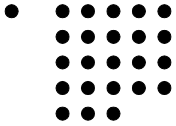

Fachhochschule Köln
Cologne University of Applied Sciences

Deutsches Zentrum
für Luft- und Raumfahrt

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science
im Studiengang Kommunikationssysteme und Netze

Automatisierte Tests verteilter Systeme unter Nutzung von Cloud-Ressourcen

Referent:	Prof. Dr.-Ing. Andreas Grebe FH-Köln
Korreferent:	Dipl.-Inform., Dipl.-Psych. Robert Mischke, DLR
vorgelegt am:	29.10.2012
vorgelegt von:	Hermann Stünz Gerhart-Hauptmann Str. 6 07318 Saalfeld

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.
Weiterhin versichere ich, dass die beiliegende CD-ROM und alle darauf enthaltene Daten auf Viren geprüft wurden und kein schädlicher, ausführbarer Code enthalten ist.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Saalfeld, den 29.10.2012

_____

Hermann Stünz
Gerhart-Hauptmann Str. 6
07318 Saalfeld

Kurzfassung

Zunächst werden entsprechende Grundlagen auf dem Gebiet der Cloud Technologien behandelt. Anschließend werden anhand von Anwendungsszenarien Anforderungen an die Implementierung zur Durchführung automatisierter, verteilter Tests ermittelt. Es folgen Evaluierungen zu Public und Private Compute Cloud Systemen sowie Cloud Abstraction APIs. Zur Vorbereitung der Testumgebung werden Hypervisor und Netzwerkanbindung konfiguriert. Weiterhin wird die Konzeptionierung der Distributed Test API in vier Phasen erläutert. Abschließend erfolgen die Implementierung des Konzepts sowie notwendige Systemtests.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Motivation

Die Einrichtung Simulations- und Softwaretechnik (SC) des DLR entwickelt zurzeit auf Basis von Eclipse das Software-Framework Remote Component Environment (RCE). Dieses wird als Basissystem für verschiedene Simulationsumgebungen im DLR eingesetzt. RCE ist ein verteiltes System, bei dem Server und Clients, deutschlandweit an den verschiedenen Standorten des DLR laufen. Momentan wird RCE nur für den nicht-verteilten Fall automatisiert getestet. Da dies nun auch für den verteilten Fall durchgeführt werden soll, ist im Rahmen dieser Arbeit eine Lösung zu entwickeln, die eine entfernte Testausführung unter Einbeziehung mehrerer RCE-Instanzen implementiert.

1.2 Ziel

Ziel der Arbeit ist es, eine Java API für automatisierte Tests von verteilt laufendem Java-Code zu entwickeln und dabei die Dienste einer Cloud, zur Bereitstellung der benötigten Rechenressourcen zu nutzen.
Momentan werden die verteilten Tests manuell ausgeführt und dabei beispielsweise nach dem Starten der RCE-Client- und Serverinstanz, die jeweiligen Funktionen manuell getestet. Hierbei werden keine strukturierten Testpläne durchlaufen, sondern ausschließlich manuelle System- bzw. Integrationstests durchgeführt. Weiterhin finden JUnit-Tests der Netzwerkschicht statt, bei denen lokal Server-Ports geöffnet werden, um die Übermittlung von RPC-Aufrufen zu prüfen.
Im Rahmen der Arbeit, gilt es zunächst ein Konzept für die API zu erstellen, um dann die Implementierung am Beispiel von RCE vorzunehmen. Die Verwaltung der Rechenressourcen einer Cloud, ist über eine entsprechende API des verwendeten Cloud Systems durchzuführen. Weiterhin sind IaaS-Systeme verschiedener Cloud Provider unter Berücksichtigung unterstützter Cloud Abstraction APIs zu evaluieren.
Die entwickelte API soll es ermöglichen, über eine geeignete Benutzerschnittstelle, Cloud Ressourcen für die Durchführung verteilter Tests zu reservieren. Die zu testenden Software-Instanzen sind dann auf ausgewählte, automatisiert vorkonfigurierte Systeme(VMs), aufzubringen und ggf. zu starten. Die durchzuführenden Tests sollten ebenfalls über die Benutzerschnittstelle übertragen, gestartet und ausgewertet werden können. Nach dem Abschluss der Tests sind die Testergebnisse an geeigneter Stelle zu sammeln und dem Anwender zur Verfügung zu stellen. Die für den Test reservierten Cloud Ressourcen sind anschließend wieder freizugeben.

1.3 Aufbau der Arbeit

Zunächst werden in Kapitel 2 Grundlagen Vermittelt, die sich mit dem Thema Cloud Computing sowie RCE auseinandersetzen.

Kapitel 3 beschreibt die Vorgehensweise beim Konzeptentwurf und führt dabei von der Anforderungsdefinition über die Evaluierung von Public und Private Compute Cloud Systemen sowie Cloud Abstraction APIs, zur Definition des Konzepts.

Nachfolgend werden in Kapitel 4 Konfigurationen zur Gewährleistung infrastrukturellen Voraussetzungen beschrieben und danach das Vorgehen bei der Implementierung erläutert. Weiterhin erfolgt die Darlegung der vorgenommenen Systemtests.

Abschließend wird in Kapitel 5 das Ergebnis der Arbeit zusammengefasst und ein Ausblick auf mögliche weiterführende Arbeiten gegeben.

2 Grundlagen

2.1 Cloud Computing

Cloud Computing beschreibt ein Modell zur Bereitstellung eines allgegenwärtigen, effizienten, bedarfsgesteuerten Netzwerkzugriffs auf ein Pool von konfigurierbaren Rechenressourcen (z.B. Netzwerke, Server, Datenspeicher, Anwendungen und Dienste), die mit einem minimalen Verwaltungsaufwand sowie minimaler Providerinterkation schnell zur Verfügung gestellt und wieder freigegeben werden können. Die Verwaltung und Optimierung der eingesetzten Ressourcen erfolgt dabei über das Cloud System mittels Messverfahren, die an die Abstraktionsebene des entsprechenden Servicetyps (z.B. Rechenkapazität, Bandbreite, aktive Benutzerzugänge) angepasst sind.
Es wird weiterhin zwischen drei Dienstmodellen unterschieden. Im Rahmen von Software as a Service (SaaS), werden dem Kunden auf der Cloud Infrastruktur des Providers betriebene Anwendungen bereitgestellt. Es ist dabei nicht vorgesehen, dass die Cloud Infrastruktur durch den Anwender gesteuert wird. In der Regel stehen diesem nur begrenzte, Nutzer-spezifische Konfigurationsmöglichkeiten der Anwendung zur Verfügung. Platform as a Service (PaaS) bietet dem Kunden die Möglichkeit, selbst entwickelte oder erworbene Anwendungen auf der Cloud Infrastruktur des Providers zu betreiben. Diese müssen dafür auf den vom Provider unterstützten Programmiersprachen, Bibliotheken und Diensten basieren. Es ist ebenfalls nicht vorgesehen, dass die Cloud Infrastruktur durch den Anwender gesteuert wird, allerdings besitzt dieser hier die Kontrolle über die betriebenen Anwendungen und Konfigurationsmöglichkeiten der Umgebung des Anwendungs-Hostings. Das Modell des Infrastructure as a Service (IaaS) beschreibt die Bereitstellung von Rechenkapazität, Datenspeicher, Netzen und weiteren grundlegenden Rechenressourcen. Der Nutzer kann beliebige Anwendungen sowie Betriebssysteme installieren, betreiben und steuern. Die darunterliegende Cloud Infrastruktur wird auch hier ausschließlich vom Provider kontrolliert, allerdings existieren eingeschränkte Möglichkeiten zur Steuerung ausgewählter Netzwerkkomponenten (z.B. Host Firewalls).
Cloud Computing wird zur Abgrenzung von Nutzern sowie Betreibern in vier Organisationsformen unterteilt. In einer Private Cloud nutzt ausschließlich eine einzige Organisation die bereitgestellte Cloud Infrastruktur. Diese kann durch die Organisation selbst oder Drittanbieter, inner- oder außerhalb eigener Räumlichkeiten Betrieben und Verwaltet werden. Eine Public Cloud kann von geschäftlichen, akademischen oder staatlichen Organisationen, in den Räumlichkeiten des Providers betrieben und von der Allgemeinheit genutzt werden. Die Organisationsform einer Community Cloud beschreibt die Bereitstellung der Cloud Infrastruktur für einen bestimmten Kundenkreis aus Organisationen mit gemeinsamem Hintergrund. Verwaltung und Betrieb übernehmen hier eine oder mehrere Organisationen innerhalb der Gemeinschaft oder ein Drittanbieter. Einen Mischansatz zwischen zwei oder mehreren Organisationsformen stellt die Hybrid

Cloud dar, bei der die verschiedenen Cloud Infrastrukturen über standardisierte oder proprietäre Verfahren verbunden werden. [1]

2.2 Remote Component Environment (RCE)

Das im Rahmen dieser Arbeit, häufig als zentrales Testobjekt verwendete RCE ist eine Integrationsplattform, die verschiedene Dienste zur Verfügung stellt, um kollaboratives Arbeiten in einer verteilten Umgebung zu ermöglichen. Übertragung und sicherer Zugriff auf verteilte Daten werden für den Entwickler transparent durch das RCE-Basissystem weitgehend automatisiert bereitgestellt.
RCE basiert auf „Eclipse™ Equinox", einer OSGi-Implementierung sowie auf der Eclipse Rich Client Platform. Teil der OSGi Service Plattform sind neben Aspekten der Modularisierung von Software-Komponenten in so genannte Bundles, Konzepte wie das Management von Zuständen, die ein solches Bundle annehmen kann. Software-Komponenten, die als Bundle in OSGi registriert sind, können über das Lifecycle-Management während der Laufzeit des Frameworks (de-)installiert, gestartet und gestoppt werden.
Die Software-Architektur von RCE basiert auf diesen Konzepten, indem all ihre Komponenten als Bundles realisiert sind. RCE stellt zu den von Eclipse™ Equinox bereitgestellten Standard-Services, wie beispielsweise Logging, Preferences, usw. eigene Dienste zur Verfügung. Diese bieten dem Anwendungsentwickler Funktionalitäten, die das Zusammenspiel verteilter Komponenten organisieren. Ein Service Broker ermöglicht das Auffinden von Diensten auf einer entfernten RCE Installation.
Die Kommunikation zwischen verschiedenen RCE-Instanzen wird durch den RCE-eigenen Communication Service realisiert. Dieser Dienst bietet verschiedene Kommunikationsprotokolle, wie zum Beispiel RMI (Remote Method Invocation), SOAP (Simple Object Access Protocol) oder RS (Remote OSGi Services) zur Verwendung an. Die Auswahl des Protokolls erfolgt abhängig von der Position der Kommunikationspartner[9]:

- Bundles im selben RCE,
- mehrere RCE-Instanzen auf der gleichen Maschine,
- Intranet-Kommunikation oder
- Internet-Kommunikation

3 Konzeptentwurf

3.1 Anforderungsdefinition

Die Anforderungen an die zu entwickelnde API, die Funktionalitäten zur Realisierung automatisierter Tests verteilter Anwendungen bereitstellen soll, wurden auf der Grundlage von RCE-Testfällen mit Verteilungsaspekt ermittelt. Den Input zu den nachfolgend aufgeführten Testfällen, die derzeit teils manuell ausgeführt werden, teils noch nicht realisiert sind, lieferten RCE-Entwickler, die anschließend die Nutzer der API darstellen.

