Netz geleitet wird. Weiter ist auch der End-to-End Throughput ein wichtiger Teil des Traffic Tests. Anhand verschiedener Bewertungskriterien werden Qualität und Charakteristika dieser Verbindungen überprüft.

Wenn von Verbindungsqualität gesprochen wird, möchte man vor allem bestimmte verbindungspezifische Parameter wie Round Trip Times, Jitter, Throughput oder Packet Loss testen. Je nach Verbindung, welche es zu testen gilt, kann man sinnvolle Grenzwerte festlegen. So können gewisse Throughput Werte auf der WAN Verbindung einer Firewall akzeptabel sein, während derselbe Wert auf der LAN Verbindung zu niedrig wäre. Es muss jeweils überlegt werden, welche Tests sinnvoll sind, um die komplette Funktionalität der Verbindung auf unterschiedlichen Test Cases abzubilden.

2.4.2. Nutzen

Um unterschiedliche Arten von Netzwerkverkehr (File Transfer, VoIP) zu testen, sind die Traffic Tests nützlich. Man könnte durch diese Tests falsch konfigurierte Quality of Service (QoS) Einstellungen finden und so das Netzwerk enorm optimieren. Durch Traffic Tests hätte man bei jeder Änderung am Netzwerk sofort eine gewisse Sicherheit, dass alles noch richtig funktioniert. Ein weiterer wichtiger Aspekt ist der Throughput im Netzwerk. Mit dem End-to-End Throughput kann man gut abschätzen, ob Services, welche viel Traffic verursachen, richtig funktionieren können. Hier wird nicht nur der einzelne Link gemessen, sondern viele Netzwerkgeräte und Links zusammen. So erhält man eine gute Gesamtübersicht über die erzielten Netzgeschwindigkeiten. Durch systematisches Testen der Verbindungen gewinnen wir Vertrauen in die Funktionalität der Verbindungen. Changes im Netzwerk, seien es einfache Konfigurationsänderungen oder Austausch eines oder mehrerer Devices, können bei auftretenden Fehlern schwerwiegende Folgen mit sich bringen. Als verantwortlicher Netzwerk Engineer trägt man eine grosse Verantwortung. Beim unsystematischen ad-hoc Testing werden leider oft essenzielle Funktionen und Parameter nicht getestet. Es wird sich ganz auf das Knowhow und die Erfahrung des verantwortlichen Engineers gestützt. Unter Zeitdruck werden oft nur sehr wenige Tests durchgeführt, bis das Netzwerk wieder für den produktiven Betrieb freigegeben wird. Mit systematischen Traffic Tests werden beispielsweise fehlende Links und Verbindungsfehler schnell erkannt.

2.4.3. Beispiele von Test Cases

Round Trip Time

Bei der Round Trip Time (RTT) möchte man herausfinden, wieviel Zeit für die Übertragung und Verarbeitung eines Datenpakets über die Verbindung benötigt wird. Es sollen Datenpakete gesendet werden, für welche die Gegenstation extrem wenig Zeit benötigt, um diese zu verarbeiten. Das Interesse besteht also hauptsächlich in der benötigten Zeit für den Übertragungsweg.

Der bekannteste Weg, um die RTT zu ermitteln, ist der ping-Befehl, welcher einen ICMP Request auslöst. Mit Ping kann man die RTT innerhalb der Broadcast Domain, aber auch über dessen Grenzen hinweg ins gesamte Internet herausfinden. Es gibt jedoch noch andere Wege, um die RTT zu bestimmen. Mittels arping wird die RTT über ARP Pakete bestimmt, oder über httping wird die Antwortzeit über das HTTP Protokoll ermittelt. Für Traffic Tests scheint ping über ICMP jedoch als der sinnvollste Weg.

Mögliche Einflussgrößen der RTT sind die Distanz, verwendete Kabel, Routing/Switching, processing, Queueing Delay oder mögliche Übertragungsfehler. Je nach Anwendungsfall können lange Antwortzeiten einen erheblichen Einfluss auf das Befinden der Netzwerkqualität haben. Besonders heikel sind Anwendungen im Voice oder Video Umfeld.

Jitter

Im Kapitel oben wurde die Funktionsweise, die Ermittlungstechniken, die Einflussgrößen und die Auswirkungen der RTT beschrieben. Im Grunde genommen gilt dies alles auch für Jitter. Beim Jitter handelt es sich um nichts anderes, als eine statistische Auswertung verschiedenster Antwortzeiten.

Für die Qualität des Netzwerks ist es zum einen wichtig, gute Antwortzeiten auf den Verbindungen zu erhalten, sprich einen guten Mittelwert davon. Entscheidend ist aber auch die Varianz in den Messwerten, da es unbefriedigend scheint, wenn häufig grosse Abweichungen vom Mittelwert entstehen. Daraus könnte resultieren, dass beispielsweise die Sprachqualität eines Telefongesprächs für eine kurze Zeit als gut empfunden wird und etwas später als sehr schlecht.

Throughput

Unter Throughput oder Durchsatz wird die Übertragungsrage von Netzwerkpaketen in Bits pro Sekunde (bps) verstanden. Der Durchsatz ist mitunter das wichtigste Merkmal für die Leistungsfähigkeit einer Verbindung. Mit automatischen Throughput Tests können leicht Flaschenhälse im Netzwerk aufgedeckt werden. In einem hierarchisch aufgebauten Netzwerk sind wir in den höheren Schichten auf sehr hohe Datenraten angewiesen, da ansonsten das gesamte Netzwerk extrem verlangsamt wird.

Wenn wir uns nun Testszenarien von Datenübertragungsraten für Traffic Tests überlegen, so wäre es sinnvoll, alles zu berücksichtigen, was sich auf den Durchsatz auswirken kann. Die Einflussgrößen wären beispielsweise Link Speed, Übertragungsprotokoll, Path MTU, Packet Loss, Dienstgüte (QoS), etc.

Die meisten Link Speeds werden heutzutage per Auto Negotiation ausgehandelt. Es ist deshalb denkbar, dass bei der Aushandlung etwas schief gegangen ist oder ein Kabel defekt ist. Deshalb können resultierende Link Speeds oder Duplex Modi vom gewünschten Zustand abweichen. Auf sehr performanceintensiven Verbindungen wird häufig auch auf Bündelung von Links (Trunking) zurückgegriffen. Falls einer dieser Links ausfällt, ist es möglich, dass dies gar nie bemerkt wird. Dies ist zwar grundsätzlich die Aufgabe des Monitorings, könnte aber auch mittels eines Tests festgestellt werden. Es ist aber auch möglich, dass der gewählte Load-Balancing Algorithmus nicht funktioniert oder eine ungleiche Lastverteilung verursacht. Mittels eines Tests könnte dies erkannt und gegebenenfalls auf einen anderen Algorithmus zurückgegriffen werden.

Auch Tests zwischen unterschiedlichen Netzwerkstandorten wären denkbar. Es ist durchaus interessant zu testen, wie schnell der Durchsatz vom einen Datacenter zum anderen über die MPLS Cloud des Providers ist. Zusammengefasst möchten wir mit den Throughput Tests hier Verbindungen vor allem End-to-End überprüfen mit dem Ziel, die angestrebten Datenraten auf allen (notwendigen) Links sicherzustellen und mögliche Flaschenhälse im Netzwerk aufzudecken.

Es gibt zwei grundlegende Varianten. Man kann über TCP oder über UDP messen. Dies macht ein grosser Unterschied in der Messung. UDP ist viel schlanker gebaut und hat dementsprechend weniger Overhead. TCP hat mit seinem grösseren Header und dem 3-Way-Handshake einen gewaltigen Nachteil beim Throughput. Dies macht sich bei den Zahlen deutlich bemerkbar.

Daher ist es wichtig die richtige Methode zu wählen. Möchte man zum Beispiel den VoIP Throughput messen, muss man UDP benutzen. Und für normalen Webverkehr nimmt man TCP. Es muss daher immer sorgfältig abgeklärt werden, welche Methode für den speziellen Fall die bessere ist.

Abschliessend gilt für Throughput Tests ganz allgemein anzumerken, dass diese den produktiven Betrieb in erheblichem Masse stören können und keinesfalls leichtfertig ausgeführt werden dürfen. Dies bedarf einer seriösen Zeitplanung und gegebenenfalls Information der Benutzer über mögliche Störungen im Netz.

Packet Loss

Paketverluste können viele Ursachen haben. Manchmal ist ein Paketverlust auch gewollt, beispielsweise bei einer Firewall mittels Drop Policy. In diesem Abschnitt möchten wir uns aber vor allem mit ungewollten Paketverlusten beschäftigen. Diese können beispielsweise bei Fehlern im Übertragungsmedium oder Überlast eines Routers sein. Die Folgen des Paketverlusts können unterschiedlich sein. Bei TCP Protokollen bewirken Paketverluste Retransmissions, deshalb wird der Durchsatz geringer. Bei Realtime Applikationen wie Video oder VoIP, welche UDP basiert sind, ist ein Qualitätsverlust bemerkbar. Eine gewisse Paketverlustrate ist kaum bemerkbar, aber ab einem gewissen Punkt ist beispielsweise ein Gespräch kaum mehr hörbar.

Man muss unbedingt bedenken, dass der Packet Loss je nach Situation variieren kann. Es ist daher beim Testen wichtig, dass man unterschiedliche Protokolle und Paketgrößen verwendet um ein aussagekräftiges Ergebnis zu erzielen. Es ist auch denkbar, dass die Paketverlustrate beim Testbetrieb extrem niedrig ist, und wiederum bei Volllast viel höher, dies gilt es ebenfalls zu berücksichtigen. Packet Loss könnte beim Ermitteln des Jitters zusätzlich errechnet werden. Ein mächtiges Tool wäre beispielsweise iperf.

Porterreichbarkeit

Sobald man in den oberen OSI-Layern angekommen ist, wird ein Port benötigt. Daher ist es wichtig, dass die benötigten Ports des jeweiligen Services auch erreichbar sind.

Ports werden hauptsächlich an zwei Geräten konfiguriert. Auf der Firewall werden Regeln hinzugefügt, welche Ports in das Netzwerk gelangen dürfen. Auf dem Server werden die Ports geöffnet und auf Listen gesetzt. Damit der Service einwandfrei läuft, müssen beide Konfigurationen stimmen. Der Firewall Teil kann man mit Konfigurationsvergleichen sicherstellen. Dies ist im oberen Abschnitt unter Device Tests aufgeführt.

Da die Firewall oft nicht auf Port Scans antwortet, muss man direkt die Erreichbarkeit des Servers testen. Durch Portanfragen auf dem Server (z.B. mit nmap) findet man schnell die offenen Ports. Es wäre auch möglich ganze Portbereiche zu scannen, um den kompletten Server zu testen, anstelle eines einzelnen Services. Voraussetzung dafür ist, dass das jeweilige Device die Port Scans nicht blockiert.

Weitere Einsatzmöglichkeit wären Security Tests. Man könnte für jeden Server einen Test schreiben, mit welchem man dessen offenen Ports abfragt. Wenn der Test erfolgreich durchläuft und keine Fehler anzeigt, ist das Netzwerk in diesem Bereich richtig konfiguriert. Findet man aber offene Ports, welche nicht offen sein sollten, sieht man dies ebenfalls in der Auswertung. Die offenen Ports können nun wieder geschlossen werden und der Server hat keine unnötigen Services aktiviert.

Mit Portscans sollte man aber vorsichtig sein, da diese in gewissen Kreisen als Angriff gewertet werden. Eine Abklärung vor dem Test ist hier von Vorteil, damit keine Missverständnisse entstehen.

QoS

Mit Quality of Service kann Netzverkehr klassifiziert werden. Dienste wie zum Beispiel VoIP benötigen eine stabile und latenzarme Leitung. Ist diese nicht gewährleistet, kann es zu Qualitätseinbußen oder sogar zum Abbruch der Verbindung kommen. Aus diesem Grund wurde Quality of Service (QoS) eingeführt. Durch QoS wird der Verkehr geregelt und priorisiert. Dabei gibt es mehrere Möglichkeiten. Bei der IntServ Strategie wird eine Verbindung zwischen Nodes priorisiert. Bei der DiffServ Strategie werden einzelne Netzwerkprotokolle priorisiert, sogenannte Verkehrsklassen. Wir konzentrieren uns auf DiffServ, da es das verbreitetste ist.

DiffServ bietet 64 verschiedene Klassen an. Nun ist es im Netzwerk wichtig, dass diese Klassen richtig zugewiesen werden. Um dies zu testen, schickt man verschiedene Pakete von einem Client zu einem anderen. Darunter sollten Pakete aus verschiedenen QoS-Klassen vorhanden sein. Beim Empfänger kann man nun das QoS Feld auslesen und auf Richtigkeit prüfen. Wenn die richtige Klasse angezeigt wird, ist der Test bestanden.

Eine weitere Möglichkeit wäre ein Test bei voller Last. So testet man, ob der Verkehr richtig priorisiert wird. In einer produktiven Umgebung sollte man damit aber eher aufpassen, da es zu Störungen kommen kann.

Webservice

Ein grosser Teil der Netzwerke beinhaltet Webservices. Diese sind nicht mehr wegzudenken. Daher ist es wichtig auch diese in einem Netzwerk zu testen. Hier gibt es Fragen wie "Ist der Dienst erreichbar?" oder "Gibt der Dienst die richtige Antwort?" zu klären. Mit den Netzwerktests beantworten wir aber nur den ersten Teil. Der Webserver soll von aussen erreichbar sein und antworten. Die Verantwortung, dass die Antwort richtig ist, liegt beim Softwareentwickler.

Ein Webservice könnte durch mehrere Komponenten beeinträchtigt werden. Wenn ein Webservice beispielsweise auf einer 3-Tier Architektur aufgebaut wurde könnte es bei jeder Schicht zu Fehlern kommen. Daher könnte es schwierig herauszufinden sein, auf welcher Schicht der Fehler entstanden ist. Mit einer falschen Firewall Regeln könnte der Webservice bereits unerreichbar werden. Daher ist es für jeden Administrator sehr wichtig, dass bei Änderungen im Netzwerk ein Feedback angezeigt wird. Falls der Service beeinträchtigt ist, könnte man schnell handeln.

Eine einfache Methode einen Webserver zu testen, ist es eine GET Anfrage zu schicken.

Diese kann wie folgt aussehen:

- GET www.example.com/index.html HTTP/1.1
- GET www.example.com/user/12 HTTP/1.1

Ist der Webservice erreichbar, wird er mit einem Statuscode (z.B. 200) antworten.

Hier in der Grafik sieht man den Vorgang:

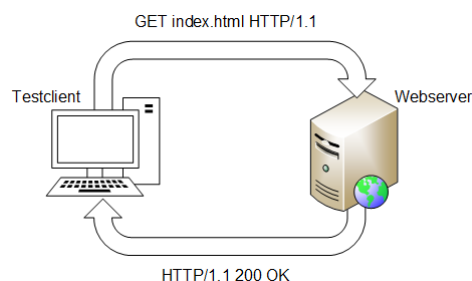


Abbildung 2: HTTP Get Vorgang

2.5. Network Service Tests

2.5.1. Scope

Mit Network Service Tests werden Services der oberen ISO/OSI Schichten getestet. Diese können zum Beispiel DNS, DHCP, Webservices usw. sein. Um sicherzustellen dass alle Services noch einwandfrei funktionieren, sollte man nach jeder Netzwerkkonfiguration alle Services testen.