Erfolgreicher Systemstart auf unterstützten Platformen
Zum Ermitteln eines erfolgreichen Systemstarts, ist auf einem 32- bzw. 64-Bit Windows oder Linux Betriebssystem zunächst die vorausgesetzte Software zu installieren und anschließend RCE zu beschaffen und extrahieren. Die Beim Start erzeugte Log-Datei, kann dem Anwender dann zur Auswertung zurückgegeben werden.
Im zu nutzenden Cloud System sollten hierfür die Betriebssysteme Ubuntu Server, OpenSUSE, Windows 7, Windows XP jeweils in 64- und 32-Bit Version, in Form von VM-Images zur Verfügung stehen. Des Weiteren wird eine Funktion zur Installation von Software sowie eine Möglichkeit zum Kopieren von Dateien zwischen der VM und dem Test-initiierenden System benötigt.

Kompatibilitäts-Test mit verschiedenen JVMs
Vor dem Starten der Anwendung ist zunächst die entsprechende JVM zu installieren, um anschließend das Verhalten bei der Ausführung, aus dem Log zurück zu gewinnen.
Hierfür werden ebenfalls eine Funktion zur Installation von Software auf der gestarteten VM sowie eine Möglichkeit zum Kopieren von Dateien zwischen der VM und dem Test-initiierenden System benötigt.

Kompatibilitäts-Test mit verschiedenen Python-Versionen
Innerhalb RCE verwenden verschiedene Komponenten Python zur Ausführung von Skripten. Um die Kompatibilität mit verschiedenen Versionen festzustellen, ist vor dem Starten der Anwendung zunächst die entsprechenden Version zu installieren, um anschließend das Verhalten bei der Ausführung, aus dem Log zurück zu gewinnen.
Hierfür werden ebenfalls eine Funktion zur Installation von Software auf der gestarteten VM sowie eine Möglichkeit zum Kopieren von Dateien zwischen der VM und dem Test-initiierenden System benötigt.

Verbindungsaufbau zwischen RCE-Instanzen
Die auf verschiedenen Systemen gestarteten RCE-Instanzen können über 3 unterschiedliche Wege, die jeweils anderen Instanzen kennenlernen. Möglich ist dies via Discovery mittels Broadcast-Anfrage, statischer Konfiguration über eine Konfigurationsdatei oder der Abfrage von Daten zu bekannten Instanzen, von einer Instanz, mit der Bereits ein Verbindungskontext besteht. Das Ergebnis des Kommunikationsvorgangs wird in einer Log-Datei protokolliert.
Um Verschiedene Netzwerkszenarien nachzustellen, ist eine variable Anzahl an VMs mit RCE-Instanzen, sowie die Möglichkeit zur Erstellung von Teilnetzen erforderlich.

Kompatibilitäts-Test zwischen bestimmten RCE-Builds
Der Kompatibilitäts-Test von RCE-Versionen mit unterschiedlichem Entwicklungsstand erfordert das starten von entsprechenden RCE-Instanzen, auf unterschiedlichen VMs sowie die Anschließende Kontrolle des durchgeführten Verbindungsaufbaus über die Log-Datei.

Netzwerkkommunikation über IPv4 und IPv6
Da der Betrieb von RCE in Zukunft auch in IPv6 Netzen ermöglicht werden soll, ist auch hier eine Fehlerfreie Kommunikation sicherzustellen. Des Weiteren ist der Einsatz von Dual-Stack zu testen, bei dem die Kommunikation sowohl zwischen einer IPv4- als auch einer IPv6-Instanz mit einer Dual-Stack-Instanz zu überprüfen ist. Für detailliertere Tests des Communication-Stacks sind entsprechende JUnit-Tests auf den beteiligten VMs auszuführen.
Hierfür ist es erforderlich, dass über das Cloud System auch IPv6 Netze konfiguriert werden können. Die weiteren Netzwerkeinstellungen für Dual-Stack, können dann über das Ausführen von Skripten auf den VMs erfolgen. Für die Ausführung von JUnit-Tests auf verschiedenen VMs ist die Einführung eines Mechanismus für die Steuerung verteilter JUnit-Tests erforderlich.

Benchmarking von Datei- und Stream-Übertragungen
Zur Messung der Leistungsfähigkeit von Datei- und Stream-Übertragungen zwischen RCE-Instanzen, sollen zunächst eine bestimmte Anzahl von Dateien, mit einer vorgegebenen Größe und zufälligem Inhalt erzeugt und anschließend übertragen werden. Hier sind sowohl die Zeit für die Übertragung als auch aufgetretene Übertragungsfehler festzustellen.
Diese Untersuchungen sind ebenfalls über die Durchführung von verteilten JUnit-Tests abzuwickeln.

Unterstützte Anzahl an kommunizierenden RCE-Instanzen
Um zu testen, ob auch bei einer hohen Anzahl an kommunizierenden RCE-Instanzen eine fehlerfreie Kommunikation ermöglicht werden kann, ist zunächst die entsprechende Menge an VMs bereitzustellen. Anschließend sind verteilte

JUnit-Test ausgeführt werden, die das Verhalten kritischer Komponenten überprüfen.

Neben aus den Testfällen gewonnenen Anforderungen sind noch folgende Rahmenbedingungen zu berücksichtigen.
Um bei der Verwaltung der Rechenressourcen einer Cloud über eine entsprechende API, unabhängig von einem bestimmten Cloud Provider bzw. System zu bleiben, ist eine Cloud Abstraction API zu verwenden. Dies würde des Weiteren auch die Durchführung eines Wechsels, zwischen einer Public und Private Cloud, mit geringem Aufwand ermöglichen. Zudem sollte die zu entwickelnde API, nach Möglichkeit keinen Kontext der zu testenden, verteilten Anwendung enthalten, da diese so auch für Tests anderer Entwicklungen verwendet werden könnte.
Innerhalb der DLR-IT und auch der IT, der Einrichtung SC, werden in verschiedenen Bereichen Virtualisierungstechnologien verwendet. Der betreibende IT-Dienstleister bietet hier, mit verschiedener Software vorinstallierte, an die Sicherheitsrichtlinien des DLR angepasste Betriebssystem-Images an. Diese sind in verschiedenen Windows- und Unix-Versionen erhältlich und werden auch für reguläre Arbeitsplatzinstallationen verwendet. Es wäre sinnvoll, die beschriebenen RCE-Tests auch auf einem solchen „DLR-System" durchzuführen, da ein Großteil der Nutzergemeinde ein solches verwendet. Daraus ergibt sich das Bedürfnis nach einer Möglichkeit zum Import von Images, im zu verwendenden Cloud System. Da der IT-Dienstleister nur VMware VMDK-Images anbietet, wäre es nützlich, wenn diese unterstützt werden würden, um so eine für die Weiterverwendung notwendige Konvertierung einzusparen.
Für den Betrieb von RCE und die durchzuführenden Tests auf einer VM, sollten hier jeweils etwa 2GB Arbeitsspeicher, mindestens 2GB freier Festplattenspeicher sowie die Leistung einer Single Core CPU mit ca. 1,5 GHz zur Verfügung stehen.
Weiterhin werden in der Abteilung SC, im Rahmen von Entwicklungen verwendete Fremdkomponenten bzw. -systeme, nach Möglichkeit aus dem Open-Source-Bereich gewählt. Daher sollten auch die in dieser Arbeit zu verwendenden Komponenten sowie Systeme stets frei erhältlich und zu Nutzen sein.

3.2 Konzeptidee

Nachfolgend werden die Ersten Vorstellungen zur Entwicklung des Konzepts wiedergegeben. Diese ergaben sich aus der Recherche über Technologien und Verfahren in den Bereichen Cloud Computing, Virtualisierung und Softwaretests sowie den in Kapitel 3.1 ermittelten Anforderungen an die API zur Realisierung automatisierter Tests von verteilten Anwendungen.
Die Initiierung eines Tests kann über einen JUnit-Test erfolgen, da dieser vom Entwickler direkt aus der Entwicklungsumgebung heraus ausgeführt, oder über einen Continous Integration Server automatisiert, von einem Built-Knoten gestartet wird. Hier ist es dann möglich, einen Aufruf zur Steuerung der verwendeten Cloud Ressourcen, über die zu entwickelnde API auszuführen, wodurch eine

bestimmte Anzahl an VMs, mit der gewünschten Platform erstellt und gestartet werden. Die Weitere Steuerung der VMs auf Betriebssystemebene, kann anschließend über eine SSH-Verbindung und entsprechende Befehlsaufrufe, die u.U. in einem Skript zusammengefasst werden, erfolgen. So wäre es möglich das System mit der Installation notwendiger Software vorzubereiten sowie die Beschaffung und Konfiguration der zu testenden Anwendung durchzuführen. Beim Starten der Anwendung wird davon ausgegangen, dass für einen Instanztest relevante Ereignisse, in eine Log-Datei geschrieben werden. Über diese erfolgt dann, je nach durchgeführtem Test, die Ermittlung von Informationen zur Ergebnisbildung. Weiterführende Tests können mit Hilfe von JUnit-Tests erfolgen, die für zu überprüfende Komponenten erstellt und auf den VMs verteilt ausgeführt werden. Die Steuerung der verteilten JUnit-Tests kann hier möglicherweise durch vorhandene Frameworks übernommen werden. Die dem Entwickler zurückgelieferten Ergebnisse können sowohl bei verteilten Instanz- als auch bei verteilten JUnit-Tests, nur die zwei Zustände, erfolgreich oder fehlgeschlagen, annehmen. Dies ist durch das Konzept des initialen JUnit-Test vorgegeben.

3.3 Evaluierung von Public Compute Cloud Systemen

Eine Möglichkeit zur Bereitstellung der Rechenressourcen besteht in der Inanspruchnahme von Leistungen eines Cloud Providers. Es wird auch von Compute Cloud Systemen gesprochen, um Produkte wie Storage Clouds bzw. Object Stores auszuschließen. Hier muss für die Zeit, die eine erstellte VM in Betrieb ist und je nach Provider, die über das Internet übertragene Datenmenge, gezahlt werden. Auswahlkriterien stellen die in Kapitel 3.1 beschriebenen Anforderungen, die sich an das zu verwendende Cloud System richten, dar. Da alle untersuchten Provider Funktionalitäten zum Erstellen, Starten und Stoppen einer VM bieten, wird auf diese Merkmale nachfolgend nicht mehr eingegangen. Im Vordergrund der Evaluierung stehen hier die erweiterten Funktionalitäten, die nicht von allen Providern unterstützt werden. Diese Punkte umfassen:

- Import von vorinstallierten Betriebssystem-Images
- Ressourcenüberwachung der in Betrieb befindlichen VMs (für z.B. CPU-, Speicherauslastung)
- Unterstützung von IPv6
- Konfigurationsmöglichkeit für Subnets
- Unterstützung durch Cloud Abstraction APIs
- Betriebssysteme die auf den VMs betrieben werden können
- Anfallende Kosten

Die Anfallenden Kosten werden jeweils für einen VM-Typ (beschreibt Leistungsprofil von VMs) angegeben, dessen Leistungsmerkmale in etwa den Anforderungen für den Betrieb von RCE und die durchzuführenden Tests entsprechen. Die

VM-Typen befinden sich dadurch auch in einer vergleichbaren Leistungs- und Preiskategorie.
Zunächst wurden Public Cloud Systeme ausgewählt, die vom Großteil der bereits recherchierten Cloud Abstraction APIs (Evaluierung und Beschreibung in Kapitel 3.7) unterstützt werden, um die Flexibilität bei der Auswahl einer Abstraction API zu gewährleisten. Des Weiteren kamen solche Provider in Frage, deren Cloud Lösungen sich bereits über mehrere Jahre am Markt behauptet haben. Diesen Kriterien entsprachen die Cloud Systeme Amazon EC2, Rackspace Cloud Servers, IBM Smartcloud und CloudSigma, die dadurch für eine weitere Untersuchung in Frage kamen.

3.3.1 Amazon EC2

Die Elastic Compute Cloud von Amazon stellt einen Webservice dar, der skalierbare Rechenkapazitäten, in Form von konfigurierbaren VMs bereitstellt. Im Kontext von Amazon EC2 wird eine VM als Instanz bezeichnet. Die Konfiguration der Instanzen erfolgt über Amazon Machine Images (AMI), welche Templates definieren, die eine Softwarekonfiguration mit Betriebssystem und Anwendungen beinhalten. Mit Hilfe der AMI werden Instanzen erstellt, die eine Kopie der AMI ausführen. Die für die Instanz reservierten virtuellen Ressourcen werden in Instanz-Typen kategorisiert, welche sich in Arbeitsspeicher, Rechenleistung, Festplattenspeicher und Prozessorarchitektur unterscheiden.[4]
Um VM Images aus einer vorhandenen Umgebung auf EC2 Instanzen zu übertragen, unterstützt Amazon derzeit den Import von VMware ESX VMDK- und Citrix Xen VHD-Images. Weiterhin ist es möglich Microsoft Hyper-V VHD-Images mit den Betriebssystemen Windows Server in der Version 2003 (R2) oder 2008 (R2 oder R3) zu importieren. Zur Umsetzung können die EC2-API-Tools oder der VMware vSphere EC2 VM Import Connector verwendet werden. Für die Durchführung von VM Imports fallen keine weiteren Kosten an. Zudem sind im System bereits eine Vielzahl von VM-Images mit Unix Betriebssystemen sowie Windows Server 2003 und 2008, in 32 sowie 64 Bit, vorhanden, die für das Erstellen von Instanzen verwendet werden können. Als Hypervisor setzt Amazon AWS Xen ein. Eine Überwachung, der für die gestarteten Instanzen reservierten Ressourcen, ist über Funktionen der API möglich. Die Erstellung von Subnets ist kostenfrei über das Produkt Virtual Private Cloud (VPC) realisierbar, bei dem bis zu 20 Subnets in der Größenordnung /18 bis /28 konfiguriert werden können.
Die VPC ermöglicht die Bereitstellung einer Virtuellen Netzwerktopologie, innerhalb der Amazon Web Services (AWS) Cloud, bei der Subnets und Routing-Tabellen individuell erstellt werden können. Weiterhin ist die VPC mit einem gewählten Sicherheitsniveau zu konfigurieren, das Abstufungen von einem einfachen Internet Gateway als VPC Zugang, bis hin zu einem IPsec-Tunnel zwischen VPC und der eigenen Netzwerkinfrastruktur bietet. Die Verbindung der einzelnen Subnets ist über eine Stern-Topologie gelöst.[3]

Die Kosten für eine Standard On-Demand Instanz mit einem Single Core Prozessor, der einem 1,7 GHz Xeon-Prozessor aus dem Jahr 2006 entspricht, 1,7 GB Arbeitsspeicher, 160 GB Festplattenspeicher und einem Unix Betriebssystem betragen 7,60 Cent pro Stunde. Soll ein Windows-Betriebssystem zum Einsatz kommen, ist ein Aufpreis von ca. 44% zu zahlen. Diese Differenz ist bei Instanzen aller Leistungsstufen zu begleichen. Bei Verwendung der On-Demand Instanzen entstehen keine Fixkosten und langfristige Vertragsbindungen. Auf den Instanzen eingehenden Datenübertragungen sind kostenfrei, wenn diese die Grenze von 10 TB nicht überschreiten. Für ausgehende Transfers fallen Kosten in Höhe von 9,6 Cent pro GB an.
Amazon EC2 bietet derzeit noch keine IPv6-Unterstützung für die Netzwerkanbindung von Instanzen, gibt aber an, dass dies für die Zukunft geplant sei. Weiterhin wird die EC2-API von den Cloud Abstraction APIs Libcloud, Deltacloud und Jclouds unterstützt.
Amazon ist der einzige Cloud Provider, bei dem alle relevanten Informationen, über die Webseite beschafft werden konnten. Die EC2-Produktbeschreibungen sind ausführlich und gut Strukturiert. Weiterhin sind detaillierte Dokumentationen zur Verwendung der angebotenen Dienste, APIs und Management-Tools erhältlich. Die API besitzt einen vergleichbar großen Funktionsumfang und könnte alle nötigen Anforderungen Bedienen. Des Weiteren stellt die EC2-API einen Quasi-Standard für Cloud APIs dar und wird auch z.T. von Private Cloud Systemen (u.A. OpenStack und Nimbus) als „Zweit-API" angeboten. Hierbei ist allerdings zu beachten, dass mit einem EC2-Client nicht verschiedene Cloud Systeme mit EC2-API gesteuert werden können, da diese sich meist in den verwendeten Authentifizierungsverfahren unterscheiden. Den einzigen Mangel, den Amazon EC2 für den vorgesehenen Einsatzzweck aufzeigt, ist die fehlende Unterstützung für IPv6.

3.3.2 Rackspace Cloud Servers

Rackspace Cloud Servers stellt die Public Compute Cloud Lösung von Rackspace dar, mit der je nach Ressourcenbedarf konfigurierbare VMs bereitgestellt werden können. Im Kontext von Rackspace Cloud Servers wird eine VM als Server bezeichnet. Zur Erstellung eines Servers ist ein Image zu bestimmen, dass eine Softwarekonfiguration mit Betriebssystem und Anwendungen beinhaltet. Es werden hier verschieden Images bereits im System zur Auswahl angeboten, wobei auch die Erstellung eigener Images, über bereits betriebene und selbst konfigurierte Cloud Server möglich ist. Die für einen Server reservierten virtuellen Ressourcen werden in Flavor kategorisiert, welche sich in Arbeitsspeicher, Priorität für CPU-Zeit und Festplattenspeicher unterscheiden.[5]
Rackspace Cloud Servers bietet keine Möglichkeit VM Images aus einer vorhandenen Umgebung zu importieren. Als Hypervisor wird auch hier Xen eingesetzt. Eine Überwachung, der Ifür die gestarteten Instanzen reservierten Ressourcen, ist über Funktionen der API möglich. Die Erstellung von Subnets wird nicht un-

terstützt. Die Auswahl an Betriebssystemen umfasst verschiedene Linux Distributionen sowie Windows Server 2008 in 32 und 64 Bit. Auch Rackspace Cloud Servers bietet derzeit noch keine IPv6-Unterstützung für die Netzwerkanbindung von Servern, gab aber die Auskunft, dass dies, falls ihr IPv4-Adressbereich ausgeschöpft ist, ermöglicht werden würde und die Infrastruktur bereits darauf vorbereitet ist. Weiterhin wird auch die Rackspace Cloud Servers API von den Cloud Abstraction APIs Libcloud, Deltacloud und Jclouds unterstützt.
Die Kosten für einen Linux Server mit 2GB Arbeitsspeicher, 80GB Festplattenspeicher und 12,5% der CPU-Ressourcen von 4 Single Core vCPUs mit jeweils 2,5GHz betragen 9,6 Cent pro Stunde. Rackspace unterscheidet hier bei der Menge der zugeordneten CPU-Ressourcen zwischen Windows und Linux Servern. Ein Windows Server mit der gleichen Hardwarekonfiguration erhält 2 vCPUs, die zu 100% genutzt werden können. Auch Rackspace erhebt keine Gebühren für eingehende Datenübertragungen auf den Servern, wenn diese die Grenze von 10 TB nicht überschreiten. Für ausgehende Transfers fallen Kosten in Höhe von 14,4 Cent pro GB an.
Die Webseite von Rackspace konnte etwa die Hälfte der relevanten Informationen, ausreichend strukturiert, liefern. Die Punkte außerhalb der Basisfunktionalitäten wurden in einer persönlichen Anfrage zufriedenstellend geklärt. Im Rahmen der unterstützten Funktionalitäten kann die gut dokumentierten API die entsprechenden Anforderungen bedienen. Einen Nachteil stellen die Fehlende Import-Möglichkeit für VM-Images, sowie die nicht unterstützte Subnet-Konfiguration dar.