Wichtige Punkte in diesem Bereich sind die korrekten Einstellungen und die Erreichbarkeit der Services. So ist es wichtig dass der DNS Service im Netzwerk erreichbar ist und korrekte Antworten liefert. Ansonsten könnten viele andere Services beeinträchtigt werden.

Um die Network Service Tests zu bestehen müssen natürlich auch die Tests der unteren Layer einwandfrei funktionieren. Erst dann kann man die Application Tests sauber durchführen.

2.5.2. Nutzen

Mit den Network Service Tests kann man sicherstellen, dass bei Änderungen am Netzwerk alle Services noch erreichbar sind. Dies ist sehr nützlich, wenn man zum Beispiel neue Firewall Regeln erfasst hat. So kann man den Fehler schnell eingrenzen und auffindig machen.

Bei Ausfällen von ganzen Services ist der Schaden enorm. Die Benutzer können gar nicht mehr arbeiten oder sie sind beträchtlich eingeschränkt. Daher ist es sehr wichtig, dass man nach jeder Umstellung oder Konfigurationsanpassung die Tests laufen kann und so eine schnelle Übersicht hat.

2.5.3. Beispiele von Test Cases

DNS

DNS ist einer der wichtigsten Dienste im Netzwerk. Es gibt weltweit tausende Server, welche die Auflösung für Millionen von Website macht. Viele Firmen haben auch einen internen DNS Server, um die lokalen Anfragen zu beantworten. Da viele netzwerkfähige Applikation einen DNS Dienst benötigen, sollte dieser auch unbedingt getestet werden.

Eine Namensauflösung kann man einfach testen. Man stellt eine Anfrage und betrachtet die Antwort. Stimmt diese mit dem gewünschten Resultat überein, ist der Test bestanden. Dazu gibt es Tools wie zum Beispiel nslookup.

Folgende Resource Records kann von einem DNS Server abfragen: SOA, NS, A/AAAA, CNAME und MX Records.

DNSSEC ist eine sehr neue Technik, aber auch hier gibt es nützliche Tests. Zum Beispiel könnte man das Ablaufdatum der Zertifikate sehr gut abfragen. Dazu schreibt man einen automatischen Test, der jeden Tag das Datum überprüft und fehlschlägt, falls das Datum zu nahe am Grenzwert ist. So kann man sich sicher sein, dass man nie ein abgelaufenes Zertifikat hat. Dies ist zwar eher eine Art Monitoring, könnte aber auch als Test funktionieren.

DHCP

Heutzutage ist der DHCP Service nicht mehr wegzudenken. Ohne DHCP Service würden Clients die IP-Adressen fehlen und sie können sich nicht ins Netzwerk verbinden.

Um einen DHCP Service zu testen, schickt man dem DHCP Server Anfragen. Für dies gibt es mehrere Tools, welche das übernehmen. Erhält man eine Antwort, ist es wichtig, dass der Adressbereich übereinstimmt. Man vergleicht die gewünschte Adresse mit der erhaltenen. Durch die Subnetzmaske weiss man genau welcher Bereich dynamisch sein darf.

Neben den IP-Adressen sind auch weitere Felder wichtig wie beispielsweise DNS Server, Default Gateway, Lease Time, usw. All diese Felder können überprüft und verglichen werden.

Time Service

Time Services wie NTP sind in verteilten Systemen unerlässlich. Viele Dienste benötigen eine korrekt synchronisierte Zeit damit sie funktionieren. Oft synchronisieren sämtliche Devices ihre Zeit von einer zentralen Ressource im Netzwerk, beispielweise vom Active Directory Domain Controller. Dieser wiederum holt sich seine Zeit von einem NTP Server im Internet.

Der Time Service sollte auf Verfügbarkeit und Korrektheit überprüft werden. Deshalb muss man testen, ob er Antwort gibt, korrekt synchronisiert ist und ob man von ihm synchronisieren kann.

3. Anforderungen

3.1. Allgemeine Beschreibung

3.1.1. Produktperspektive

Mit Network Unit Tests wird ein automatisches und wiederholbares Testing Framework für Netzwerkkumgebungen bereitgestellt. Das Testing der Funktionalitäten im Netzwerk wird meist manuell ausgeführt. Oft werden wichtige Tests vergessen und dadurch Fehler produziert. Mittels automatischem Testen soll ein Werkzeug bereitgestellt werden, um diese Problematik einzudämmen.

3.1.2. Produktfunktionen

Das Programm soll den Anwendern erlauben, Testszenarien zu konfigurieren, auszuführen und auszuwerten. Diese Testszenarien sollen wiederholbar sein, damit sie nach wichtigen Änderungen im Netz einfach ausgeführt werden können.

3.1.3. Benutzer Charakteristik

Zielpersonen sind Netzwerk Engineers, System Administratoren und weitere Informatiker mit Tätigkeit im Netzwerkbereich. Es werden solide Grundkenntnisse in IP-Netzwerken sowie Kenntnisse der verwendeten Netzwerk Devices vorausgesetzt. Der Anwender muss das zu testende Netzwerk im Detail kennen.

3.1.4. Einschränkungen

Die Software wird auf Linux Plattformen unterstützt. Vorerst werden nur Cisco Netzwerk Devices und Linux Server als zu testende Devices unterstützt.

3.2. Use Cases

3.2.1. Use Case Diagramm

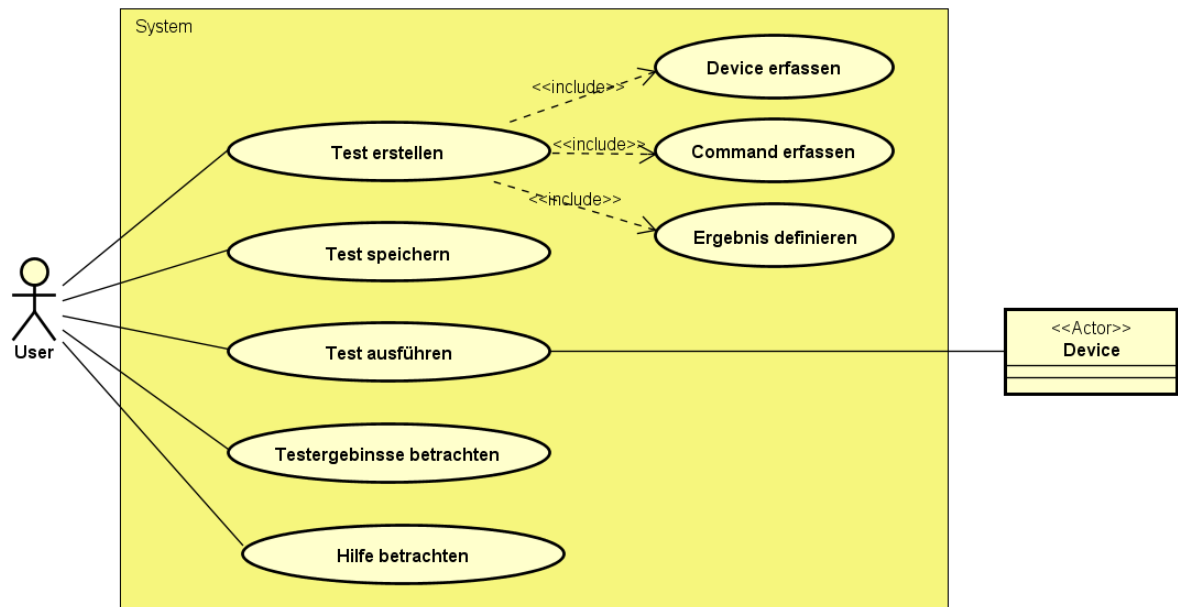


Abbildung 3: Use Case Diagramm

3.2.2. Aktoren

Der Benutzer der Applikation ist in diesem System der einzige primäre Aktor. Zu testende Netzwerk Devices agieren als technische Aktoren in diesem System.

3.2.3. Beschreibungen (Brief)

Test erstellen

Der User möchte einzelne oder mehrere Tests erstellen. Damit dies möglich ist, sind drei weitere Use Cases nötig.

Device erfassen

Für einen Test sind ein- oder mehrere Devices notwendig. Devices haben Eigenschaften wie beispielsweise Name, IP-Adresse, Device-Typ (Router, Switch,...) und Login Daten. Diese Devices müssen mittels einem Setup definiert werden.

Command erfassen

Sobald man ein Device erfasst hat, möchte man Kommandos auf diesem ausführen. Dies könnten beispielsweise show-Befehle oder andere Programme zum Senden von Daten sein. Diese Kommandos sind als Queries auf die Devices zu verstehen.

Ergebnis definieren

Sobald Devices und Commands erfasst wurden, kann nun das erwartete Ergebnis formuliert werden.

Test speichern

Der User hat die Möglichkeit erstellte Tests abzuspeichern um sie später wiederzuverwenden.

Test ausführen

Ein fertig formulierter Test kann ausgeführt werden. Mit diesem Vorgang wird die Verbindung zum Device aufgebaut und das im Test definierte Kommando ausgeführt.

Testergebnisse betrachten

Ein ausgeführter Test ist entweder erfolgreich oder er schlägt fehl. Nach ausgeführten Tests kann der User die Testergebnisse betrachten.

Hilfe betrachten

Wenn der Benutzer Hilfestellungen benötigt, so kann er die Hilfe betrachten. Die Hilfe bietet Unterstützung für die unterschiedlichen Test Commands.

3.3. Nichtfunktionale Anforderungen

In diesem Kapitel behandeln wir die nichtfunktionalen Anforderungen an das Projekt. Wir behandeln Aspekte und Anforderungen aus den Bereichen Qualität, Schnittstellen und Randbedingungen.

3.3.1. Qualität

Bei der Softwarequalität stützen wir uns auf die ISO/IEC 9126 Normen. Es werden die Merkmale Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Wartbarkeit und Übertragbarkeit aufgeführt.

Funktionalität

Netzwerkdevices können von vielen unterschiedlichen Herstellern kommen. Diese Hersteller verwenden unterschiedliche Syntax und Ausgabeformate. Um die Funktionalität bestmöglich sicherzustellen, wird die Herstellerunterstützung vorerst stark eingeschränkt. Vorgesehen sind vorerst Cisco Netzwerkdevices und Linux Hosts.

Zuverlässigkeit

Tests sind wichtig und nützlich, jedoch nicht businesskritisch bei einem möglichen Ausfall. Es muss vor allem darauf geachtet werden, dass bei ssh Verbindungen ein sauberes Exception Handling implementiert wird, falls beim Verbindungsaufbau oder bei abgesetzten Kommandos etwas schief geht. Es müssen für gewisse Tests auch Timeouts eingeplant werden, damit das Programm nicht unendlich lange blockieren kann.

Benutzbarkeit

Wir möchten ein schmales User Interface auf Konsolen Ebene bieten. Der Anwender muss über Kenntnisse auf der Linux Shell verfügen. Mittels eingebauter Hilfe soll es versierten Benutzern möglich sein, die Software zu verwenden.

Effizienz

Die Software wird lokal betrieben. Die einzelnen Unit Tests laufen seriell ab. Die gesamt benötigte Ausführungszeit ist also die Summe aller Ausführungszeiten einzelner Tests.

Wartbarkeit

Eigene Test Cases sollen mit den notwendigen Kenntnissen selbst ergänzt werden können. Command-Mapping und Outputs müssen bekannt sein, dann ist eine Erweiterung des Funktionsumfangs der Test Cases denkbar.

Übertragbarkeit

Die Übertragbarkeit auf andere Plattformen oder Hersteller ist schwierig und vorerst nicht vorgesehen.

3.3.2. Schnittstellen

Benutzerschnittstellen

Die Steuerung des Programms ist mittels Tastatur über die Linux Shell vorgesehen. Ein graphisches User Interface ist in diesem Projekt nicht vorgesehen.

Netzwerkschnittstellen

Im Programm können alle Devices verwendet werden, welche über ein lokales Netzwerkinterface über TCP/IP erreichbar sind.

3.3.3. Sicherheit

Um auf Devices verbinden zu können, muss man sich auf diesen authentifizieren. Administrator Zugänge müssen deshalb best möglichst geschützt sein. Die Übertragung muss verschlüsselt sein (SSH) und die Passwörter dürfen, wenn überhaupt, nur mittels sicherem Hashverfahren abgelegt werden.

3.4. Configuration Management Tool

3.4.1. Einleitung

Die zu entwickelnde Software nimmt vom User erstellte Unit Tests entgegen. Diese Unit Tests werden in einer strukturierten Textform geschrieben. Nun wird ein Tool benötigt, welches die gewünschten Tests ausführen kann. Dazu müssen Devices und Credentials verwaltet-, Tests entwickelt-, Tasks ausgeführt- und Outputs formatiert werden können. Als agentenlose Tools kamen Ansible und SaltStack in Frage. Beide Tools scheinen die gewünschten Anforderungen abzudecken.

3.4.2. Anforderungen

Agentless

Devices müssen ohne Zusatzinstallationen seitens Configuration Management Tool benutzbar sein.

SSH Verbindungen auf Remote Devices möglich

Das Configuration Management Tool muss sich auf Remote Devices über das SSH Protokoll verbinden können.

Kommandos über SSH absetzbar

Devicespezifische müssen abgesetzt werden können nachdem sich das Configuration Management Tool über SSH mit dem Device verbunden hat.

Weiterverarbeitung der Device Outputs

Nachdem ein Kommando abgesetzt wurde, wird ein Output erwartet. Das Parsing dieser Outputs ist ein sehr wesentlicher Bestandteil dieser Arbeit. Die Outputs können entweder strukturiert oder unstrukturiert ausfallen.

Verwaltung von Credentials

Devices sind meist mittels Username und Passwort geschützt. Um den Testprozess bestmöglich zu automatisieren, ist eine Ausführung ohne wiederholte Eingabe der Credentials wünschenswert.

Timeouts

Falls ein Task zu lange dauert, sollte er nach einem gewissen Schwellwert (Timeout) fehlschlagen und abgebrochen werden.

Entwicklung eigener Testmodule

Damit die benötigten Befehle auf den passenden Devices ausgeführt werden können, müssen eigene Module im Configuration Management Tool programmiert werden können.

3.4.3. Ansible

Ansible ist eine Open Source Software, mit welcher man Devices konfigurieren und administrieren kann. Die zu verwaltenden Devices werden über SSH angesprochen und erfordern keinerlei zusätzliche Software. Ansible kann Software verteilen, Konfigurationen ändern, aber auch Abfragen auf Devices ausführen. Sogenannte Playbooks werden im YAML Format erfasst. Playbooks können Verbindungen, Devices und Kommandos enthalten. Dadurch könnte es potenziell für Unit Tests verwendet werden.

SSH Verbindungsaufbau

SSH Verbindungen können in Ansible auf verschiedene Wege aufgebaut werden. Sogenannte Module stehen standardmässig zur Verfügung. Es existieren auch spezielle Module für unterschiedliche Hersteller wie Cisco oder Juniper. Dadurch können Verbindungen einfach aufgebaut werden.

Absetzen von Commands

Durch die Standardmodule ist das Absetzen von Kommandos sehr einfach. Es muss nur das jeweils richtige Modul für das Device verwendet werden, dann können Kommandos abgesetzt werden, als wäre man direkt mit dem Device verbunden.

Output Formatierung

Zurückgegebene Outputs der Devices werden im JSON Format dargestellt. Die Outputs werden Zeile-für-Zeile in ein JSON Objekt geschrieben. Für die Weiterverarbeitung müssten die benötigten Werte aus dem JSON Objekt geparkt werden.