3.3.3 IBM Smartcloud Enterprise

Das IaaS Produkt von IBM beschreibt, wie auch Amazon EC2, VMs als Instanzen. Die bereitgestellten Images beinhalten Softwarekonfigurationen mit Betriebssystemen und Anwendungen, die einer zu erstellenden Instanz zugewiesen werden können. Bei den zur Verfügung stehenden Anwendungen handelt es sich bis auf wenige Ausnahmen, um Produkte aus dem IBM Software-Portfolio. Zur Reservierung von virtuellen Ressourcen für eine Instanz, kann aus 4 Leistungsprofilen gewählt werden, die sich in Arbeitsspeicher, Anzahl an CPUs (mit je 1,25 GHz) sowie Festplattenspeicher unterscheiden.
Ist der Nutzer bereit, eine monatliche Gebühr von 58 EUR für IBM Premium Support zu zahlen, können VM-Images aus einer vorhandenen Umgebung, im Open Virtualization Format (OVF), durch diesen importiert werden. Als Hypervisor wird derzeit KVM eingesetzt, welcher in Kürze durch ein geeignetes Produkt von VMWare ersetzt werden soll. Eine Überwachung der Instanzen über die zur Verfügung gestellte API ist nicht möglich. Die Realisierung von Subnets kann über die Segmentierung in VLANs erfolgen, welche mit Hilfe des Supports konfiguriert wird (Auch hierfür ist eine Buchung des Premium Supports notwendig). Weiterhin ergeben sich Einrichtungsgebühren von 78 EUR pro VLAN, von denen max. fünf zur Verfügung gestellt werden können. Die Auswahl an verfügbaren VM-

Images umfasst verschiedene Linux Distributionen sowie Windows Server 2003 und 2008, in 32 und 64 Bit. Aufgrund noch ungeklärter Lizenzierungsthemen, werden auch hier keine Windows XP bzw. Windows 7 Images angeboten und der Import dieser unterbunden. Auch IBM verzichtet derzeit noch auf eine Netzwerkanbindung der Instanzen über IPv6, da die Nachfrage diesbezüglich bisher nur gering war. Die IBM SmartCloud API wird von den Cloud Abstraction IPs Deltacloud, Jclouds und Libcloud Unterstützt. Libcloud stellt hier allerdings keine Funktionalitäten für das Software-Deployment über SSH zur Verfügung.
Der Betrieb einer Single Core Instanz mit 2 GB Arbeitsspeicher und 60 GB Festplattenspeicher beläuft sich auf 7,3 Cent pro Stunde. IBM berechnet für eingehende sowie ausgehende Datenübertragungen der Instanzen 9,6 Cent pro GB, innerhalb eines Volumens von 10 TB.
Wie der Zusatz „Enterprise" in der Bezeichnung des Cloud-Produkts von IBM vermuten lässt, ist diese auf Geschäftskunden ausgerichtet und schließt Privatkunden von der Nutzung aus. Es wird zur Gewährleistung eines höheren Sicherheitsniveaus ein persönlicher Kontakt zu den Kunden gepflegt und eine grobe Übersicht der jeweiligen Anwendungsszenarien gewünscht. Die Informationen über die Smartcloud, welche durch die Website zur Verfügung gestellt werden, fallen unübersichtlich aus und sind sehr allgemein gehalten. Unterstützte Funktionen des IaaS-Systems werden nur mit wenig Details erläutert. Dafür setzt IBM auf Kunden- und Anwendungsorientierte Betreuung und nimmt sich bereits bei ersten Gesprächen über das Cloud-Produkt, mit kompetenten Mitarbeitern, viel Zeit, um auf die Bedürfnisse des Interessenten einzugehen. Auch der Cloud-Nutzer wird bei entsprechender Support-Buchung individuell, im Projekt betreut und es können Spezial-Lösungen implementiert werden, die bei einem Großteil der Self-Service-Orientierten Cloud Providern ausgeschlossen sind. Für die verteilten Tests ist dies allerdings, in den meisten Fällen nicht von Nutzen, da jegliche Operationen zur Steuerung der Cloud, automatisiert über die API, abgewickelt werden sollten. Die IBM Smartcloud bietet weiterhin keine Möglichkeit die API bzw. deren Funktionsumfang einzusehen, wodurch diesbezüglich keine Bewertung vorgenommen werden kann. Die Einschränkungen bei der Konfiguration von Subnetzen, welche manuell durch den Support vorgenommen werden müssen, sind für die automatisierten, verteilten Tests von Nachteil. Diese sollten kurzfristig, zur Änderung der Testumgebung, mit Hilfe der API eingesteuert werden können. Weiterhin ist hier die Beschränkung auf 5 VLANs und die zusätzlich dafür anfallenden Kosten ein Kritikpunkt.

3.3.4 CloudSigma

CloudSigma stellt im Gegensatz zu den Konkurrenten, einen reinen Cloud-Provider dar, der zudem ausschließlich IaaS anbietet. Wie bei den anderen Providern, ist es auch hier möglich, die Rechenressourcen der zu erstellenden VM über CPU, Arbeitsspeicher und Festplattenspeicher zu definieren. Im Gegensatz zur Konkurrenz bietet CloudSigma die Möglichkeit, neben einer Auswahl der An-

zahl der CPU-Kerne, die Taktfrequenz zu bestimmen. Weiterhin lässt sich auch das Volumen des Festplattenspeichers flexibel wählen, ohne auf vorgegebene Werte beschränkt zu sein. Neben vorkonfigurierten VM-Images mit verschiedenen Windows- und Linux-Betriebssystemen, bietet sich dem Nutzer die Möglichkeit mit einer eigenen Installations-CD bzw. –Image, ein Betriebssystem auf einer VM zu installieren.
VMWare Images können nach einer Konvertierung in ein VM RAW-Format, in das CloudSigma IaaS-System importiert und auf dem dort eingesetzten KVM-Hypervisor in einer VM betrieben werden. Eine Ressourcenüberwachung der VMs kann über die zur Verfügung gestellte API erfolgen. Die Erstellung von Subnets ist mit der Möglichkeit zur Segmentierung in VLANs gegeben, wobei maximal 4096 VLANs erstellt werden können und je VLAN, 7 EUR Nutzungsgebühr pro Monat anfallen. Die Auswahl an verfügbaren VM-Images umfasst verschiedene Linux Distributionen sowie Windows Server 2003 und 2008, in 32 und 64 Bit. Windows XP bzw. Windows 7 könnten über einen Import von VM-Images oder eine entsprechende Betriebssysteminstallation eingebracht werden. Auch CloudSigma bietet derzeit noch keine Unterstützung der IPv6-Adressierung, gab nach einer Anfrage allerdings an, dass eine Einführung für Ende 2012 geplant sei. Für die Nutzung einer VM mit einer Single Core CPU mit 1,75 GHz, 2 GB Arbeitsspeicher und 20 GB Festplattenspeicher fallen pro Stunde 7,8 Cent an. Im Gegensatz zu den anderen Providern erfolgt hier allerdings die Abrechnung im 5-Minuten Takt und nicht stündlich. Hinzu kommen 3,50 EUR je statischer IP im Monat sowie eine Gebühr für ausgehenden Datentransfer in Höhe von 4,55 Cent pro GB. Die CloudSigma API wird von den Cloud Abstraction IPs Deltacloud, Jclouds und Libcloud Unterstützt. Libcloud stellt auch hier keine Funktionalitäten für das Software-Deployment über SSH zur Verfügung.
CloudSigma bietet eine hohe Flexibilität bei der Konfiguration der Rechenressourcen einer VM, sowie eine uneingeschränkte Nutzung eigener VM-Images und Installations-Medien. Dadurch besitzt der Kunde auch mehr Einfluss auf die Preisgestaltung. Die Struktur der Cloud sowie verwendete Technologien, bis hin zur eingesetzten IT-Infrastruktur in den Rechenzentren, werden auf der Webseite dargestellt. Zur Recherche konnte so ein Großteil der Informationen auf diesem Wege bezogen werden. Die ausführlich dokumentierte API bietet alle nötigen Funktionalitäten, um den definierten Anforderungen gerecht zu werden. Als einziger Kritikpunkt kann hier, wie auch bei Amazon, nur die fehlende IPv6-Unterstützung angebracht werden.

3.4 Bewertung der Public Compute Cloud Systeme

Die evaluierten Cloud Systeme sind alle grundsätzlich geeignet, für die Realisierung der verteilten Tests, eingesetzt zu werden. Das System von Rackspace wies im Vergleich die meisten Mängel, mit den fehlenden Funktionalitäten für Image-Import, Subnetz-Konfiguration und IPv6-Unterstützung, auf. Die für den Betrieb einer VM anfallenden Kosten unterscheiden sich bei den untersuchten Providern

nur geringfügig voneinander. Rackspace fiel hier mit den höchsten Kosten für den VM-Betrieb je Stunde auf. Werden die Gesamtkosten betrachtet, würde Amazon den günstigsten Provider für die geplanten Anwendungsfälle darstellen, da hier lediglich für den VM-Betrieb und ausgehende Datenübertragungen Kosten entstehen. Die ausgehenden Datenübertragungen besitzen allerdings keine große Relevanz, da diese nur mit geringen Datenmengen, wie beispielsweise Log-Dateien, belastet werden würden. Zusatzkosten wie die Gebühr für zu nutzende VLANs bei IBM und CloudSigma, sowie der für einige Operationen notwendige, kostenpflichtige Support von IBM, müssen berücksichtigt werden, da diese relevante Funktionalitäten zur Umsetzung der verteilten Tests bereitstellen. Die IBM Smartcloud ist aufgrund der manuellen VLAN-Konfiguration durch den Support weniger für automatisierte Tests geeignet, da es hier dementsprechend keine Möglichkeit gibt, dies über die API abzuwickeln. Weiterhin sind auch die Kosten mit 78 EUR je VLAN vergleichsweise hoch. CloudSigma und Amazon können bis auf die IPv6-Adressierung alle relevanten Funktionalitäten unterstützen. Amazon besitzt allerdings noch einen Preisvorteil, da hier keine zusätzlichen Kosten für die Subnet-Konfiguration anfallen würden. Weiterhin bietet die Subnet-Konfiguration bei Amazon in einer Virtual Privat Cloud mehr Einstellungsmöglichkeiten für Routingfunktionalitäten. Die EC2-API ist im Bereich IaaS-Cloud eine weit verbreitete Schnittstelle zur Steuerung der zur Verfügung stehenden Ressourcen. So wäre die Nutzung von Amazon EC2 auch für einen möglichen Providerwechsel, bzw. den parallelen Betrieb einer Private Cloud sinnvoll, da hier oftmals auch eine EC2-API bereitgestellt wird. Des Weiteren unterstützen die Cloud Abstraction APIs nahezu alle Funktionalitäten der EC2-API. Bei anderen Cloud Systemen ist dies häufig nicht der Fall. Dies ist sicherlich darauf zurückzuführen, das die EC2-API eine zentrale Rolle bei den Cloud APIs einnimmt und häufig als Vorlage, bzw. Maßstab in diesem Bereich herangezogen wird. Zur Bereitstellung einer Public Cloud Infrastruktur für die verteilten Tests wäre daher Amazon zu bevorzugen.

3.5 Evaluierung von Cloud Management Systemen

Um Tests auch innerhalb des DLR-Netzes, ohne Beanspruchung der Dienste eines Cloud Providers, durchführen zu können, wurden neben den Public Compute Cloud Systemen auch die Möglichkeit zum Aufbau einer Private Cloud untersucht. Für diesen Ansatz sprechen das Vorhandensein eines, durch die Abteilung betriebenen VMware ESX Hypervisors, sowie die Möglichkeit, vertrauliche Daten innerhalb eines Tests, nicht in ein Fremdnetz übergehen zu lassen. Der vorhandene Hypervisor könnte so später mit in die Private Cloud eingebunden werden, um die vorhandenen Ressourcen besser zu nutzen. Zur Auswahl der zentralen Steuerungskomponente innerhalb der aufzubauenden IaaS Cloud, wurden die drei größten und weitverbreitetsten Open Source IaaS Cloud Management Systeme (CMS) Eucalyptus, OpenStack und OpenNebula näher untersucht. Eucalyp-

tus ist als Open Source sowie einer nicht freien Enterprise Version erhältlich. Da nur die Enterprise Version VMware Hypervisor sowie Windows VMs unterstützt, wurde dieses Produkt nicht weiter untersucht. Mit der Open Source Version könnte so der vorhandene ESX Hypervisor nicht genutzt und keine Tests auf Windows-Platformen durchgeführt werden.
Nach der Konfiguration, der zum Betrieb eines CMS vorausgesetzten Infrastruktur, erfolgte die Evaluierung der Systeme OpenStack und OpenNebula.
Auswahlkriterien stellen, wie bei der Evaluierung der Public Cloud Systeme, die in Kapitel 3.1 beschriebenen Anforderungen, die sich an das zu verwendende Cloud System richten, dar. Da auch hier die untersuchten Systeme Funktionalitäten zum Erstellen, Starten und Stoppen einer VM bieten, wird nachfolgend nur noch auf besondere Eigenschaften dieser Funktionen eingegangen. Im Vordergrund der Evaluierung stehen hier ebenfalls die erweiterten Funktionalitäten, die nicht allen Public Cloud Providern zu finden waren. Diese umfassen:

- Import von vorinstallierten Betriebssystem-Images
- Ressourcenüberwachung der in Betrieb befindlichen VMs (für z.B. CPU-, Speicherauslastung)
- Unterstützung von IPv6
- Konfigurationsmöglichkeit für Subnets
- Unterstützung durch Cloud Abstraction APIs
- Betriebssysteme die auf den VMs betrieben werden können

3.5.1 Vorbereiten der Testumgebung

Zum Betrieb eines IaaS CMS ist mindestens ein Hypervisor als Virtualisierungsplatform für die VMs sowie ein System für das CMS, zur Verfügung zu stellen.

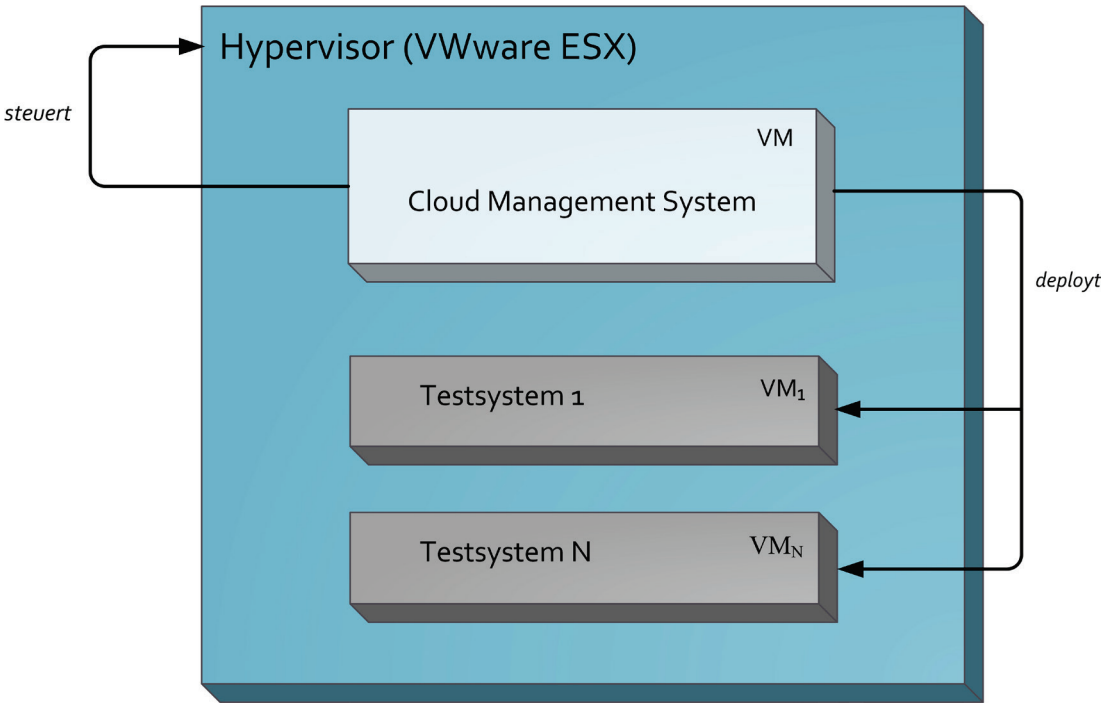

Abbildung 1: Hypervisor und CMS

Um im ersten Testbetrieb, den produktiv in der Abteilung betriebenen VSphere Hypervisor nicht zu involvieren, wurde für diese Aufgaben zunächst ein gesonderter Server angeschafft. Zur Einsparung von Hardwareressourcen wurde als Platform für das CMS eine VM innerhalb des Hypervisors vorgesehen, welcher dann durch dieses gesteuert werden soll. Wie in Abbildung 1 dargestellt, werden zudem die durch das CMS auf dem Hypervisor deployten VMs, welche damit die Testsysteme für die automatisierten, verteilten Tests bereitstellen, betrieben.
Als Hypervisor wurde hier, der zum Zeitpunkt der Evaluierungen frei erhältliche VMWare ESXi 5 eingesetzt. Diese Version entspricht auch der, in Zukunft im Abteilungsnetz betriebenen Version, mit dem Unterschied, dass dort die kostenpflichtige Variante, mit einem größeren Funktionsumfang eingesetzt wird.
Zur Spezifikation der erforderlichen Hardware wurde zunächst davon ausgegangen, für die verteilten Tests innerhalb der Private Cloud max. 8 VMs zu betreiben. Pro Test-VM würden dabei etwa 2GB Arbeitsspeicher und 10GB Festplattenspeicher benötigt. Weiterhin würde hier die VM des CMS betrieben, welcher ebenfalls ca. 2GB Arbeitsspeicher und 6GB Festplattenspeicher zugewiesen werden sollten. Hinzu kommt ein Speicheroverhead von ca. 200MB je 64Bit VM mit 2GB Arbeitsspeicher, sowie 1GB Arbeitsspeicher für den ESXi selbst. Durch das Speichermanagement des ESXi werden diese Overheads allerdings wieder kom-

pensiert, indem den betriebenen VMs das tatsächlich benötigte Speichervolumen, während des Betriebs, live im Hypervisor reserviert und wieder freigegeben wird. Weiterhin werden über das ESX Transparent Page Sharing identische Inhalte, die sich im Speicher mehrerer VMs befinden, nur einmal im physischen Speicher vorgehalten. Dies ist beispielsweise in Bereichen des Betriebssystem-Kernels der Fall. Neben dem Festplattenspeicher für die VMs werden noch ca. 100GB zur Vorhaltung der VM-Images benötigt. Anhand dieser Parameter wurde als Hardware ein Dell PowerEdge T110 II, mit einem 3,2 GHz Intel Xeon Quad-Core Prozessor, 16GB Arbeitsspeicher und 2 TB Festplattenspeicher (um etwas Reserve zu gewährleisten) ausgewählt.
Vor der anschließenden Installation des ESXi 5 Hypervisors war zunächst im Bios das CPU Feature „Intel VT" (Virtualization Technology) zu aktivieren, da sonst die Hardwarevoraussetzungen nicht erfüllt werden und die Installation abbricht. Der Setup-Dialog ist ansonsten klar strukturiert und innerhalb ca. 15 min. abgeschlossen. Die Netzwerkkarte des Systems wurde an das DLR-Netz angebunden und erhält seine IP via DHCP.
Die anschließende Konfiguration erfolgt über den VSphere Client, der kostenfrei von VMware erhältlich ist und die GUI für den Nutzer bereitstellt. Dieser ist auf einem Client System, mit Netzwerkzugriff auf den ESXi Server, zu installieren und anschließend mit dem ESXi über den root-Account zu verbinden. Zur besseren Konfiguration und Wartung wurden noch über die lokale Service-Konsole auf dem ESXi, der SSH-Zugang und die lokale Shell aktiviert.

3.5.2 Integration in das DLR-Netz

Der ESXi Hypervisor erhielt nach der Installation einen direkten Zugang zum DLR-Netz. Da nach der im DLR geltenden Security Policy, generell nur vom IT-Dienstleister geprüfte Systeme an das interne Netzwerk angebunden werden dürfen, stellt der Hypervisor hier eine Ausnahme dar. Um zur Einhaltung der Security Policy, das CMS und die durch dieses deployten VMs, nicht direkt im DLR-Netz zu betreiben, musste zunächst ein entsprechendes Testnetz implementiert werden, das diese Systeme verbirgt.

3.5.2.1 Netzdesign

Das in Abbildung 2 dargestellte Netzdesign ermöglicht die sichere Anbindung des Testnetzes an das DLR-Netz über die, am Netzwerkübergang positionierte Firewall. Diese verbirgt CMS sowie Test-VMs und maskiert die notwendigen Netzwerkzugriffe in das DLR-Netz sowie Internet über eine Network Address Translation (NAT) auf die statische IP 129.247.111.141.