Timeouts

Für Tasks in Ansible könnten Timeouts und Retries festgesetzt werden.

Eigene Module

Für die Entwicklung eigener Module für Unit Tests bietet Ansible eine API. Andere Module können dabei nicht angesprochen werden. Grundsätzlich gestaltet sich die Entwicklung eigener Module schwierig und ist wenig intuitiv und die Dokumentation im Internet ist dürftig. Beispielsweise ist ein Verbindungsaufbau über SSH in einem eigenen Modul schwierig zu implementieren und würde eine grosse Einarbeitungszeit bedeuten.

Device- und Credentialverwaltung

Devices, Verbindungsinformationen und Credentials können in Ansible in sogenannten Inventory Files abgelegt werden.

3.4.4. SaltStack

Ähnlich wie Ansible ist auch SaltStack eine Open Source Software für Konfigurationsmanagement und Administration von Devices. Anders als bei Ansible kann mit SaltStack ein Client-Server Ansatz verfolgt werden. Dafür gibt es in SaltStack sogenannte Master und Minions. Wie in Ansible ist aber auch eine agentenlose Verbindung über SSH möglich. Der Funktionsumfang und die abgedeckten Use Cases sind ähnlich wie bei Ansible, deshalb könnte es potenziell für Unit Tests verwendet werden.

SSH Verbindungsaufbau

Mittels dem Unterprogramm "salt-ssh" kann einfach eine Verbindung zu einem Remote Device aufgebaut werden. Anders als bei Ansible existieren keine vorgefertigten Module um Unterschiedliche Typen von Devices anzusprechen.

Absetzen von Commands

Das Absetzen von Kommandos über SSH geschieht rudimentär, das heisst es müssen sämtliche benötigten SSH Kommandos selbst abgesetzt werden. Dadurch müsste man z.B. auf einem Cisco Device vorgängig "enable" mitgeschickt werden, bevor Abfragen gemacht werden könnten.

Output Formatierung

Standardmässig wird der Output unformatiert zurückgegeben. Man müsste den Output von Hand in strukturierte Form (JSON/XML) bringen oder die benötigten Infos mittels regulären Ausdrücken selbst parsen.

Timeouts

Für Tasks in SaltStack können Timeouts festgelegt werden.

Eigene Module

Für die Entwicklung eigener Module für Unit Tests bietet SaltStack eine API. Verglichen mit Ansible können eigene Module viel einfacher und intuitiver entwickelt werden. SSH Verbindungen oder HTTP GET Anfragen können sehr einfach formuliert werden.

Device- und Credentialverwaltung

Devices, Verbindungsinformationen und Credentials können in Ansible in sogenannten Roster Files abgelegt werden.

3.4.5. Auswahl

Die Auswahl des zu verwendenden Tools hat für die Entwicklung eine grosse Tragweite. Die Software interagiert direkt mit dem verwendeten Tool. Die unterstützten Tests werden selbst im Tool in einem eigenen Modul formuliert.

Beide Tools decken grundsätzlich die gewünschten Anforderungen ab. Ansible punktet mit vorgefertigten Modulen und JSON Output Formatierung, SaltStack punktet mit der einfachen Entwicklung von Modulen. Da auch mit Ansible die erhaltenen JSON Objekte geparst werden müssen fällt dieser Vorteil nicht ins Gewicht. Bei eigenen Modulen können andere Module wie Beispielsweise Cisco IOS nicht verwendet werden. Deshalb haben wir uns für SaltStack entschieden, da ein neues Modul für Unit Tests einfach, schnell und intuitiv entwickelt werden kann.

4. Design

4.1. Systemübersicht

In der nachfolgenden Abbildung ist eine Darstellung des Systems ersichtlich. Der User muss Unit Tests und Devices mittels YAML File erfassen. Diese Files sind beim Start von nuts (Network Unit Testing System) mittels Parameter mitzugeben. Nuts validiert diese Files, erstellt benötigte Objekte und führt die Kommandos auf der zugrundeliegenden SaltStack Installation durch. SaltStack wird mit einem eigenen Testing Modul ausgestattet, sodass die richtigen Kommandos auf die dafür vorgesehenen Devices absetzt. SSH oder HTTP Verbindungen werden von SaltStack aufgebaut. Die eigene Modulentwicklung stellt sicher, dass die Outputs in benötigter Form (JSON) der Applikation zurückgegeben werden.

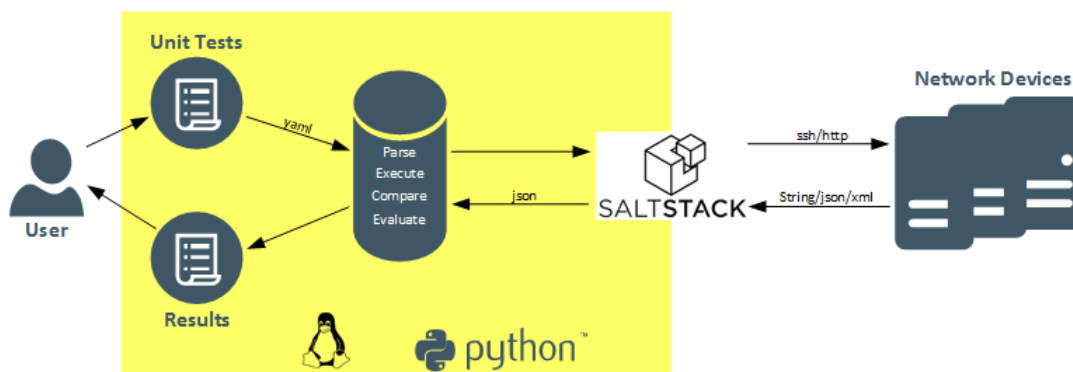
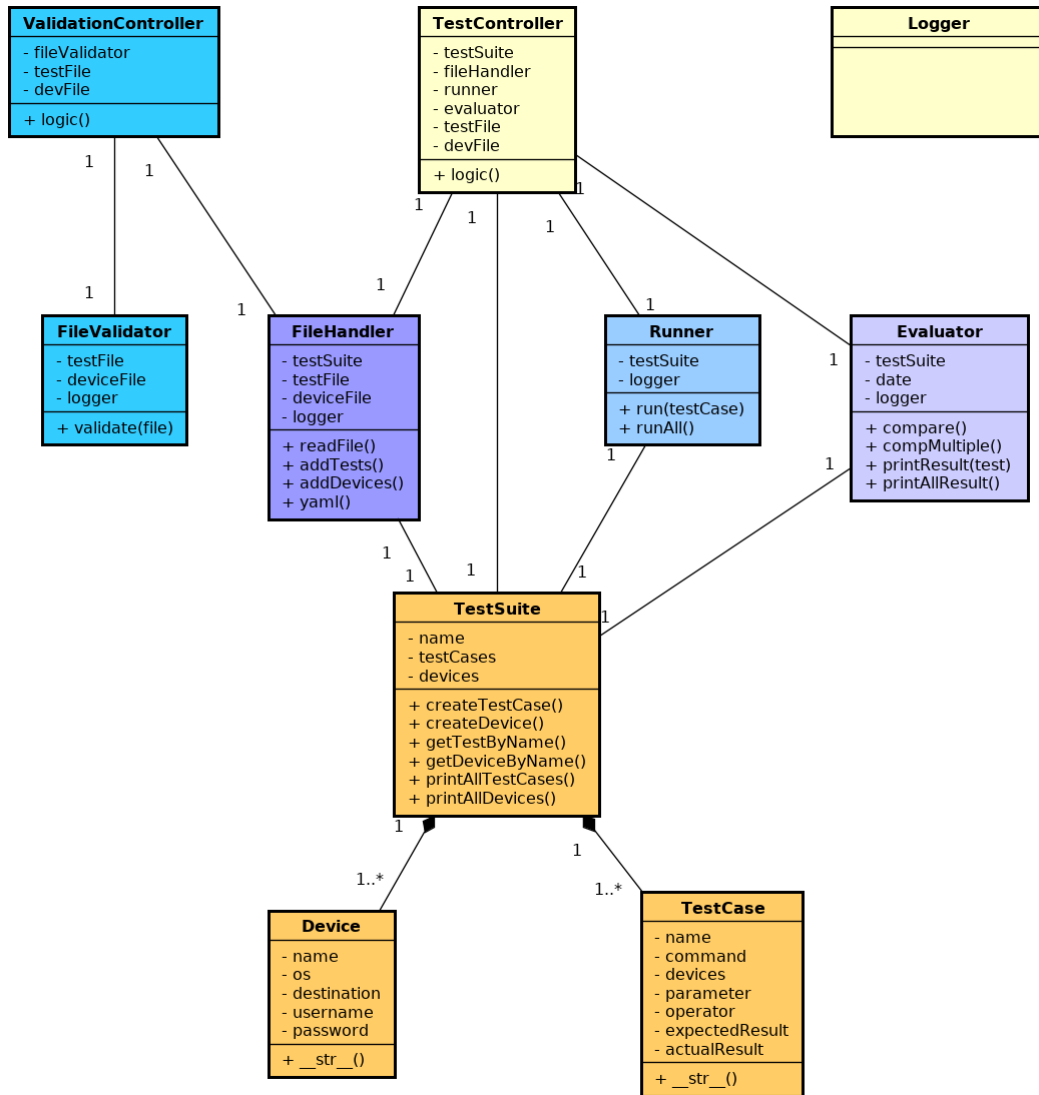


Abbildung 4: Systemübersicht

Die zurückgegebenen Werte von SaltStack können nun mit den vom User getroffenen Annahmen verglichen und ausgewertet werden. Es kann nun festgestellt werden ob der Test bestanden wurde oder fehlgeschlagen ist.

4.2. Klassenstruktur

4.2.1. Klassendiagramm



powered by Astah

Abbildung 5: Klassendiagramm

4.2.2. Klassenbeschreibungen

TestController

Der TestController wird von der Main Methode gestartet, sofern die vorgesehenen Parameter mitgegeben wurden. Der TestController erzeugt im Konstruktor die benötigten Objekte und speichert sie als Referenz ab. Diese Klasse übernimmt die Steuerung und den Ablauf für die Erstellung, Durchführung und Überprüfung der Tests.

Eigenschaft	Beschreibung
testSuite	Referenz auf das TestSuite Objekt
fileHandler	Referenz auf das FileHandler Objekt
runner	Referenz auf das Runner Objekt
evaluator	Referenz auf das Evaluator Objekt
logic()	Methode, welche sämtlichen Operationen in der richtigen Reihenfolge auf die referenzierten Objekte durchführt

Tabelle 1: **TestController**

ValidationController

Der ValidationController wird von der Main Methode gestartet, sofern die vorgesehenen Parameter mitgegeben wurden. Der TestController erzeugt im Konstruktor die benötigten Objekte und speichert sie als Referenz ab. Diese Klasse übernimmt die Steuerung und den Ablauf für die Erstellung, Durchführung und Validierung des TestFiles.

Eigenschaft	Beschreibung
fileValidator	Referenz auf das FileValidator Objekt
testFile	YAML Testfile als Dictionary
logic()	Methode, welche sämtlichen Operationen in der richtigen Reihenfolge auf die referenzierten Objekte durchführt

Tabelle 2: **ValidatonController**

Logger

Die Loggerklasse beinhaltet alle Einstellungen für die zwei Logger, welche im Tool verwendet werden. Diese Klasse wird beim Starten einmal aufgerufen, damit die Logger definiert sind. Danach wird diese Klasse nicht mehr direkt benötigt.

Eigenschaft	Beschreibung
-	-

Tabelle 3: **Logger**

FileValidator

Der FileValidator validiert das mitgegebene YAML Testfile auf Gültigkeit.

Eigenschaft	Beschreibung
testFile	YAML Testfile als Dictionary
deviceFile	YAML Devicefile als Dictionary
logger	Referenz zur Loggerinstanz
validate()	Validiert das mitgelieferte YAML File und gibt die Resultate aus.

Tabelle 4: **FileValidator**

FileHandler

Der FileHandler wird vom TestController erzeugt und erhält das TestSuite Objekt von diesem. Diese Klasse liest die YAML Files ein, speichert sie in ein Dictionary.

Eigenschaft	Beschreibung
testSuite	Referenz auf das TestSuite Objekt
testFile	Dictionary Repräsentation des YAML Test Files
deviceFile	Dictionary Repräsentation des YAML Device Files
readFile()	Liest vom User mitgegebenes File ein und speichert es in die Filevariable
addTests()	Iteriert durch das testFile Dictionary und erstellt Tests auf dem testSuite Objekt
addDevices()	Iteriert durch das deviceFile Dictionary und erstellt Devices auf dem testSuite Objekt
yaml()	Methode, welche dem readFile aus dem YAML ein Dictionary macht.

Tabelle 5: FileHandler

Runner

Die Runner Klasse führt die TestCases der TestSuite auf SaltStack aus.

Eigenschaft	Beschreibung
testSuite	Referenz auf das TestSuite Objekt
logger	Referenz zur Loggerinstanz
run()	Führt die Tests der testSuite über SaltStack aus
runAll	Holt alle Tests aus der Testsuite und iteriert darüber und führt dabei run() aus.

Tabelle 6: Runner

Evaluator

Die Evaluator Klasse vergleicht Annahmen und erzielte Resultate der Tests und wertet/gibt diese aus.

Eigenschaft	Beschreibung
testSuite	Referenz auf das TestSuite Objekt
date	Variable für das aktuelle Datum und Zeit, um die Log-Datei eindeutig zu benennen
logger	Referenz zur Loggerinstanz
compare()	Überprüft Annahme und erzielt Resultat von Tests
compMultiple()	Hilfsmethode, welche zwei Listen miteinander vergleicht
printResult()	Gibt das Ergebnis des Tests aus
printAllResult()	Gibt die Ergebnisse aller Tests aus

Tabelle 7: **Evaluator**

TestSuite

Die TestSuite Klasse beinhaltet sämtliche TestCases und die verwendeten Devices.

Eigenschaft	Beschreibung
name	Name für die TestSuite
testCases	Liste von TestCase Objekten
devices	Liste von Device Objekten
createTestCase()	Erzeugt ein TestCase Objekt und fügt es der TestSuite hinzu
createDevice()	Erzeugt ein Device Objekt und fügt es der TestSuite hinzu
getTestByName()	Holt anhand des Testnames den richtigen Testcase aus der Liste
getDeviceByName()	Holt anhand des Devicenames das richtige Device aus der Liste
printAllTestCases()	Gibt alle TestCases der TestSuite aus
printAllDevices()	Gibt alle Devices der TestSuite aus

Tabelle 8: **TestSuite**

Device

Die Device Klasse beinhaltet sämtliche Informationen eines Devices.

Eigenschaft	Beschreibung
name	Name für das Device
os	Angabe des zugrundeliegenden Betriebssystems
destination	IP-Adresse oder Hostname des Devices
username	Username fürs Login
password	Passwort fürs Login
__str__()	Stringmethode, welche für die Printmethode benötigt wird.