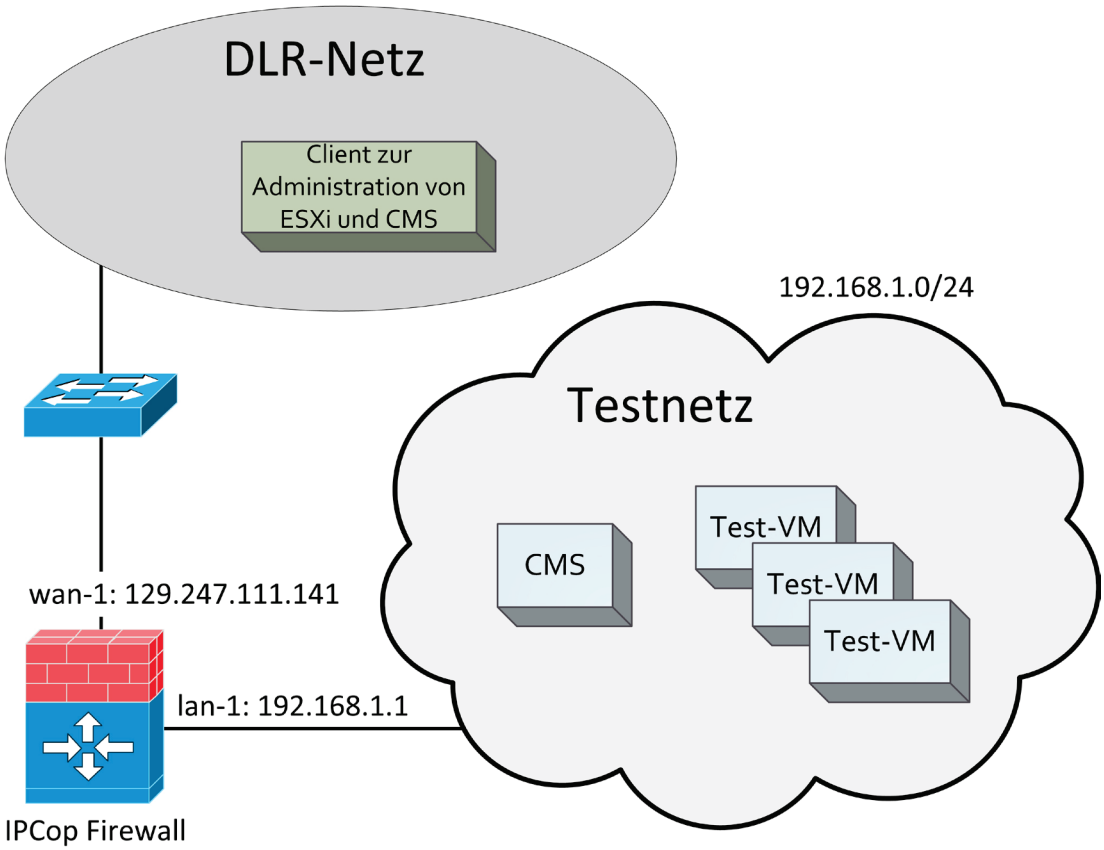

Abbildung 2: Netzdesign zur Anbindung des Testnetzes

Da über den ESXi keine Routing- und Firewall-Funktionalitäten für die virtuellen Netze zur Verfügung gestellt werden, war ein zusätzliches Firewall-System aufzusetzen. Dies erfolgte innerhalb einer VM, um so das virtuelle Testnetz und das, über die physische Netzwerkschnittstelle des ESXi angebundene DLR-Netz zu verknüpfen. Für einen ähnlichen Fall musste in der Abteilung bereits ein anderes Versuchsnetz an das DLR-Netz über eine Firewall angebunden werden. In diesem Zusammenhang wurden bereits erste Erfahrungen gesammelt und die IPCop Firewall von der IT-Sicherheit als akzeptable Lösung eingestuft. Diese Richtlinie galt es auch für diesen Netzwerkübergang einzuhalten und das System wurde auf der VM entsprechend Konfiguriert.

3.5.2.2 ESXi-Netz

Über die physische Netzwerkschnittstelle des Host-Systems wird der ESXi an das DLR-Netz angebunden. Die Firewall besitzt zur Verbindung der beiden Netze entsprechend zwei virtuelle Netzwerkschnittstellen. Die verschiedenen Netze werden innerhalb des ESXi, auf zwei virtuelle Switches verteilt, von denen einer an die physische Netzwerkschnittstelle und der andere nur an das virtuelle Testnetz angebunden ist. Über die physische Netzwerkschnittstelle kommunizieren dann der ESXi und der IPCop auf Layer 2, mit dem direkt angebundenen DLR-Switch sowie dem nächsten Router, über die MAC-Adresse der jeweiligen virtuellen Schnittstelle. Der ESXi selbst ist auf Layer 3, über vmk0 mit der per DHCP bezogenen IP 129.24.111.220 mit dem DLR-Netz verbunden. Abbildung 3 zeigt die Netzwerkstruktur auf dem ESXi, in der VMs, virtuellen Switches über Portgruppen zugeordnet werden.

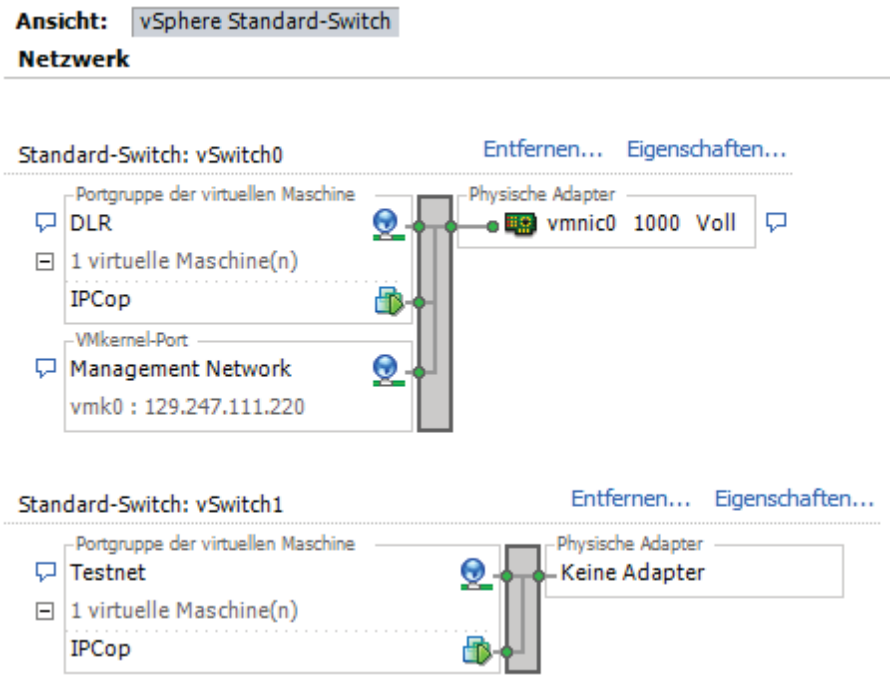

Abbildung 3: ESXi Netzwerkstruktur

Portgruppen definieren für die angebundenen VMs erweiterte Netzwerkkonfigurationen wie VLAN-Einstellungen, Loadbalancing und Layer 2 Sicherheitsmaßnahmen. Weiterhin veranschaulichen sie eine logische Trennung zwischen Gruppen von VMs. Es bestehen allerdings weiterhin volle Zugriffsmöglichkeiten zwischen VMs verschiedener Portgruppen auf einem Switch, solange diese keine VLANs konfiguriert haben.

3.5.2.3 Firewall-Konfiguration

Der IPCop ist eine gehärtete, Open Source Linux Distribution, die speziell für den Einsatz als Firewall zugeschnitten ist. Es laufen hier nur die, für die Funktion als Router und Firewall relevanten Dienste, wie DNS- und DHCP-Server sowie iptables. Für das IPCop-System wurde eine VM mit 512 MB Arbeitsspeicher, 2 GB Festplattenspeicher sowie 2 Netzwerkschnittstellen angelegt. Die Installation erfolgte über den VSphere Client, mit Hilfe des CD-Image-Files der IPCop-Version 2.0.3. Ist die Grundkonfiguration via Konsole abgeschlossen, erfolgen die weiteren Einstellungen über ein Webinterface. Die im DLR-Netz befindliche Schnittstelle, wan-1, erhielt die statische IP 129.247.111.141 und die im Testnetz positionierte Schnittstelle, lan-1, die IP 192.168.1.1 (siehe Kapitel 3.5.2.1, Abbildung 2). Bei der Konfiguration über die GUI, werden dann zur deutlichen Abgrenzung von internem und externem Netz, wan-1 als „red" und lan-1 als „green" bezeichnet.
Die Default Policy der Firewall sieht vor, alle an wan-1 eingehenden Verbindungen an den IPCop selbst sowie lan-1 zu verwerfen und alle von lan-1 ausgehenden Verbindungen zu erlauben. Zur Einschränkung der Kommunikationsmöglichkeit der Test-VMs in das DLR-Netz, wurde die Default Policy für ausgehende Verbindungen an lan-1 so geändert, dass auch hier alle Verbindungsanfragen verworfen werden.
Des Weiteren maskiert der IPCop die erlaubten Netzwerkzugriffe in das DLR-Netz sowie Internet mit der statischen IP von wan-1. Zur Bereitstellung einer Zugriffsmöglichkeit auf die Dienste, der im Testnetz befindlichen Systeme, wurden später noch entsprechende Portforwards eingerichtet.

3.5.3 OpenNebula

3.5.3.1 Architektur

Das Open Source Private Cloud System OpenNebula, setzt sich aus einem Front-end, Image Repository und einer Netzwerk-Komponente zusammen. Das Front-end beinhaltet die zentralen OpenNebula Services, welche den Management Daemon (oned), Scheduler, das Monitoring und Accounting sowie ein Webinterface umfassen. Weiterhin werden hier die Cloud APIs EC2 sowie OCCI zur Verfügung gestellt. Die einzelnen Komponenten kommunizieren über XML-RPC. Das Front-end setzt das Vorhandensein mindestens eines Hypervisors voraus, auf dem die, über das Front-end übermittelten VM-Images, betrieben werden. Zur Steuerung des Hypervisors über SSH, werden auf dem Front-end entsprechenden Treiberbibliotheken für Xen, KVM sowie VMware Hypervisor bereitgestellt. Auf dem Hypervisor ist dafür ein Administrator-Account mit SSH-Zugang bereit zu stellen. Der für den OpenNebula-Zugriff konfigurierte Hypervisor wird hier auch als Cluster Node bezeichnet, da u.A. zur gleichmäßigen Lastverteilung, VMs, die auf einem voll ausgelasteten Hypervisor laufen, auf einen, mit entsprechend geringerer Ressourcenauslastung, mitgriert werden können.[11]

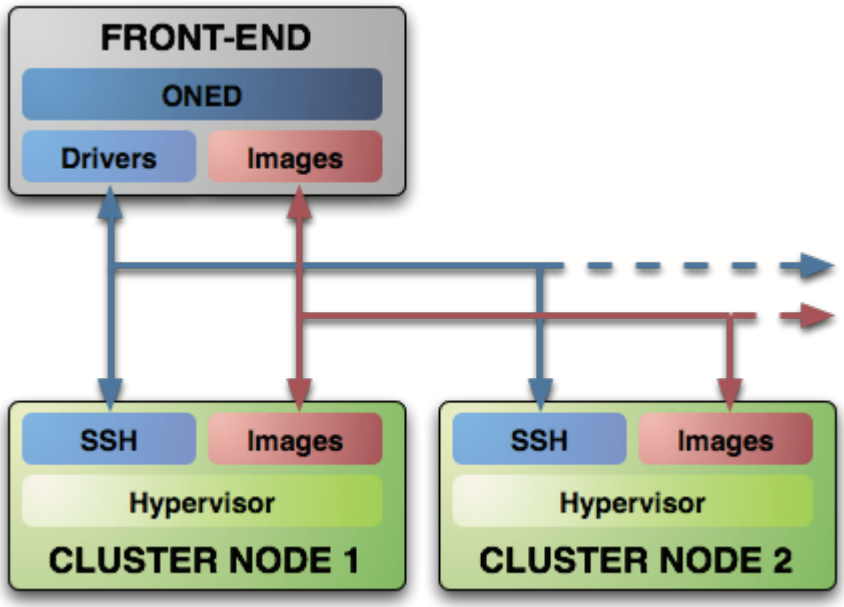

Abbildung 4: Beziehung von OpenNebula Front-end und Hypervisor, nach [10]

Dem Front-end, sowie dem Hypervisor ist via NAS, SAN oder lokalem Dateisystem ein Image Repository bereitzustellen, das zum Transfer der VM-Images verwendet wird.