Tabelle 9: **Device**

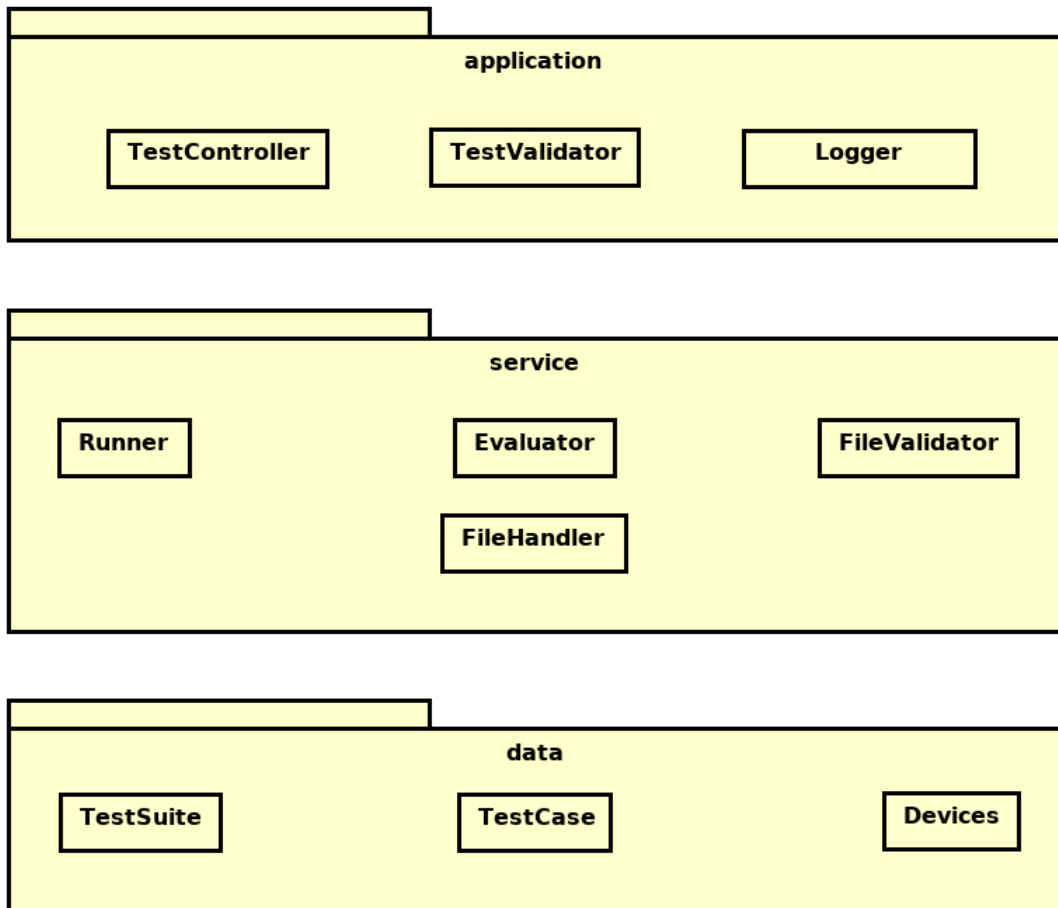
TestCase

Die Device Klasse beinhaltet sämtliche Informationen eines Devices.

Eigenschaft	Beschreibung
name	Name für das Device
command	Das zu verwendende Kommando im SaltStack Modul
devices	Verwendetes Device
parameter	Zusatzparameter
operator	Vergleichsoperator für die Tests.
expectedResult	Resultatannahme des Users
actualResult	Erzieltes Resultat des TestCases
__str__()	Stringmethode, welche für die Printmethode benötigt wird.

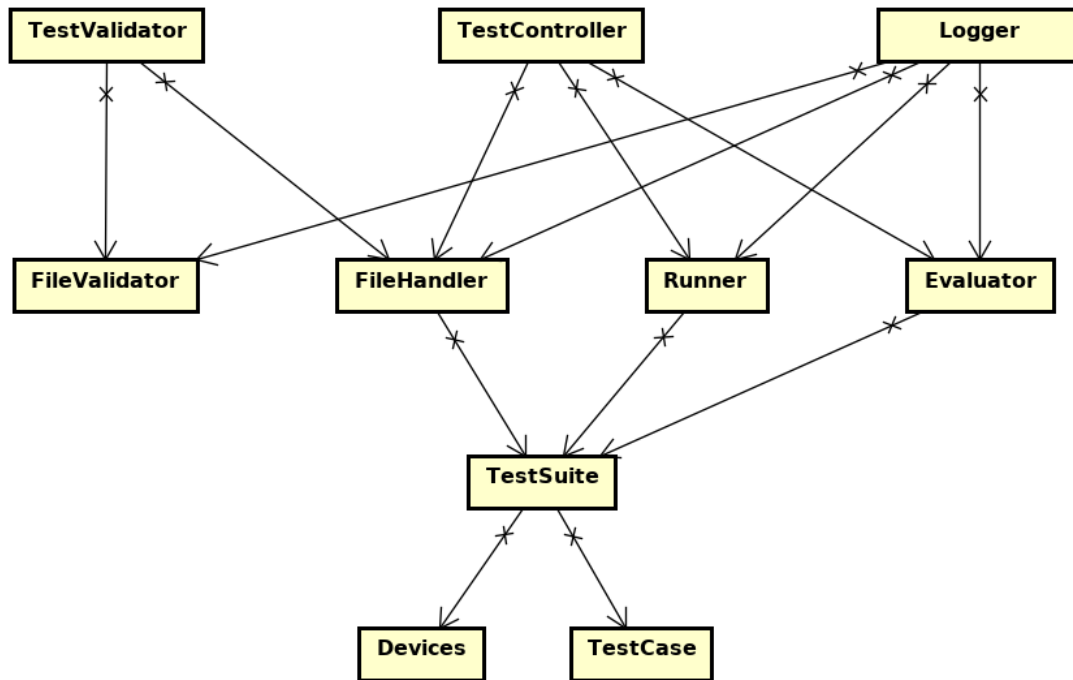
Tabelle 10: **TestCase**

4.3. Logische Architektur



powered by Astah

Abbildung 6: Software Architektur



powered by Astah

Abbildung 7: Dependence Diagramm

4.3.1. Application Layer

Im Application-Layer befinden sich alle Abläufe, um durch das Programm zu führen. Das heisst, dass jeder Parameter seine eigene Controller-Klasse erhalten wird. Da momentan nur zwei Parameter (-i und -v) vorhanden sind, gibt es auch nur zwei Controller.

Klassenstruktur

Klassenname	Beschreibung
ValidationController	Der ValidationController beinhaltet die Logik und Ausgaben, um durch die Validierung einer Datei zu führen.
TestController	Der TestController beinhaltet die gesamte Logik um Tests zu erstellen, auszuführen und zu testen. Dazu verwendet er Teile aus dem Service-Layer.
Logger	Der Logger beinhaltet die Definition der zwei verwendeten Logger des Tools

Tabelle 11: Application Layer

Schnittstellen

Der Application-Layer hat momentan nur eine Schnittstelle in den Service-Layer. So kann er die verschiedenen Aufgaben an die untere schickt weiterleiten.

4.3.2. Service Layer

Der Service-Layer beinhaltet die gesamte Programmlogik. Das heisst es gibt vier verschiedene Klassen - Runner, FileHandler, Evaluator, FileValidator – welche jeweils ein Teil der gesamten Aufgabe übernehmen.

Klassenstruktur

Klassenname	Beschreibung
Runner	Alle erfassten Tests werden durch den Runner von der TestSuite geholt und es wird ein CMD für Salt zusammengestellt. Dieses wird dann ausgeführt und die Testresultate werden der TestSuite zurückgegeben.
Evaluator	Der Evaluator vergleicht die erwarteten Werte aus der TestSuite mit den tatsächlichen Resultaten.
FileHandler	Mit dem FileHandler werden die Test und die Inventory Files eingelesen und in Objekte umgewandelt.
FileValidator	Der FileValidator überprüft ein Test und gibt zurück, ob dieses korrekt formatiert ist und ob die eingegebenen Parameter gültig sind.

Tabelle 12: **Service Layer**

Schnittstellen

Der Service Layer hat eine Schnittstelle zum Data-Layer. Durch den Service Layer werden die Datenobjekte erstellt, bearbeitet und ausgewertet. Obere schichten müssen so immer über den Service Layer, um eine saubere Abtrennung der Schichten zu gewährleisten.

4.3.3. Data Layer

Der Data-Layer beinhaltet alle Datenobjekte, welche vom laufenden Programm benötigt werden.

Klassenstruktur

Klassenname	Beschreibung
TestSuite	Die Testsuite beinhaltet die Logik, um neue Testcases und Devices anzulegen und abfragen auf die Daten zu machen.
TestCase	In der Klasse TestCase, werden alle Daten zu einem TestCase gespeichert. Diese kommen vom Test File, welches der User vorgängig erfasst hat.
Device	Alle Daten zu den einzelnen Devices aus dem Inventory File, werden hier als Objekt abgespeichert.

Tabelle 13: Data Layer

Schnittstellen

Da der Data Layer der unterste Layer ist, gibt es hier keine weiteren Schnittstellen.

4.4. Deployment Diagramm

Lokale Installation

Hier sieht man das Deployment Diagramm mit all den wichtigen Komponenten. Man sieht wo sich nuts befindet und wo das Modul implementiert wird.

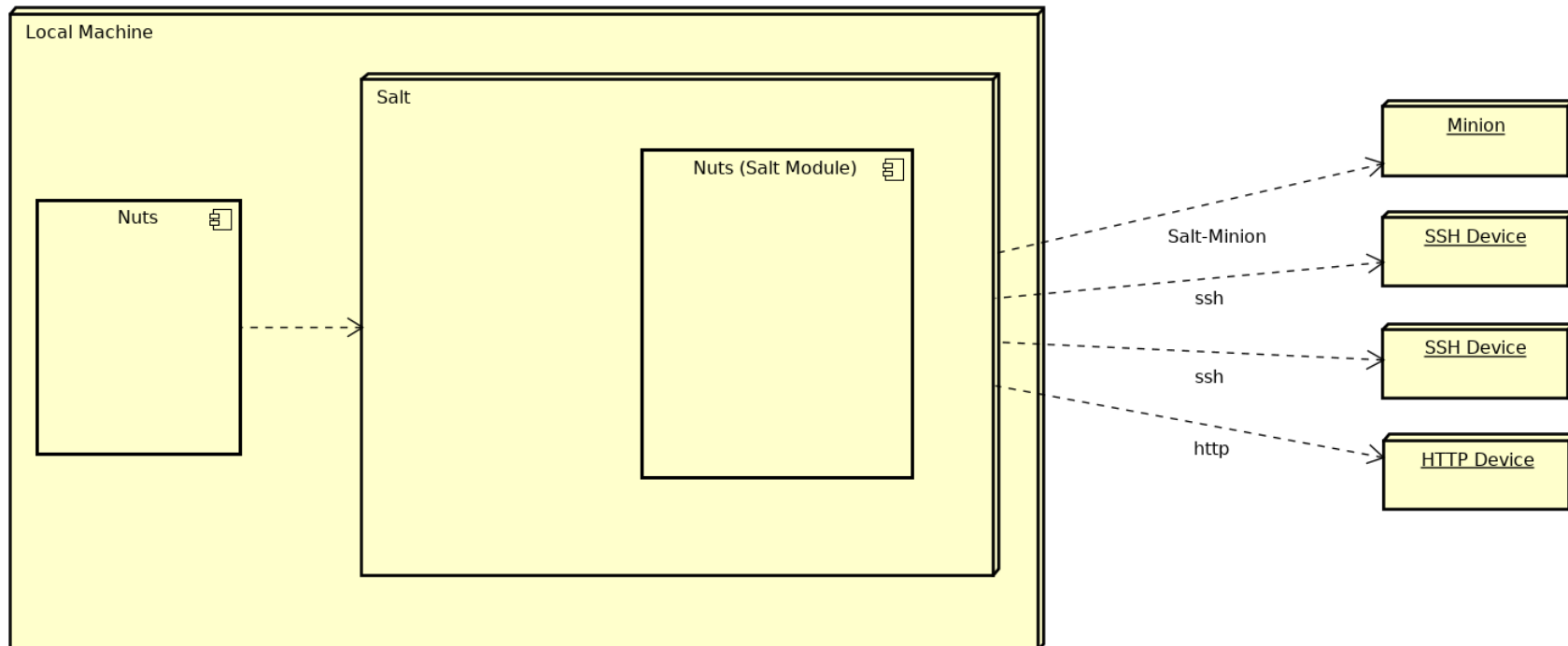


Abbildung 8: Deployment Diagramm

4.5. Sequenzdiagramm

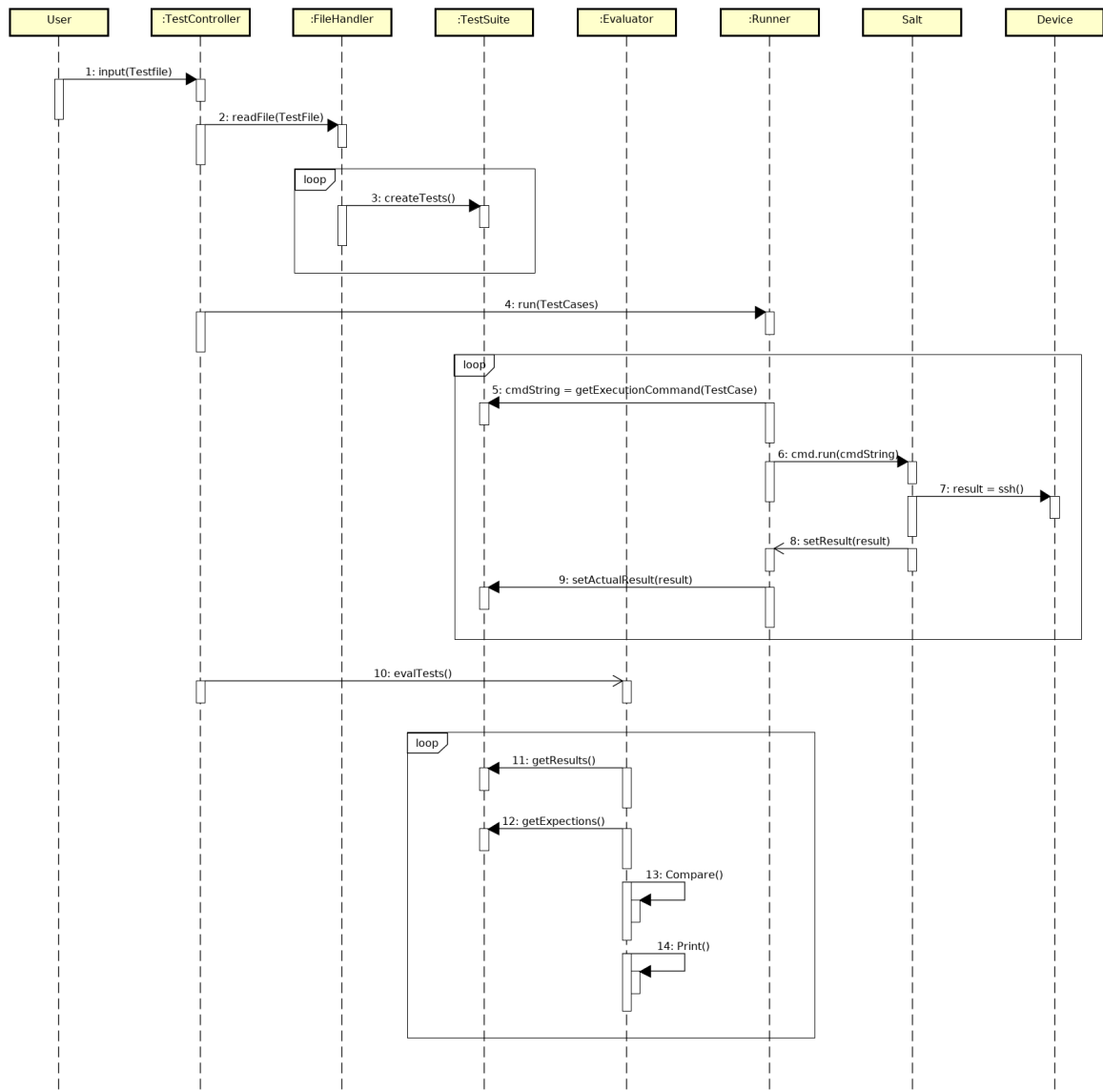


Abbildung 9: Sequenzdiagramm

4.6. Datensicherung

Es werden nur sehr wenige Daten gespeichert. Dazu gehört ein Result-Log pro Ausführung und ein Error Log für die Fehler.