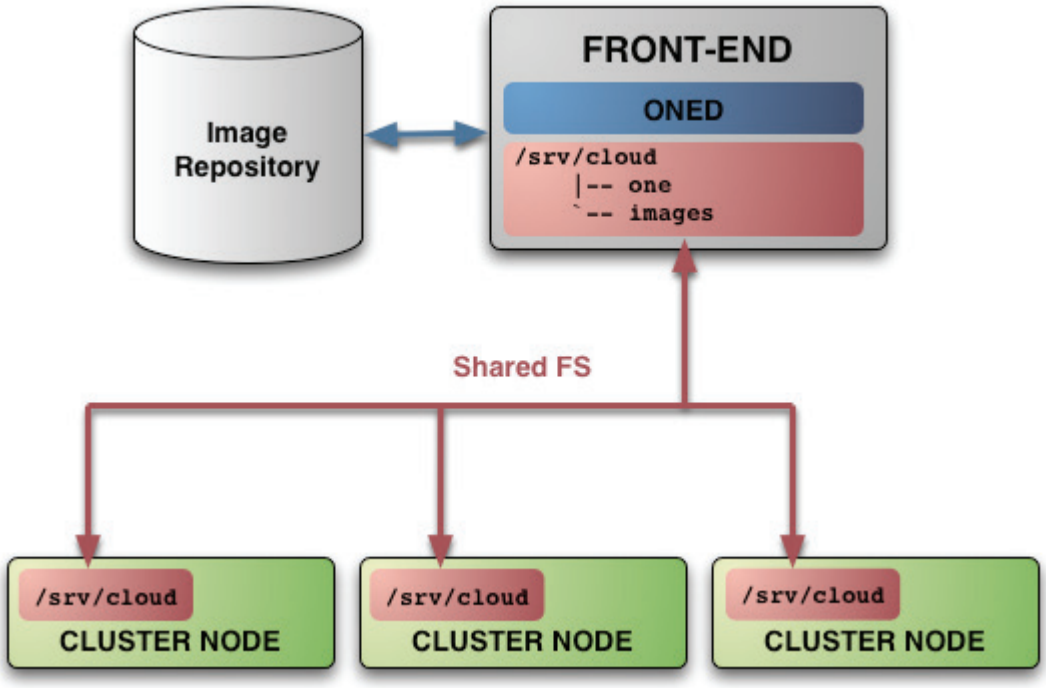

Abbildung 5: Image-Transfer Front-end - Hypervisor (Cluster Node), nach [10]

Beim Deployment einer VM wird das entsprechende Image aus dem Image Repository an den Hypervisor übermittelt. Je nach verwendeter Speichertechnologie kann dies in Form einer Kopie, eines symbolischen Links, oder eines iSCSI Target erfolgen. Es wird die Verwendung eines verteilten Dateisystems empfohlen, um so auch Live Migrationen zu ermöglichen.
Zur Gewährleistung der Netzwerkkonnektivität von VMs über verschiedene Hypervisor hinweg, wird die Netzwerkschnittstelle jeder VM, mit der konfigurierten Network Bridge auf dem Hypervisor, bei VMware mit der entsprechenden Portgruppe, verbunden. Die Bezeichnungen für Portgruppen bzw. Bridges sind dafür auf allen Hypervisor sowie in der OpenNebula-Konfiguration einheitlich zu wählen.[11]

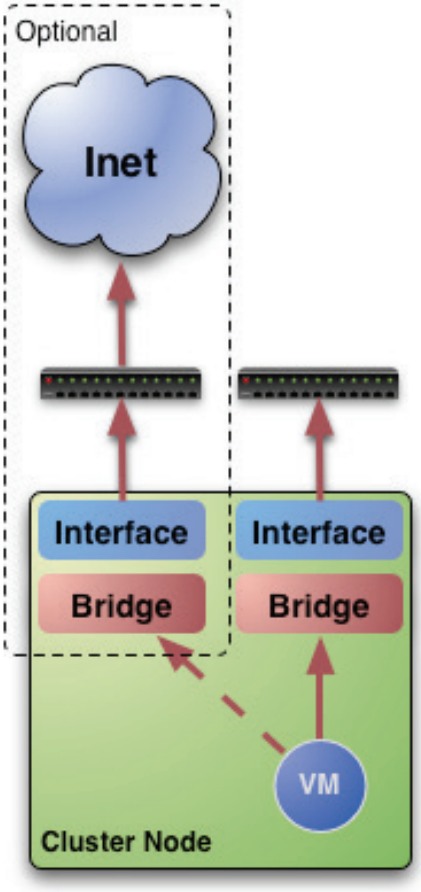

Abbildung 6: OpenNebula Networking, nach [10]

3.5.3.2 Unterstütze Funktionalitäten

Bezüglich der für die geplanten Anwendungsfälle relevanten Funktionen, ermöglicht OpenNebula zunächst den Import von VM-Images der Hypervisor Xen, KVM und VMware ESX(i). Die Ressourcenüberwachung der VMs lässt sich über Statistics-Funktionalitäten der APIs abwickeln. Wie auch bei den Public Cloud Systemen wird auch hier keine IPv6-Addressierung der VMs unterstützt. OpenNebula selbst besitzt keine Komponente, die Netzwerkfunktionalitäten für die VMs bereitstellt. Es wird nur die Netzwerkkonfiguration des Hypervisors verwaltet und die Erstellung von Subnets kann dort ausschließlich über die Segmentierung in VLANs durchgeführt werden. Die Unterstützung der OpenNebula API ist durch die Cloud Abstraction APIs Libcloud, Deltacloud und Jclouds gegeben. Als Betriebssystem können alle mit dem eingesetzten ESXi 5 Hypervisor kompatiblen Architekturen eingesetzt werden.

3.5.3.3 Installation und Konfiguration

Die Installation von OpenNebula erfolgte in der Version 3.4 auf einer Ubuntu 10.04 Desktop x64 VM des ESXi mit 2GB Arbeitsspeicher, 100GB Festplattenspeicher und einer Single Core CPU. Nach der Installation sind zunächst Umgebungsvariablen zu registrieren, die den OpenNebula Benutzer oneadmin, zur Ausführung von Steuerungsbefehlen über die Konsole authorisieren. Um keinen externen Server als Image Store verwenden zu müssen, kann hierfür das Arbeitsverzeichnis von OpenNebula /var/lib/one per NFS exportiert werden. Dieses ist vom ESXi als Datastore einzubinden. Weiterhin ist hier zunächst ein Benutzer oneadmin in der Gruppe der Administratoren anzulegen und eine Public Key Authentifizierung einzurichten. Auf dem ESXi 5 ist dafür der Public Key in die authorized_keys unter /etc/ssh/keys-oneadmin/ einzutragen. Dies ist notwendig da OpenNebula zur Steuerung des ESXi auch lokale Befehle via SSH ausführt.
Auf dem OpenNebula System ist weiterhin die libvirt, zum Zugriff auf die virsh des ESXi einzurichten. Die virsh stellt eine Administrationskonsole auf dem ESXi dar, auf die auch remote zugegriffen werden kann. Anschließend wurde über die zentrale Konfigurationsdatei /etc/one/oned.conf die Nutzung des NFS Datastore sowie VMWare Hypervisor konfiguriert. Weiterhin sind über die Konfiguration in /etc/vmwarerc, die Credentials des oneadmin-Accounts auf dem ESXi, sowie die für den Zugriff auf die virsh benötigte libvirt-URI anzugeben. Nach dem Einrichten des Netzes für die zu verwaltenden VMs erfolgte der Import des Betriebssystem Images Ubuntu 10.04 Desktop x64. Zur Erstellung einer VM ist zunächst eine Template Datei anzulegen, welche die auf dem ESXi zu konfigurierenden VM-Ressourcen, sowie das zu verwendende Image spezifiziert. Der Start einer VM war erst nach dem Anpassen zahlreicher Ruby-Dateien realisierbar. OpenNebula stellt dem ESXi beim Erstellen einer VM, über das NFS-Dateisystem die VMDK Image-Dateien zur Verfügung. Da die Entwicklung der entsprechenden

Methoden hierfür mit einer älteren Version des vSphere Hypervisor durchgeführt wurde, stimmten hier verschiedene Pfadangaben und Dateiendungen nicht mehr. Aufgrund der auf dem ESXi fehlenden Sudo Funktion konnten neben dem Starten und Stoppen von VMs keine weiteren Operationen über OpenNebula ausgeführt werden. Zur Verwendung der OpenNebula API müssen auch zunächst Template-Dateien zur Beschreibung der zu verwendenden Funktionen erstellt werden.

3.5.4 OpenStack

3.5.4.1 Architektur

OpenStack Compute alias Nova setzt sich aus den Komponenten API Server, Message Queue, Compute Worker, Network Controller Volume Worker und Scheduler zusammen. Dabei bietet der API Server die Schnittstelle zur Steuerung der OpenStack Infrastruktur über eine EC2 API sowie die native OpenStack API an. Auch wenn die Steuerung über Konsole-Kommandos abgewickelt wird, sprechen diese zunächst mit der API. Die Kommunikation zwischen den Nova Komponenten erfolgt über das Advanced Message Queue Protocol (AMQP). Als Message Queue wird dabei der Rabbit MQ Server eingesetzt. Zur Verwaltung des Lebenszyklus der VM Instanzen kommt der Compute Worker nova-compute zum Einsatz. Dieser erhält seine Anfragen über die Message Queue und liefert die durchzuführenden Operationen aus.[7] Die Netzwerkkonfiguration des OpenStack Systems erfolgt über den Network Controller nova-network. Dieser übernimmt weiterhin die Zuweisung von IP Adressen zu den VM, die VLAN-Konfiguration sowie die Konfiguration des virtuellen VM-Netzes. Dabei kommt iptables zur Einrichtung von NAT-Funktionalitäten zum Einsatz. Als DHCP- und DNS-Server wird Dnsmasq verwendet.[8] Über den Volume Worker nova-volume können VMs persistenten Speicher in Form von LVM-basierten Volumes zur Verfügung stellen. Der Scheduler nova-scheduler weist die Nova API Aufrufe den entsprechende OpenStack Komponenten zu.[7]