4.6.1. Testdateien

Die erstellten Testdateien müssen von jedem Benutzer selber abgespeichert und verwaltet werden. Nuts speichert diese selber nicht ab.

4.6.2. Auswertungen

Jeder ausgeführte Test ergibt eine Log-Datei. Dieser Testlog wird unter `/var/log/` gespeichert. Die Datei beinhaltet die Testergebnisse und weitere Details, falls der Test nicht bestanden ist.

4.6.3. Error Log

Falls es beim ausführen des Programms zu einem Fehler kommt, wird dieser Fehler im Error Log erfasst. Das Error Log wird unter `error.log` abgespeichert.

5. Realisierung

5.1. Input Files

Bevor Unit Tests ausgeführt werden können, sind Input Files vom Benutzer zu erfassen. Diese Input Files werden vom System eingelesen, validiert und ausgeführt. Es musste ein Format gewählt werden, welches für Benutzer gut lesbar-, und vom System einfach zu parsen ist.

5.1.1. Formate

XML

XML bietet sich als weit bekanntes- und verbreitetes strukturiertes Format für Inputfiles an. Informationen könnten gut in Tags verpackt- und so einfach von Systemen geparkt werden. XML ist jedoch nicht besonders intuitiv von Hand zu schreiben und hat einen grossen Overhead.

```
<produkt >
  <id>blureg</id>
  <anzahl>4</anzahl >
  <beschreibung>Blauer Regenschirm</beschreibung >
</produkt >
```

JSON

JSON ist ebenfalls sehr bekannt und verbreitet. Verglichen mit XML ist es einfacher von Hand zu schreiben und hat einen geringeren Overhead. Das Parsing würde sich ebenfalls einfach gestalten. Für vom User zu erstellende Inputfiles würde sich das JSON Format besser eignen als das XML Format.

```
{
  "product": {
    "id": "blureg",
    "anzahl": "4",
    "beschreibung": "Blauer Regenschirm"
  }
}
```

YAML

Das für Benutzer wohl einfachste zu lesende und schreibende Format ist YAML. In den gängigen Configuration Management Tools wie Ansible oder SaltStack ist YAML bereits ein gängiges, unterstütztes Format. Informationen können auf sehr einfache Weise erfasst und hierarchisch strukturiert werden. Deshalb wurde YAML als Standard Input Format für die benötigten Input Files ausgewählt.

```
produkt :
  - id      : blureg
    anzahl  : 4
    beschreibung : Blauer Regenschirm
```

5.1.2. Test

Unit Tests müssen vom Benutzer im YAML Format erfasst werden. Es musste vorgängig überlegt werden, welche Angaben und Parameter für die Durchführung eines Tests notwendig sind. Die Struktur eines Unit Tests sieht folgendermassen aus:

```
- name :
  command :
  devices :
  parameter :
  operator :
  expected :
```

In einem Testfile sind mehrere Unit Tests möglich, mittels "#" am Zeilenanfang können die Tests auch mit Kommentaren versehen werden.

name

Das "name" Feld kann vom Benutzer frei beschrieben werden. Es dient dazu, einem Test einen treffenden Namen zu vergeben. Ein Beispiel wäre "OSPF Neighbors on Distribution Switch 01". Das "name" Feld ist ein Pflichtfeld.

command

Im "command" Feld muss der richtige Test aus dem Custom SaltStack Modul ausgewählt werden. Nach dem Einlesen des Testfiles wird die gleichnamige Funktion im SaltStack Modul ausgeführt. Es gibt ein 1:1 Mapping zwischen dem "command" Feld und der Funktion im SaltStack Modul "nuts.py". Das "command" Feld ist ein Pflichtfeld.

devices

Im devices Feld sind die Namen der im Test beteiligten Devices erfasst. Die Angaben der Devices selbst sind in einem eigenen File ausgelagert (siehe späteres Kapitel). Es gibt ein 1:1 Mapping zwischen dem "devices" Feld und dem "name" Feld im Device File. Das "devices" Feld ist ein Pflichtfeld.

parameter

Das "parameter" Feld unterscheidet sich in unterschiedlichen Tests stark. Gewisse Tests benötigen unterschiedliche Parameter, einige jedoch gar keine. Parameter Beispiele wären Portnummer, IP Adressen, Interfaces, etc. Das "parameter" Feld ist ein optionales Feld.

operator

Das "operator" Feld bezieht sich auf den Vergleich der im Feld "expected" getroffenen Annahme. So kann ">", "=" oder "<" gewählt werden. Häufig muss "=" gewählt werden, bei Traffic Tests beispielsweise eignet sich ein "grösser-" oder "kleiner als" meist besser. Das "operator" Feld ist ein Pflichtfeld.

expected

Im "expected" Feld muss eine Annahme über das Testresultat getroffen werden. Der Benutzer erwartet ein Resultat von einer Testausführung. Dieses erwartete Resultat wird nach der Testausführung mit dem tatsächlichen Resultat verglichen. Falls der Vergleich erfolgreich ist, ist auch der Test erfolgreich, andernfalls schlägt er fehl. Das "expected" Feld ist ein Pflichtfeld.

Beispiel Linux Tests

```
# Ping Test Linux
- name: Ping 8.8.8.8
  command: connectivity
  devices: arch
  parameter: ['8.8.4.4']
  operator: '='
  expected: True

# Ping Test Linux
- name: Ping www.google.com
  command: connectivity
  devices: arch
  parameter: ['www.google.com']
  operator: '='
  expected: True
```

Beispiel Arista Tests

```
# Arista User Test
- name: Test 7
  command: checkuser
  devices: arista
  parameter: []
  operator: '='
  expected: ['admin', 'eapi']

# Arista Version Test Linux
- name: Test 9
  command: checkversion
  devices: arista
  parameter: []
  operator: '='
  expected: '4.14.16M'
```

Beispiel Cisco Tests

```
# Cisco User Test
- name: Test 2
  command: checkospfneighborsstatus
  devices: DS1
  parameter: []
  operator: '='
  expected: ['10.0.2.1:FULL/DR', '10.0.0.6:FULL/BDR']

# Cisco User Test
- name: Ospf Neighbors DS1
  command: checkospfneighbors
  devices: DS1
  parameter: []
  operator: '='
  expected: ['10.0.2.1', '10.0.0.6']
```

Um die Felder "parameter" und "expected" korrekt zu setzen, sind Kenntnisse über die Output Formate und das Parsing notwendig. Sie können sich teils stark zwischen den Devices unterscheiden.

5.1.3. Device

Devices, welche in Unit Tests verwendet werden, müssen vorgängig in einem File im YAML Format erfasst werden. Mögliche Devices wären beispielsweise Router, Switches, Computer, etc. Die Struktur eines Devices sieht folgendermassen aus:

```
- name :  
  os :  
  destination :  
  username :  
  password :
```

In einem Devicefile ist die Erfassung mehrerer Devices möglich, mittels "#" am Zeilenanfang können die Tests auch mit Kommentaren versehen werden.

name

Das "name" Feld kann vom Benutzer frei beschrieben werden. Es dient dazu, einem Device einen treffenden Namen zu vergeben. Ein Beispiel wäre "DS01" für Distribution Switch 01. Es empfiehlt sich, einen kurzen Namen ohne Abstände und Sonderzeichen zu wählen, weil dieses Feld im Testfile referenziert werden muss. Das "name" Feld ist ein Pflichtfeld.

os

In diesem Feld muss das Betriebssystem des Devices angegeben werden. Es können nur vom System unterstützte Betriebssysteme angegeben werden. Momentan sind dies "arista", "linux" und "ios". Die Namen müssen richtig geschrieben werden und dürfen nicht mit Versionen oder weiteren Angaben versehen werden. Das "os" Feld ist ein Pflichtfeld.

destination

Im "destination" Feld muss das erfasste Device im Salt Roster File referenziert werden. Das "destination" Feld ist ein Pflichtfeld.

username

Hier muss der entsprechende Benutzername auf dem Device angegeben werden. Dies wird fürs Login auf das Device benötigt. Das "username" Feld ist ein Pflichtfeld.

password

Das Passwort für den entsprechenden Benutzer muss hier im Klartext erfasst werden. Das "password" Feld ist ein Pflichtfeld.

Beispiel Device File

```
- name: arch
  os: linux
  destination: arch
  username: '-'
  password: '-'

- name: DS1
  os: ios
  destination: DS1
  username: admin
  password: '12345678'
```

5.1.4. Validierung

Die vom Benutzer erfassten Input Files können vom System validiert werden. Input Files können auf 3 verschiedene Arten validiert werden:

- Testfile und Devicefile gleichzeitig
- Testfile einzeln
- Devicefile einzeln

Die verschiedenen Kommandozeilen Aufrufe lauten:

```
nuts -v <YAML Testfile> <YAML Devicefile>
nuts -vt <YAML Testfile>
nuts -vd <YAML Devicefile>
```

Die Validierung findet mit Hilfe des pykwalify Frameworks statt. Pykwalify ist eine Validation Library für YAML und JSON Files. Die FileValidator Klasse im Service Package vergleicht die mitgegebenen YAML Files mit den definierten Schema Files für Test- und Devicefiles. Sobald die Eingabe das entsprechende Schema verletzt, wird von pykwalify eine Fehlermeldung ausgegeben.

testSchema.py

```
type: seq
sequence:
  - type: map
    mapping:
      "name":
        type: str
        required: yes
        unique: yes
      "command":
        type: str
        required: yes
      "devices":
        type: str
        required: yes
      "parameter":
        type: any
        required: yes
      "operator":
        type: str
        required: yes
      "expected":
        type: any
        required: yes
```

deviceSchema.py

```
type: seq
sequence:
  - type: map
    mapping:
      "name": { type: str, required: yes, unique: yes }
      "os":
        type: str
        required: yes
      "destination":
        type: str
        required: yes
      "username":
        type: str
        required: yes
      "password":
        type: str
        required: yes
```

5.2. SaltStack

SaltStack wurde als Configuration Management Tool ausgewählt. Befehle für Unit Tests werden von nuts an SaltStack weitergegeben und von dort ausgeführt. Das SaltStack Custom Modul wird wie folgt angesprochen:

```
salt-call nuts.cmd device parameter os username password
```

Nach dem Ausführen der Befehle kommt man das Resultat als JSON Objekt zurück und kann weiter verarbeitet werden. SaltStack ist daher einfach ausgedrückt für die Deviceverwaltung und für das Ausführen von Befehlen auf den Devices zuständig.

5.2.1. Minions

Mit den Minions ist der Einsatz von SaltStack sehr einfach. Voraussetzungen dafür sind eine Python Umgebung auf dem gewünschten Endgerät. Danach installiert man den Salt-Minion und gibt an, welchen Hostnamen oder IP-Adresse der Salt Master hat. Sobald der Salt Master den Minion akzeptiert, kann sich der Master auf den Minion verbinden und Befehle ausführen. Diese Befehle werden direkt auf dem Minion ausgeführt und geben die Resultate an den Master zurück. So wird ein Befehl auf einem Minon ausgeführt:

```
salt 'minionname' cmd.run 'ping -c 3 8.8.8.8'
```

5.2.2. Salt-SSH

Wenn Python auf einem Gerät nicht unterstützt wird gibt es die Möglichkeit, sich per SSH zu verbinden. Dazu wird der von SaltStack entwickelte "salt-ssh" Client verwendet. Jedes Gerät das man per SSH ansprechen möchte, benötigt einen Eintrag im sogenannten Roster-File. Dieses liegt im Normalfall im Verzeichnis /etc/salt/ und heisst roster. Ein Roster Eintrag benötigt einen Namen, einen Host, ein Username und ein Passwort pro Gerät. Da diese Daten im Klartext vorliegen, sollte man die Datei mit den richtigen Berechtigungen versehen, nicht das alle diese Daten einsehen können. Hier sieht man ein Beispiel Roster File:

```
DistributionSwitch1:
  host: 10.0.0.9
  user: admin
  passwd: 12345678

AccessSwitch1:
  host: 10.0.1.201
  user: admin
  passwd: 12345678
```

5.2.3. Custom Module

Nuts im SaltStack ein eigenes Modul. Dieses Modul wird beim Salt Master erfasst und synchronisiert sich auf alle Salt Minions. Das Modul beinhaltet alle wichtigen Testbefehle, sowie Helpmethoden für das Parsen und Zurückgeben der Daten.

Hilfsmethoden return

Jedes Modul schickt seine verarbeiteten Daten zurück an den Salt Master. Damit die Daten leicht weiterverarbeitet werden können, wird das JSON Format verwendet. JSON wurde verwendet, weil die modernen Geräte alle RESTful API's anbieten und daher in den meisten Fällen mit JSON als Outputformatierung arbeiten. Es gibt zwei Möglichkeiten Daten zu retournieren, entweder einzelne Werte wie True/False oder eine Liste von Strings.

```
def returnMultiple(result):
    json_data = {}
    json_data["result"] = result
    json_data["resulttype"] = "multiple"
    return json.dumps(json_data)

def returnSingle(result):
    json_data = {}
    json_data["result"] = result
    json_data["resulttype"] = "single"
    return json.dumps(json_data)
```

Verarbeitung von Minion basierten Rückgaben

Minionbefehle sind sehr einfach zu implementieren. Man gibt den gewünschten Minion an, schickt den Befehl zu dem Minion und erhält die Antwort. Leider Die unterstützen

die meisten Befehle, wie beispielsweise ping, keine formatierte Ausgabe der Daten, sondern nur eine Plain Text Ausgabe. Dies bedeutet, dass jeder Rückgabewert mit einem regulären Ausdruck geparkt werden muss, um an die Daten zu gelangen. Hier sieht man ein einfaches Linux Beispiel:

```
def bandwidth(dst, host, os, user, pwd):
    if os == "linux":
        local.cmd(dst, 'cmd.run', ['iperf3 -s -D -1'])
        result = local.cmd(host, 'cmd.run', ['iperf3 -c '
            + dst])
        text = bytes(result).decode(encoding="utf-8",
            errors='ignore')
        regex = "([0-9.]{4})(\[a-zA-Z]bits\/sec)([\\s]*
            receiver)"
        r = re.compile(regex)
        m = r.search(text)
        return returnSingle(float(m.group(1)) *1000.0
            *1000.0)
```

Verarbeitung von XML Rückgaben

Momentan sind es nur die Cisco Geräte, welche XML retournieren. Für dieses XML Format gibt es die getCiscoXML() Methode. Man verbindet sich mit ßalt-ssh über das Gerät und sendet den entsprechenden Befehl. Der Befehl wird auf dem Gerät ausgeführt und man erhält das XML Objekt. Mit der xml.etree.ElementTree Library wird nun das XML geparkt und als Objekt an die Methode zurückgegeben.

```
def getCiscoXML(dst, param, cmd, attribut):
    value = master.cmd('cmd.run', ["salt-ssh " + dst + " -i -r
        ' " + cmd + " " + str(param) + " " + attribut + " |
        format flash:nuts.odm' --roster-file=/etc/salt/roster"
    ])
    xml = value[dst][value[dst].index('<'):len(value[dst])]
    return ET.fromstring(xml)
```

Die Methode sucht sich im XML die passenden Daten und gibt diese an den Salt Master zurück.

```
def checkospneighbors(dst, os, user, pwd):
    resultList = []
    if os == 'ios':
```

```
namespace = "{ODM://flash:nuts.odm//  
    show_ip_ospf_neighbor}"  
tree = getCiscoXML(dst, "", "sh ip ospf neighbor",  
    "")  
for id in tree.iter(tag=namespace + 'NeighborID'):  
    resultList.append(id.text)  
return returnMultiple(resultList)
```

Verarbeitung JSON Objekten

Das Verarbeiten von JSON Objekten ist mit Abstand die einfachste und eleganteste Variante. Mit der jsonrpc lib kann die REST API angesprochen werden und so hat man bereits ein JSON Format mit den gewünschten Daten. Dieses kann meist ohne grosse Anpassung zurückgegeben werden.

```
def checkversion(dst, os, user, pwd):  
    if os == 'arista':  
        switch = Server("http://" + user + ":" + pwd + "@"  
            + dst + "/command-api")  
        res = switch.runCmds(1, [ "show version" ])  
        return returnSingle(res[0][ "version" ])
```

5.3. Device Output Formats

Nach abgesetzten Kommandos auf den Devices liegen die Outputs in einem bestimmten Format vor. Die Outputs variieren sehr stark. Grundsätzlich können Outputs strukturiert oder unstrukturiert vorliegen. Die Vorteile an strukturierten Outputs sind enorm. Für die Weiterverarbeitung in Software müssen gewisse Informationen aus dem Output geparkt werden. Bei strukturierten Daten gestaltet sich diese Aufgabe weitaus einfacher als bei unstrukturiertem Plain Text. Bei den strukturierten Outputs trifft man meist auf XML und JSON Formate. Beide Outputs können grundsätzlich leicht über ihre Attributnamen geparkt werden.

JSON

JavaScript Object Notation oder kurz JSON ist ein weit verbreitetes Format. Die meisten Web-APIs setzen auf JSON als Rückgabeformat. Das Parsing gestaltet sich wegen den Attributen extrem einfach und macht dieses Format sehr angenehm, um Werte aus den Rückgaben zu extrahieren. Neue Devices unterstützen oft die Abfrage über RESTful HTTP mit JSON als Rückgabeobjekten. Leider ist dieses Format auf vielen Legacy Netzwerkdevices nicht unterstützt.

XML

Ähnlich wie bei JSON können auch XML Formate gut weiterverarbeitet werden. Die Informationen sind angenehm strukturiert und die Elemente können leicht über ihre Namen geparkt werden. XML Formate sind häufig auf Cisco Devices anzutreffen. Grundsätzlich können sämtliche "show" Befehle mittels XML formatiert zurückgegeben werden. Bei RESTful API's ist das XML Format eher unverbretet.

Bei Cisco Devices müssen die XML Strukturen für spezifische "show" Befehle zuerst manuell erstellt und in das Device importiert werden. Die sogenannten "ODM-Spec-Files". Das Erstellen dieser Spec-Files ist schlecht dokumentiert und Beispiele sind schwierig zu finden. Das Outputformat der show-Befehle ist entscheidend, ob ein Spec-File erstellt werden kann, welches die Anzeige in XML Form anzeigt. Wohlgeformte Tabellenformate lassen sich weitaus einfacher strukturieren als unformatierter Text.

Damit möglichst viele Outputs formatiert zurückgegeben werden kann, wurde ein Spec-File "nuts.odm" für benötigte "show" Outputs auf Cisco Devices erstellt. Bevor Cisco Devices mit nuts getestet werden können muss dieses Spec-File auf die Geräte geladen


```

arch:
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=45 time=18.3 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=45 time=21.0 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=45 time=20.8 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 18.304/20.070/21.038/1.250 ms

```

Abbildung 10: Raw-Text Beispiel

werden. Dafür ist ein TFTP Server notwendig. Mit dem Befehl "copy tftp: flash:" kann das Spec-File "nuts.odm" vom entsprechenden TFTP Server auf das Device geladen werden.

5.4. Output Parsing

Beim Output Parsing gibt es grundsätzlich drei grosse Varianten. Diese wären JSON, XML und Plain Text. JSON und XML sind ziemlich einfach zu parsen, da es bereits strukturierte Daten sind, während Plain Text sehr umständlich werden kann.

5.4.1. Plain Text

Hier sieht man ein Beispiel von Plain Text. Der ping Befehl unter Linux gibt dieses Format zurück. Hier gibt es keine andere Möglichkeit als einen regulären Ausdruck. Um an die richtigen Daten zu gelangen, muss der Rückgabewert zuerst von Byte zu String umgewandelt werden:

```
text = bytes(result).decode(encoding="utf-8", errors='ignore')
```

Danach wird ein Regex erstellt und dieser sucht im Text nach dem Pattern:

```

regex = "([0-9]*)% packet loss"
r = re.compile(regex)
m = r.search(text)

```

Hier im Beispiel wird nach "Packet loss" gesucht. Sobald dieser kleiner als 100% ist, ist die Connectivity bestätigt und der Test bestanden. Daher sucht man nach der Gruppe 1 im Regex, was die Prozentzahl bei Packet Loss ist und vergleicht diese mit 100.

```
return returnSingle((int(float(m.group(1))) < 100))
```

Da jeder Fall bei Plain Text unterschiedlich ist, ist es sehr aufwändig, die richtigen Daten zu bekommen.

5.4.2. XML

XML ist ein strukturiertes Datenformat. Das bedeutet, dass man keine regulären Ausdrücke mehr benötigt und nur noch auf die richtigen Indizes zugreifen muss. Speziell an XML sind die Namespaces. Diese werden benötigt, damit man im XML Baum die richtigen Einträge findet.

```
ns= "{ODM://flash:nuts.odm//show_ip_ospf_neighbor}"
```

Mit dem getCiscoXML Aufruf holt man sich nun die XML Daten vom gewünschten Gerät.

```
tree = getCiscoXML(dst, "", "sh ip ospf neighbor", "")
```

Nun kann man durch die Entries iterieren und überprüfen, ob der gewünschte Eintrag vorhanden ist.

```
for id in tree.iter(tag=namespace + 'entry'):  
    resultList.append(id.find(ns + 'NeighborID').text + ":" +  
        id.find(ns + 'State').text)
```

Wenn dieser gefunden wurde, fügt man ihn der resultList hinzu und gibt diese zurück an den Salt Master.

5.4.3. JSON

JSON ist wie XML bereits ein strukturiertes Datenformat. Daher ist auch das Parsen der Daten sehr einfach. Dies zeigt sich am Beispiel des Arista Check Version Befehls. Als ersten Schritt holt man sich die Daten von der REST API:

```
switch = Server("http://" + user + ":" + pwd + "@" + dst + "/"  
    command-api")  
res = switch.runCmds(1, [ "show version" ])
```

```

{
  "jsonrpc": "2.0",
  "result": [
    {
      "interfaceStatuses": {
        "Ethernet9": {
          "vlanInformation": {
            "interfaceMode": "bridged",
            "vlanId": 13,
            "interfaceForwardingModel": "bridged"
          },
          "bandwidth": 1000000000,
          "interfaceType": "1000BASE-T",
          "description": "",
          "duplex": "duplexFull",
          "autoNegotigateActive": true,
          "linkStatus": "connected"
        },
      }
    }
  ]
  "id": "CapiExplorer-123"
}

```

Nachdem man die Daten erhalten hat, kann man per Index sehr leicht darauf zugreifen und mit einem Befehl die Daten erhalten und zurückgeben.

```
return returnSingle(res[0]["version"])
```

5.5. Ergebnisanzeige

Damit die Ergebnisse korrekt angezeigt werden, müssen die Tests ohne Fehler durchlaufen. Dabei wird bei jedem Test von SaltStack das "actualResult" gegeben. Dieses wird im Test Case in einer Variable gespeichert.

Nachdem alle Tests ausgeführt wurden, startet der Evaluator und beginnt mit dem Vergleichen

```
def compare(self, testCase):
    if testCase.operator == "=":
        if testCase.actualResult["resulttype"] == "single"
        :
            return testCase.expectedResult == testCase
                .actualResult["result"]
        elif testCase.actualResult["resulttype"] == "
        multiple":
            return self.comp(testCase.actualResult["
                result"], testCase.expectedResult)
    elif testCase.operator == "<":
        return testCase.expectedResult < testCase.
            actualResult["result"]
    elif testCase.operator == ">":
        return testCase.expectedResult > testCase.
            actualResult["result"]
    elif testCase.operator == "not":
        return testCase.expectedResult != testCase.
            actualResult["result"]
```

Aus der Testcase Datei, wird der Operator ausgelesen und der richtige Case gewählt. Bei den "<", ">" und "!=" Operatoren sind nur Single Vergleiche möglich. Dies liegt daran, dass eine Liste nicht mit beispielsweise ">" verglichen werden kann.

Der "=" Operator ist ein wenig komplizierter. Hier gibt es zwei Möglichkeiten. Die erste Möglichkeit ist der Single-Vergleich. Dabei wird das "expectedResult" mit dem "actualResult" verglichen und "True" oder "False" zurückgegeben.

Bei einem Multiple-Vergleich werden zwei Listen miteinander verglichen. Dazu wurde eine Hilfsmethode verwendet, die wie folgt aussieht:

```
def comp(self, list1, list2):  
    if len(list1) != len(list2):  
        return False  
    for val in list1:  
        if val not in list2:  
            return False  
    return True
```

Sobald der Vergleich vollzogen wurde, wird das Resultat auf der Konsole angezeigt und in den Log geschrieben. Dies sieht dann so aus:

```
Ping 8.8.8.8: Test bestanden -----  
Expected: True = Actual: True  
Ping www.google.com: Test bestanden -----  
Expected: True = Actual: True
```

Abbildung 11: Testresultprint Beispiel

5.6. Error Handling und Logging

Zum Loggen aller Fehler und dem Schreiben der Testresultate in einer Log-Datei wird das von Python bereitgestellte Logging Modul verwendet. Bei jedem Start des Tools wird dieses wie folgt initialisiert:

```
def __init__(self):
    self.errorlogger = logging.getLogger('nuts_error_log')
    self.inforlogger = logging.getLogger('nuts_info_log')

    errorformatter = logging.Formatter('%(asctime)s - %(name)s
        - %(levelname)s - %(message)s')
    errorhandler = logging.FileHandler('/var/log/nuts/error.
        log')
    errorhandler.setFormatter(errorformatter)
    self.errorlogger.addHandler(errorhandler)

    dateTag = datetime.datetime.now().strftime("%Y-%b-%d_%H-%M
        -%S")
    infoformatter = logging.Formatter('%(asctime)s - %(message
        )s')
    infohandler = logging.FileHandler('/var/log/nuts/' +
        dateTag + 'testresults.log')
    infohandler.setFormatter(infoformatter)
    self.inforlogger.addHandler(infohandler)
```

Wie man sieht, werden zwei Logger verwendet. Ein Errorlog und ein Infolog. Dabei wird die Zeit, der Fehlername, das Loglevel und die Message gespeichert.

5.6.1. Errorlog

Um die Software sicher vor Abstürzen zu machen und um allfällige Fehler an einem zentralen Ort gespeichert zu haben, wurde ein Error Handling eingebaut. Der Error-Log schreibt alle entstandenen Fehler in die error.log Datei unter /var/log/nuts/. Die Fehler wurden dabei von dem Error Handling abgefunden. Hier sieht man ein Beispiel, wie der Log angesprochen wird:

```

except Exception as e:
    self.testSuite.setActualResult(testCase, json.loads('{
        resulttype": "single", "result": "ERROR"}'))
    print("Runner-Fehler beim Befehl: " + cmd)
    self.logger.exception("Error beim Befehl: " + cmd + '\
        nSalt Error:' + result + '\n\n')

```

Alle kritischen Methoden wurden mit Try/Except behandelt, damit die Fehler nicht zu einem Absturz führen. Die Fehler werden abgefangen und die Software wird in einen konsistenten Zustand gebracht. Hier ein Beispiel von dem FileHandler:

```

try:
    from yaml import CLoader as Loader, CDumper as Dumper
except ImportError:
    from yaml import Loader, Dumper

```

Alle Fehler, die beim Ausführen von Salt Modulen entstanden sind, werden an die Software weitergegeben. Hier sieht man ein Beispiel, wie die Fehler abgefangen werden. Wenn ein Fehler erkannt wird, wird eine Exception geworfen, welche ins Log gespeichert wird.

```

result = proc.communicate()[0].decode('utf-8')
if "ERROR" in result:
    raise Exception('Ein Salt Error ist aufgetreten!\n' +
        result)

```

5.6.2. Infolog

Der Info-Log wird für die Testergebnisse verwendet. Diese werden wie der Errorlog im Verzeichnis /var/log/nuts/ gespeichert. Als Name der Datei wird das aktuelle Datum und die aktuelle Zeit der Programmausführung verwendet.

Einzig in der Evaluatorklasse wird der Infolog verwendet. Hier ein Beispiel wie dieser befüllt wird:

```
1 Test 6: Test bestanden -----
2 Expected: 4505 4506 = Actual: 4505 4506
3 Test 8: Test bestanden -----
4 Expected: admin = Actual: admin
```

Abbildung 12: Log Example

5.7. Unit Testing

Für die Unit Tests Software wurde das Modul "pytest" verwendet. Neben dem Source Ordner (src) wurde eine zweite Ordnerstruktur aufgebaut. Diese hat den Namen Test. Darin befindet sich pro zu testende Klasse eine test_Klassenname.py Datei.

Jede dieser Testdateien wurde mit einer Setupmethode gestartet. Diese sieht wie folgt aus:

```
def setup_class(cls):
    cls.testSuite.createDevice("Server01", "Linux", "192.168.100.1",
                               "root", "1234")
    cls.testSuite.createDevice("Server02", "Linux", "192.168.100.2",
                               "root", "1234")
    cls.testSuite.createDevice("Switch01", "Cisco", "192.168.200.1",
                               "admin", "1234")
    cls.testSuite.createTest("testPingFromAToB", "connectivity", '
    Server01', '8.8.8.8', "=", 'True')
    cls.testSuite.createTest("checkuser", "checkuser", 'Server02',
                              '8.8.8.8', "=", 'admin')
    cls.testSuite.createTest("Count ospf neighbors", "
    countospfneighbors", 'Switch1', '', "=", '3')
```


Es wurden immer drei Devices und drei Tests erstellt.

Mit diesen Tests wurden alle relevanten Methoden der Klasse getestet. Dies heisst, dass alles ausser die `init()`-, `get()`-, `set()`-, und `print()`-Methoden getestet wurden. Hier sieht man ein einfaches Beispiel der `comp()`-Methode. Diese vergleicht zwei Listen miteinander und überprüft, ob beide Listen den gleichen Inhalt besitzen.

```
def test_compMultiple(self):  
    list1 = []  
    list2 = []  
    list1.append('admin')  
    list1.append('user')  
    list2.append('user')  
    list2.append('admin')  
    self.eval.comp(list1, list2) is True
```

Es wurde auf ein aufwändiges Unit Testing mit Faking und Mocking verzichtet. Stattdessen wurden nur Klassen mit unabhängigen Methoden getestet.

6. Ergebnisse

6.1. nuts

Mit "nuts" wurde eine lauffähige Software in Python entwickelt. Die erstellte Software Architektur entspricht dem geplanten Design. Unit Tests und Devices können im gut lesbaren YAML Format erfasst und vom entwickelten Tool ausgeführt werden. Die in der Anforderungsspezifikation geplanten Use Cases konnten vollständig umgesetzt werden.

Mit zusätzlichem Zeitaufwand wären einige Verbesserungen möglich. So könnte man ein grafisches User Interface erstellen. Dadurch würde sich die Usability stark verbessern. Ebenfalls könnte man einen Automatismus einbauen, welcher aus dem eingegebenen Device File das Salt Roster File erstellt, was momentan noch manuell erledigt werden muss. Mittels weiteren Refactorings, besserer Abdeckung durch Unit Tests und Ausbau des Error Handlings könnte die Robustheit der Applikation noch verbessert werden.

6.2. Deviceunterstützung

Der Kreis der unterstützten Devices ist wie erwartet eher klein. In dieser Arbeit wurde der Fokus auf das Testing der Devices über SSH Verbindungen gelegt. SaltStack bietet mit den Minions eine echte SSH Alternative welche vor allem auf Linux Devices eingesetzt wurde. Leider unterstützen ältere Cisco Devices kein Python, weshalb für diese Geräte keine Salt Minions eingesetzt werden konnten. Zudem wurden Unit Test Module für Arista Devices über RESTful HTTP entwickelt.

Wie erwartet gestaltete sich das Parsing der unterschiedlichen Outputs schwierig. Schnell wurde klar, dass für die Unterstützung der entwickelten Unit Tests auf mehreren Devices viel Fleissarbeit notwendig ist. Für jedes Device muss das Parsing des entsprechenden Outputs eigens entwickelt werden. Dieses Vorgehen ist nicht besonders elegant und es muss damit gerechnet werden, dass auf Basis von SSH Verbindungen eine Weiterentwicklung eher unwahrscheinlich ist. Weitere Unterstützung von Devices für Verbindungen über RESTful HTTP wären leicht durchführbar. Die strukturierten JSON Outputs sind weitaus einfacher zu verwenden.

6.3. Kommandos für Unit Tests

Im Kapitel "Testing in Netzwerken" wurden viele Test Cases beschrieben. Folgende Tests konnten tatsächlich umgesetzt werden:

Name	Devices	Beschreibung
connectivity	Linux, Cisco	Überprüft Connectivity mittels ping
traceroute	Cisco	Überprüft Routing Hops
bandwidth	Linux	Testet Verbindungs-geschwindigkeit
dnscheck	Linux	Überprüft DNS Response
dhcpcheck	Linux	Überprüft DHCP Response
webresponse	Linux	Überprüft HTTP Response
portresponse	Linux	Überprüft Erreichbarkeit über eine Portnummer
checkuser	Cisco, Arista	Überprüft die lokalen User auf einem Device
checkversion	Cisco, Arista	Überprüft die SW-Version auf einem Device
checkospfneighbors	Cisco	Überprüft ob alle OSPF Neighbors vorhanden sind
countospfneighbors	Cisco	Überprüft die Anzahl OSPF Neighbors
checkospf-neighborsstatus	Cisco	Überprüft den Status eines OSPF Neighbors
stpinterfacestate	Cisco	Überprüft den Spanning Tree Status pro VLAN
stpinterfacerole	Cisco	Überprüft die Spanning Tree Rolle auf einem Port
stpinterfacecost	Cisco	Überprüft die Spanning Tree Kosten auf einem Interface
stprootid	Cisco	Überprüft die Spanning Tree Root auf VLANs
stprootcost	Cisco	Überprüft die Root Kosten auf einem Spanning Tree Interface
stpvlaninterfaces	Cisco	Überprüft die Spanning Tree Interfaces auf einem spezifischen VLAN

stpvlanblockedports	Cisco	Überprüft alle geblockten Ports auf einem VLAN
vlanports	Cisco	Überprüft alle vorhanden VLAN auf einem Device
interfacestatus	Cisco	Überprüft den Status eines Interfaces (up/down)
interfacevlan	Cisco	Überprüft in welchem VLAN sich das Interface befindet
interfaceduplex	Cisco	Überprüft den Duplexmodus eines Interfaces
interfacespeed	Cisco	Überprüft den Link Speed eines Interfaces
cdpneighbor	Cisco	Überprüft die CDP Neighbors eines Devices
cdpneighborcount	Cisco	Überprüft die Anzahl CDP Neighbors eines Devices
showarp	Cisco	Überprüft die ARP Tabelle eines Devices

Tabelle 14: **Verfügbare Befehle**

Grundsätzlich waren die Module für Linux Devices einfach umzusetzen. Durch den Einsatz von Salt Minions gestaltete sich der Zugriff leicht. Jedoch war das Parsing der Outputs umständlich, da die verwendeten Tools keine formatierten Ausgaben mittels XML oder JSON unterstützen. Benötigte Informationen mussten mittels regulären Ausdrücken aus den Rückgabestrings geparkt werden.

Die getesteten Cisco Devices unterstützten lediglich Verbindungen über SSH. Daraus resultieren einige Nachteile. Man konnte in SaltStack pro geöffnete Verbindung jeweils nur einen einzelnen Befehl absetzen. Bei Cisco Devices ist dies häufig problematisch, da man sich für die meisten Befehle im "enable" Modus befinden muss. Um in den "enable" Modus zu gelangen, muss das "enable" Keyword auf der Console eingegeben werden. Es existiert ein Workaround, indem man dem lokalen User entsprechende Rechte vergibt, damit eine Eingabe des Enable Keywords nicht mehr notwendig ist. Dies ist nicht unproblematisch, da möglicherweise die Eingabe des enable secrets umgangen wird, sofern eines gesetzt wurde. Sämtliche IP SLA Befehle benötigen ein Setup, was

entsprechend mehrere Konsoleneingaben bedeutet. Leider konnten diese Schwierigkeiten nicht gelöst werden und es musste daher auf alle geplanten Tests über IP SLA verzichtet werden.

Die Outputs der Kommandos sind bei Cisco unterschiedlich formatiert. Manchmal können Informationen im XML Format abgerufen werden, häufig jedoch bleibt nur der Weg über reguläre Ausdrücke.

Alle Tests mit Arista Devices wurden über RESTful HTTP implementiert. Verglichen mit Cisco war kein vorgängiges Spezifizieren der Outputs mittels eigenem File notwendig. Die JSON Outputs sind leicht weiterzuverarbeiten und benötigen wenig Aufwand fürs Parsing der benötigten Informationen.

Durch die umgesetzten Tests ist bereits ein grosses Repertoire an Werkzeugen für die Überprüfung von Netzwerken vorhanden. Vor allem in den Bereichen Spanning-Tree und VLAN können bereits sehr viel Funktion getestet werden. Verbesserungen im Umfang könnten durch Hinzufügen von weiteren Tests erzielt werden. Im Bereich Routing müssten noch weitere Tests eingebaut werden, damit wichtige Aspekte des Layer 3 überprüft werden könnten.

7. Ausblick

Mit dieser Studienarbeit konnte eine Software für Unit Tests im Netzwerkumfeld entwickelt werden. Automatisiertes Unit Testing ist ein interessanter und sehr nützlicher Use Case. Eine Weiterentwicklung könnte für viele Netzwerk Administratoren lohnenswert sein.

Damit eine leichte und intuitive Bedienung mit einer grossen Unterstützung von Herstellern und Devices möglich ist, müssten weitere Arbeiten vorgenommen werden. Eine Unterstützung von SSH Verbindungen scheint zum heutigen Zeitpunkt sinnvoll, jedoch skaliert dieser Ansatz schlecht mit einer breiten Unterstützung unterschiedlicher Devices.

Es bleibt zu hoffen, dass die Hersteller zukünftig auf die Bedienung über REST API's setzen. Mit diesem Ansatz wäre eine generischere Entwicklung über mehrere Hersteller denkbar und weitaus leichter umzusetzen als über SSH.

8. Glossar

REST API	Interface, welches über HTTP erreichbar ist und standardisierte Funktionen bereitstellt
SNMP	Simple Network Management Protocol
Dashboard	Übersichtsanzeige
MSTP	Multiple Spanning Tree, ermöglicht mehrere Spanning Tree Instanzen (pro VLAN)
AS	Autonomous System
OSPF	Open Shortest Path First, weit verbreitetes Routing Protokoll innerhalb der Organisation
BGP	Border Gateway Protocol, Routing Protokoll im Internet
Link State Advertisements	Nachricht eines Routers an seine Nachbarn, enthält Topologieinformationen
SPF Tree	Shortest Path First Tree
Load Sharing	Lastverteilung über mehrere Links
Queueing Delay	Verzögerungen zwecks Warteschlangen
Path MTU	Maximale Rahmengrösse über einen Pfad
Auto Negotiation	Automatische Aushandlung von Link Speed und Duplex Modus
MPLS	Multiprotocol Label Switching, Overlay Technik für Routing auf Layer 2
Drop Policy	Firewall Regel, welche beim Match Pakete verwirft
Retransmissions	Erneutes Senden eines Datenpakets
JSON	JavaScript Object Notation, strukturiertes Datenformat
XML	eXtensible Markup Language, strukturiertes Datenformat
YAML	YAML ain't Markup Language, strukturiertes Datenformat
TFTP	Trivial File Transfer Protocol, Netzwerkprotokoll zur Übertragung von Dateien

Abbildungsverzeichnis

1.	Logischer Netzwerkplan	6
2.	HTTP Get Vorgang	15
3.	Use Case Diagramm	20
4.	Systemübersicht	29
5.	Klassendiagramm	30
6.	Software Architektur	36
7.	Dependence Diagramm	37
8.	Deployment Diagramm	40
9.	Sequenzdiagramm	41
10.	Raw-Text Beispiel	57
11.	Testresultprint Beispiel	61
12.	Log Example	64
13.	Projektphasen	2
14.	Risiken des Projekts	7
15.	Arbeitsstunden pro Kalenderwoche	9
16.	Aufwand der Projektphasen	10

II. Anhang

1. Projektplanung

1.1. Projektübersicht

Nach durchgeführten Changes im Netzwerkbereich wird die gewünschte Funktionalität meist von Hand getestet. In den meisten Fällen geschieht dies durch einfache Tools wie ping oder traceroute. Es wird vom durchführenden Netzwerk Engineer erwartet, dass er selbständig weiss, welche Funktionalität er nach durchgeführten Changes testen muss. Je nach Komplexität der Infrastruktur ist der Scope kaum in den Griff zu kriegen.

In der Software Entwicklung nutzt man schon lange Unit Tests. Optimalerweise werden für Softwareklassen zuerst Unit Tests geschrieben und anschliessend wird der Code implementiert. Somit existiert für jede Klasse und deren Methoden eine Testsammlung, welche bei späteren Changes immer ausgeführt werden kann.

Mit dieser Arbeit wird die Möglichkeit der Nutzung für Netzwerk Unit Tests angestrebt.

1.2. Zweck und Ziel

Die Studienarbeit soll den Nachweis der Problemlösungsfähigkeit unter Anwendung ingenieurmässiger Methoden nachweisen. Entsprechend verfügt die Arbeit über einen konzeptionellen, theoretischen und einen praktischen Anteil.

Mit Unit Tests für Netzwerkinfrastrukturen möchte man ein technisches Hilfsmittel für die Qualitätssicherung in der IT bereitstellen. Ähnlich wie in der Software Entwicklung soll eine Systematik entwickelt werden, um Komponenten im Netzwerk auf ihre Konfiguration und Funktionalität zu testen.

1.3. Projektorganisation

Vorname	Name	E-Mail
Andreas	Stalder	astalder@hsr.ch
David	Meister	dmeister@hsr.ch

Tabelle 15: Teammitglieder

Das Projekt wird von Prof. Beat Stettler und Urs Baumann betreut und benotet.

1.4. Management Abläufe

1.4.1. Zeitbudget

Der Projektstart ist am Montag, dem 22. September 2016. Die Projektdauer beträgt 14 Wochen, und das Projektende ist am Freitag, dem 23. Dezember 2016.

Während diesen 14 Wochen sind 240 Arbeitsstunden pro Projektmitglied eingeplant. Das entspricht pro Mitglied eine Arbeitszeit von ca. 18 Stunden pro Woche. Dies ergibt einen totalen Aufwand von ca. 480 Stunden.

Die wöchentliche Arbeitszeit von 18 Stunden kann bei Verzug oder bei unerwarteten Problemen auf maximal 24 Stunden erhöht werden.

1.4.2. Projektphasen

Das Projekt wird in fünf Phasen unterteilt: Initialisierung, Analyse, Design, Realisierung und Abschluss.

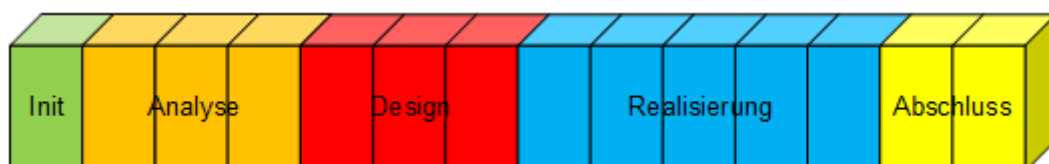


Abbildung 13: Projektphasen

1.4.3. Meilensteine

Das Projekt beinhaltet insgesamt fünf Meilensteine.

Meilenstein	Beschreibung	Datum
MS1	Anforderungen und Scope definiert	13.10.16
MS2	Architektur und Design beschrieben	03.11.16
MS3	Betaversion fertiggestellt	24.11.16
MS4	Software und Dokumentationen fertiggestellt	08.12.16
MS5	Arbeitsabgabe	23.12.16

Tabelle 16: **Projekt Meilensteine**

1.4.4. Iterationen

Die Dauer eines Iterationszyklus beträgt jeweils eine Woche.

Iteration	Inhalt	Start	Ende
Initialisierung	Projektstart und Kickoff-Meeting	15.09.2016	22.09.2016
Analyse 1	Projektplanung	23.09.2016	30.09.2016
Analyse 2	Netzwerk Tests Analyse	01.10.2016	06.10.2016
Analyse 3	Requirements & Evaluation	07.10.2016	13.10.2016
Analyse 4	Analyse Testcases detailliert	14.10.2016	20.10.2016
Design 1	Eval Configuration Management Tools	21.10.2016	27.10.2016
Design 2	Prototyp erstellen	28.10.2016	03.11.2016
Realisierung 1	FileHandler, Runner und Data	04.11.2016	10.11.2016
Realisierung 2	Evaluator, SaltStack Modul	11.11.2016	17.11.2016
Realisierung 3	Outputformatierung und Validierung	18.11.2016	24.11.2016
Realisierung 4	Loggin und Error Handling	25.11.2016	01.12.2016
Realisierung 5	TestLab und Unit Testing	02.12.2016	08.12.2016
Abschluss 1	Installationspackage und Doku	09.12.2016	15.12.2016
Abschluss 2	Schlussbericht	16.12.2016	23.12.2016

Tabelle 17: **Projekt Iterationen**

1.4.5. Arbeitspakete (Tickets)

Name	Inhalt	Iteration	Wer	Soll	Ist
Projektstart	Projektübersicht verschaffen	Initialisierung	alle	1	1
Projektplanung	Zeitplanung und Projektphasen, Iterationen, Meilensteine definieren	Analyse 1	alle	18	16
Netzwerktests Analyse	Recherche, Diskussion, Dokumentation	Analyse 2	alle	34	39
Risikomanagement	Projektrisiken abschätzen	Analyse 3	alle	8	8
Anforderungen	Use Cases definieren und NFA definieren	Analyse 3	alle	8	2
U Analyse Datenformate	Analyse der Datenformate (JSON, YAML, XML, Plain Text)	Analyse 3	alle	8	4
Analyse Testcases detailliert	Detaillierte Übersicht aller relevanten Testcases	Analyse 4	alle	24	31
Evaluation Configuration Management Tool	Vergleich von Ansible und SaltStack	Design 1	alle	32	34
Erstellung Prototyp	Erstellung eines ersten Prototypen	Design 2	alle	24	21
Architektur	Software Architektur und Design, Systemübersicht, Deployment Diagramm	Design 2	alle	16	27
FileHandler	Erstellen der Klasse FileHandler	Realisierung 1	alle	16	17
Data Layer	Erstellen des Data Layers	Realisierung 1	alle	16	10

Runner	Erstellen der Klasse Runner	Realisierung 1	alle	16	22
Evaluator	Erstellen der Klasse Evaluator	Realisierung 2	alle	24	32
SaltStack Module	Erstellen des SaltStack Modul	Realisierung 2	alle	40	38
Outputformatierung	Outputformatierung des SaltStack Moduls	Realisierung 3	alle	32	52
Filevalidierung	Erstellen der Klasse FileValidator	Realisierung 3	alle	16	5
Logging und Error Handling	Logging und Error Handling implementieren	Realisierung 4	alle	16	12
Aufbau Testlab	TestLab aufbauen	Realisierung 4	alle	48	38
Unit Testing	Unit Testing implementieren	Realisierung 5	alle	8	6
Refactoring	Refactoring des gesamten Codes	Realisierung 5	alle	24	20
Installationspackages erstellen	Package erstellen	Abschluss 1	alle	24	20
Benutzeranleitungen	Benutzer- und Installationsdoku schreiben	Abschluss 1	alle	16	12
Schlussbericht erstellen	Schlussbericht erstellen und abgeben	Abschluss 2	alle	32	42

Tabelle 18: **Arbeitspakete**

1.5. Risikomanagement

1.5.1. Risiken

Nr	Titel	Beschreibung	max. Schaden [h]	Eintrittswahrscheinlichkeit	Gewichteter Schaden	Vorbeugung	Verhalten bei Eintitt
R1	Probleme beim Einlesen/Parsen des Test-Files	Parsingprobleme mit YAML, Problem bei der Objekterzeugung	24	10%	2.4	Prototyp bis 3.11. klärt	Auf anderes Format (z.B JSON) ausweichen
R2	Probleme mit Roster File	Speichern/Laden von Devices und Credentials funktioniert nicht	24	10%	2.4	Prototyp bis 3.11. klärt	Ohne Roster File arbeiten
R3	Connectionprobleme in SaltStack	SSH Verbindung Auf- und Abbau auf Devices funktioniert nicht	48	10%	4.8	Prototyp bis 3.11. klärt	Muss gelöst sein, kein Workaround möglich
R4	Probleme beim Absetzen mehrerer Kommandos über SaltStack	z.B. IPSLA, "enable" auf Cisco Devices,	24	20%	4.8	Prototyp bis 3.11. klärt	Device anpassen (kein enable mehr nötig), IPSLA Tests streichen
R5	Probleme mit XML Output Formatierung/Parsing in SaltStack	XML Output von Devices schwierig zu Parsen, stark unterschiedliche Outputs	16	10%	1.6	Prototyp bis 3.11. klärt	Alternatives Outputformat prüfen, Plain Text nehmen und in SaltStack umwandeln
R6	Probleme mit JSON Output Formatierung/Parsing in SaltStack	JSON Output von Devices schwierig zu Parsen, stark unterschiedliche Outputs	24	30%	7.2	Prototyp bis 3.11. klärt	Alternatives Outputformat prüfen, Raw String nehmen und in SaltStack umwandeln
R7	Probleme mit String Output Formatierung/Parsing in SaltStack	String Output von Devices schwierig zu Parsen, stark unterschiedliche Outputs	48	50%	24	Prototyp bis 3.11. klärt	Unterstützte Devices einschränken, sodass Output einheitlicher wird
R8	Probleme beim Vergleich von erwarteten und tatsächlichen Testergebnissen	Gelieferte Testergebnisse müssen verglichen werden können	36	30%	10.8	Output Formatierung	Output Formatierung verbessern
Summe			244		58		

Abbildung 14: Risiken des Projekts

1.5.2. Umgang mit Risiken

Die aufgeführten Risiken sind in der Zeitplanung nicht speziell vorgesehen. Falls beim Eintreten eines geplanten Risikos ein erhöhter Zeitbedarf entsteht, so muss dies mit hoher Wahrscheinlichkeit mit Mehrarbeit der Teammitglieder kompensiert werden. Falls die nötige Mehrarbeit ausserhalb der Möglichkeiten liegt, so muss in Absprache aller Teammitglieder mit dem Betreuer nach einer anderen Lösung (z.B. Einschränkung von Programmfeatures) gesucht werden.

2. Zeitauswertung

Die Studienarbeit wird mit 8 ECTS Punkten honoriert. Pro ECTS Punkt wird mit einem Aufwand von ca. 30 Stunden gerechnet. Daraus ergibt sich für die Studienarbeit ein geplanter Zeitaufwand von 240 Stunden pro Teammitglied, also 480 Stunden Gesamtaufwand. Mit den erstellten Arbeitspaketen wurde ein Aufwand von 501 Stunden geschätzt. Die Endabrechnung zeigt einen ungefähren Zeitaufwand von 509 Stunden, was als Punktlandung angesehen werden kann. Daraus resultiert ein Gesamtaufwand von 254.5 Stunden pro Teammitglied. Die geforderten 240 Stunden pro Teammitglied wurden damit geringfügig überschritten.

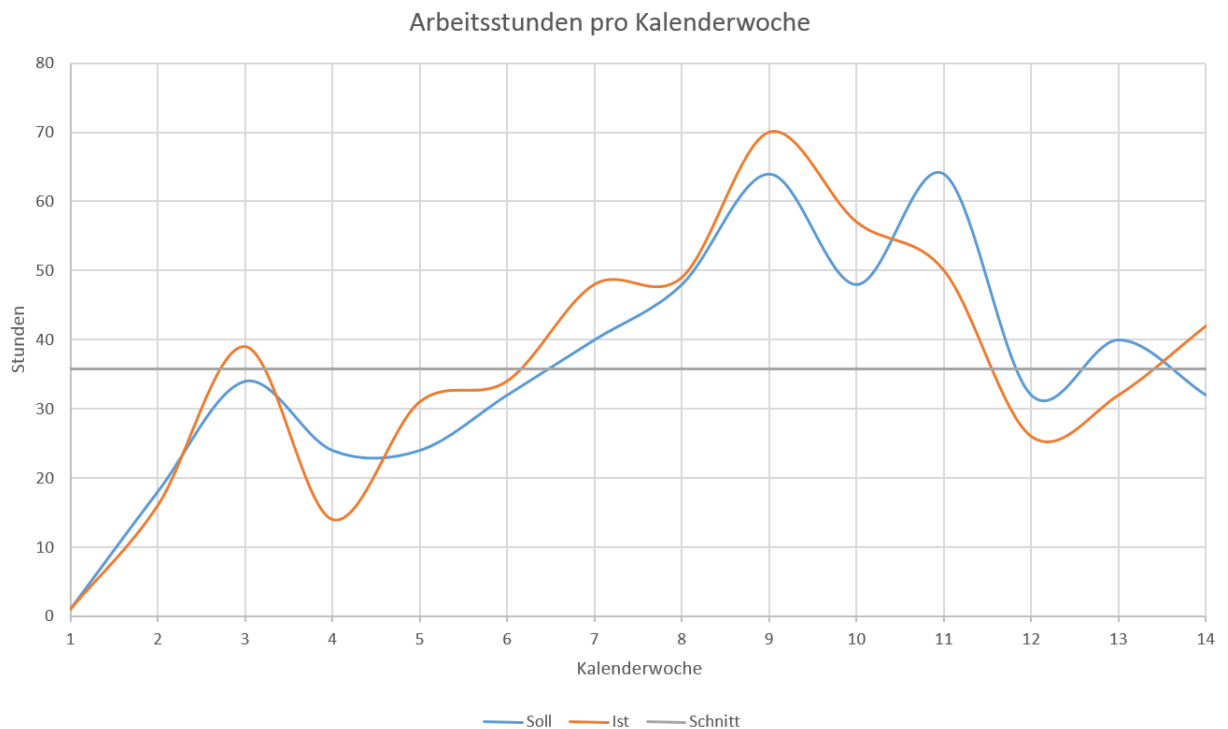


Abbildung 15: Arbeitsstunden pro Kalenderwoche

Die Zeitverteilung pro Kalenderwoche ist unterschiedlich. Anfangs wurde zum Teil deutlich zu wenig Zeit in das Projekt investiert, dafür wurde in der Realisierungsphase deutlich mehr Arbeitszeit investiert. Nachträglich hätte man die Phasen "Analyse" und "Design" verkürzen, und die Phase "Realisierung" verlängert werden sollen.

Aufwand Projektphasen

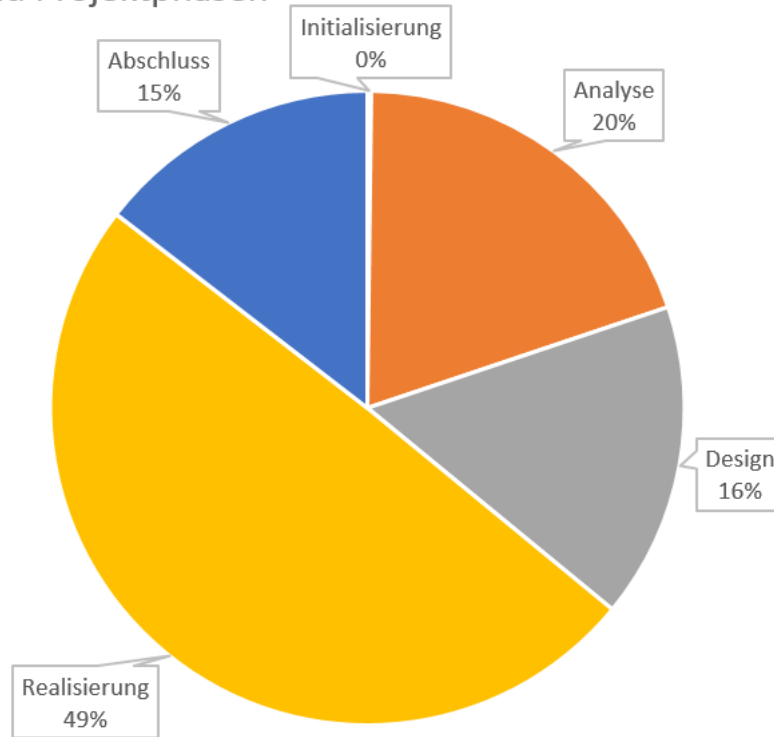


Abbildung 16: Aufwand der Projektphasen

3. Benutzeranleitung

3.1. Installation

3.1.1. Python

Für die Benutzung von nuts wird eine Python 2.7 Umgebung benötigt. Höhere Versionen werden momentan nicht unterstützt.

3.1.2. Arch Linux

Für Arch Linux gibt es ein fertiges Installationspaket. Dieses lädt man herunter:

```
wget https://github.com/asta1992/Nuts/blob/master/PKGBUILD
```

Nun wird das Programm mit makepkg installiert:

```
makepkg -sri
```

Alle Abhängigkeiten werden so direkt installiert und das Programm ist für die Verwendung bereit.

3.1.3. Ubuntu

Zuerst muss das aktuellste SaltStack Repository hinzugefügt werden:

```
wget -O - https://repo.saltstack.com/apt/ubuntu/16.04/amd64/latest/SALTSTACK-GPG-KEY.pub | sudo apt-key add -
```

Nun fügt man folgende Zeile in die "/etc/apt/sources.list.d/saltstack.list" Datei hinzu:

```
deb http://repo.saltstack.com/apt/ubuntu/16.04/amd64/latest xenial main
```

Danach holt man sich die neusten Updates:

```
sudo apt-get update
```

Und installiert sich SaltStack:

```
sudo apt-get install salt-master salt-minion salt-ssh salt-syndic salt-cloud salt-api
```

Zum Schluss installiert man sich noch nuts:

```
pip install nuts
```

3.1.4. Nuts

Wenn man bereits eine SaltStack-Umgebung installiert hat, muss man nur noch nuts installieren:

```
pip install nuts
```

3.2. SaltStack

Da SaltStack für dieses Programm wichtig ist, gibt es hier noch einen Link, der beim Einstieg in SaltStack sehr nützlich ist:

<https://docs.saltstack.com/en/latest/topics/tutorials/walkthrough.html>

3.3. SaltStack Modul

Um unser zusätzliches Nuts-Modul verwenden zu können, holt man sich das aktuelle Modul von GitHub:

```
wget https://github.com/asta1992/Nuts/blob/master/_modules/nuts.py
```

und kopiertes in den Ordner /srv/salt/_modules/

```
cp nuts.py /srv/salt/_modules/
```

Nun müssen alle Minion synchronisiert werden, damit sie das neue Modul finden:

```
sudo salt '*' saltutil.sync_all
```

3.3.1. SSH Geräte in Salt erfassen

Damit man per SSH auf entfernte Geräte zugreifen kann, muss pro Geräte ein Roster Eintrag erstellt werden. Hier findet man die genaue Anleitung von SaltStack:

<https://docs.saltstack.com/en/latest/topics/ssh/roster.html>

3.4. Cisco Spec-File

Um von den Cisco-Geräten die richtigen XML Rückgabewerte zu erhalten, muss man das Spec-File von GitHub herunterladen:

```
wget https://github.com/asta1992/Nuts/blob/master/_xml-spec/nuts.odm
```

Diese Datei kopiert man auf die Cisco-Geräte in den Flash mit dem Namen nuts.odm

3.5. Benutzung

```
nuts [-h] [-i INPUT INPUT] [-v VALIDATE VALIDATE] [-vt
    VALIDATETEST]
    [-vd VALIDATEDEVICE]
optional arguments:
  -h, --help            show this help message and exit
  -i INPUT INPUT, --input INPUT INPUT
                        Start a Testrun with a Testcase File and a
                        Device File
  -v VALIDATE VALIDATE, --validate VALIDATE VALIDATE
                        Validate a Testcase and Device File at the
                        same time
  -vt VALIDATETEST, --validatetest VALIDATETEST
                        Validate Testcase File
  -vd VALIDATEDEVICE, --validatedevice VALIDATEDEVICE
                        Validate Device File
```