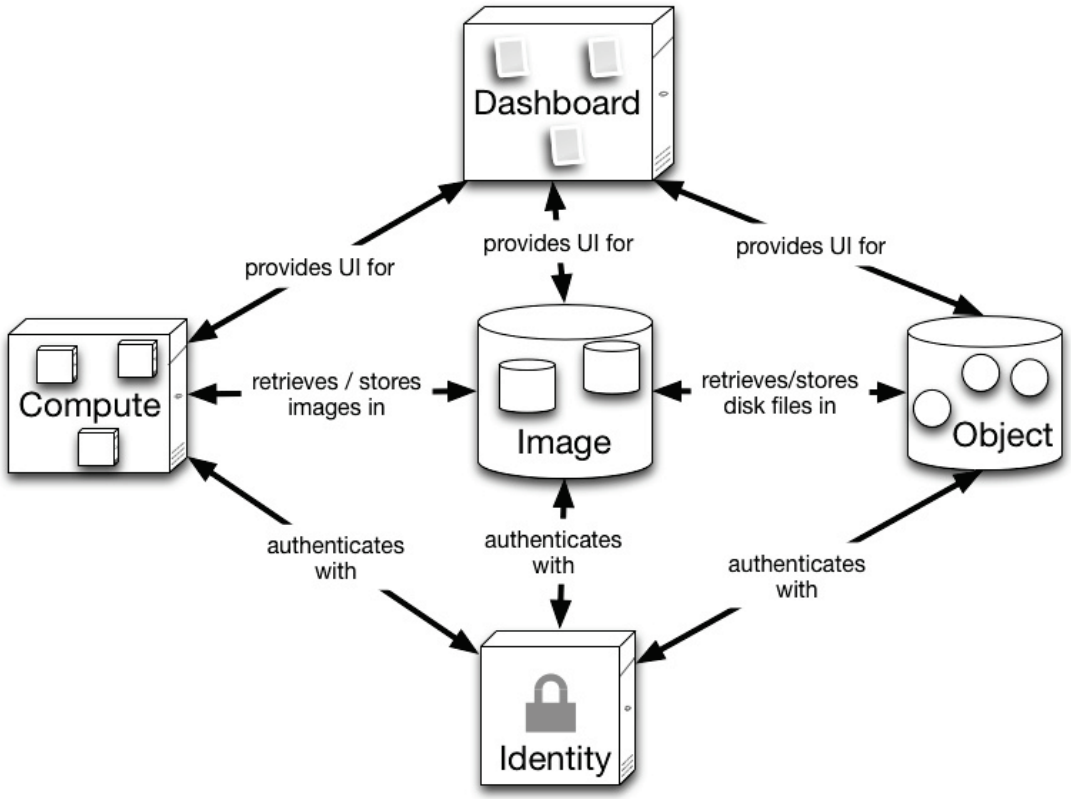

Abbildung 7: OpenStack Architektur, nach [6]

Zur Verwaltung der Virtual Machine Images setzt OpenStack den Imaging Service Glance ein. Dieser ermöglicht das Importieren von VM-Images und stellt diese nova-compute zur Zuweisung an VM-Instanzen zur Verfügung.
Weiterhin besitzt OpenStack einen Virtual Object Store namens Swift, der eine Datenspeicherung über mehrere OpenStack Systeme verteilt in Form von Objekten arrangiert. Da dieser für diese Arbeit nicht weiter relevant ist, wird darauf nicht weiter eingegangen.
Als GUI bietet das openstack-dashboard in Form eines Webinterfaces die Möglichkeit, in eingeschränkter Form die Funktionalitäten von OpenStack Compute, Image- und Object-Service grafisch zu nutzen.
Die Authentifizierung und Verwaltung von Benutzern, denen Zugriff auf die APIs gewährt werden soll, setzt OpenStack den Identity Service Keystone ein.[7]

3.5.4.2 Unterstütze Funktionalitäten

OpenStack ermöglicht den Import von VM-Images der Hypervisor Xen, KVM, QEMU, LXC, Microsoft Hyper-V und VMware ESX(i). Die Ressourcenüberwachung der VMs lässt sich nicht über Funktionalitäten der API abwickeln und kann nur lokal auf dem OpenStack System, über die libvirt API vorgenommen werden. OpenStack unterstützt eine IPv6-Addressierung der VMs über die Stateless Address Autoconfiguration. Die Erstellung von Subnets kann direkt im OpenStack System, durch die Erstellung verschiedener Netze, denen dann die entsprechen-

den VMs zugewiesen werden können, erfolgen. Für die VMs läuft die Anbindung an das entsprechende Netz über ein Bridge-Interface auf dem OpenStack System. Weiterhin kann hier auch eine Segmentierung in VLANs stattfinden. Die Unterstützung der OpenStack API ist durch die Cloud Abstraction APIs Libcloud, Deltacloud und Jclouds gegeben. Als Betriebssystem können auch hier alle mit dem eingesetzten ESXi 5 Hypervisor kompatiblen Architekturen eingesetzt werden.

3.5.4.3 Installation und Konfiguration

3.5.4.3.1 Distributionen

Neben dem OpenStack Packages für verschieden Linux Distributionen sind eigenständige Distributionen erhältlich, die bis zu einem gewissen Punkt eine geführt Installation ermöglichen. Zunächst wäre hier die Distribution Crowbar zu erwähnen, die auf einem Ubuntu System basiert. Es erfolgte im ersten Schritt die Installation der Crowbar Diablo version, welche die OpenStack Version Diablo bereitstellt. Nach einem Ubuntu-Ähnlichen Installationsverlauf kann die Grundkonfiguration von OpenStack über ein Webinterface vorgenommen werden. Die Einstellungsmöglichkeiten sind hier allerdings sehr beschränkt und beziehen sich nur auf recht allgemeine Punkte wie die Anzahl der zu verwendenden OpenStack Compute Knoten, IP-Adressbereich für VM-Netz sowie physisches Netz oder welche SAN bzw. NAS Systeme verwendet werden sollen. Nach Abschluss dieses Konfigurationsdialogs bleiben die Komponenten des OpenStack Systems bis auf die allgemeinen, angegebenen Punkte unkonfiguriert. Hier müsste dann zunächst damit begonnen werden, im Identity Service neue Benutzer, Rollen und Projekte anzulegen, um dann mit den weiteren Konfigurationen fortfahren zu können. Um zu ermitteln ob sich in der Nachfolgeversion Crowbar Essex diesbezüglich etwas verbessert hat, wurde diese zunächst neu Kompiliert und in ein aktuelles Ubuntu-Release integriert, was einen nicht unerheblichen Konfigurationsaufwand darstellt. Nach der Installation und fast der gleich ablaufenden Grundkonfiguration steht dem Nutzer allerdings wieder das rohe System zur Verfügung.
Schließlich wurde die Distribution StackOps 0.5 installiert, die ebenfalls auf einem Ubuntu-System basiert und das OpenStack Diablo Release bereitstellt. Diese besitzt ebenfalls einen Webinterface-basierten Konfigurationsdialog, für ähnliche Grundkonfigurationen wie in der Crowbar Distribution. Hier erhält der Nutzer allerdings ein vorkonfiguriertes System, das mit verhältnismäßig geringem Aufwand in Betriebsbereiten Zustand versetzt werden kann.
Installation erfolgte in einer VM auf dem ESXi mit 2GB Arbeitsspeicher, einer Single Core CPU und 100GB Festplattenspeicher, um ausreichend Platz für zu speichernde VM-Images zur Verfügung zu stellen

3.5.4.3.2 Grundkonfiguration

Über die Konsole auf dem OpenStack System kann der Befehl nova, sowie zahlreiche euca-Kommandos, wie sie auch über das Amazon Private Cloud System Eucalyptus zur Verfügung stehen, genutzt werden, um nova-compute sowie nova-network zu konfigurieren bzw. zu steuern. Über beide Befehle werden in etwa dieselben Funktionalitäten zur VM-Verwaltung bereitgestellt. Zum erfolgreichen Ausführen der Befehle sind entsprechende Benutzer-Credentials, Endpoint-URLs der EC2 und OpenStack API sowie ein Authentikation Token als Umgebungsvariablen zu registrieren. Hierzu steht im Arbeitsverzeichnis von StackOps ein entsprechendes Skript zur Verfügung. Um mit dem ESXi Hypervisor zu arbeiten ist zunächst ein Tomcat Server zu installieren und die VMware vSphere API über die Tomcat-webapps zur Verfügung zu stellen. Ist diese vorhanden kann der OpenStack mit der Soap-Schnittstelle des ESXi sprechen. Weiterhin sind die Konfigurationen in /etc/nova, zur Anpassung der Einstellungen bezüglich nova-compute und nova-network, zu überarbeiten. Hier sind zunächst, um eine Verbindung mit dem ESXi zu gewährleisten, dessen root-Credentials, IP-Adresse, URL zur vSphere API auf dem Tomcat Server anzugeben.

3.5.4.3.3 Anwendung und erweiterte Konfiguration

Zum Betrieb von VMs sind zunächst die entsprechenden Image-Dateien zu importieren. Hierzu wurde in einer VM auf dem ESXi ein Ubuntu 10.04 Desktop x64 installiert und die flat-vmdk-Datei, die das eigentlich VM-Image enthält, auf den OpenStack kopiert. Dort erfolgt der Import über einen Befehlsaufruf des Image-Service glance. Weiterhin wurde als „Privates Netz" für die VMs, das Netz 10.0.0.8/24 und als externes Netz 192.168.1.0/24 konfiguriert. Den VMs im „Privaten Netz", das über ein Bridge-Interface angebunden ist, weist der OpenStack IP-Adressen per DHCP über Dnsmasq zu. Anschließend wird auch im externen Netz eine NAT-Adresse aus dem Bereich 192.168.1.0/24 konfiguriert, die als Floating IP bezeichnet wird. Für eine Zuordnung in ein virtuelles Netz auf dem ESXi, ist eine Portgruppe mit der gleichen Bezeichnung wie das Bridge-Interface auf dem OpenStack anzulegen. Weiterhin wird standardmäßig ein Portforward von der Floating IP auf Port 22 der „Privaten IP" über iptables eingerichtet.
Beim Erstellen einer VM über den nova-Befehl ist ein Image, zusammen mit einem Flavor-Profil anzugeben. Über den Flavor werden Leistungprofile für VMs angeboten, die sich in den Größen Arbeitsspeicher, Anzahl der CPU-Kerne und Festplattenspeicher unterscheiden. Nach Absetzen des Erstellungs-Befehls kann das Anlegen der VM auf dem ESXi beobachtet werden. Erfolgt keine Konfiguration eines externen Image Stores, wie beispielsweise über einen ISCSI-Server, werden die VM-Images vom OpenStack auf den ESXi, über dessen Soap-Schnittstelle übertragen. Dabei benötigt man für die Übertragung eines 5 GB Images ca. 15 Minuten. Für den produktiven Betrieb wäre dies inakzeptabel, al-

lerdings im Rahmen erster Tests zunächst annehmbar. Nach erfolgreichem Start ist die VM über die Floating IP auf Port 22 mittels SSH erreichbar.
Die Funktionen nova suspend und nova resume funktionieren in der Diablo Version, in Verbindung mit dem ESXi Hypervisor nicht korrekt. Hier erfolgt bei einem suspend Befehl das Löschen der VM vom ESXi. Bei resume wird dieses wieder neu aus dem entsprechenden VM-Image erstellt.
Um mehr als 100 Floating IPs zu allokieren, muss der entsprechende Quota Wert, der für das Default-Netzwerk-Profil hardcoded in der Datei /var/lib/nova/nova/quota.py eingetragen ist, angepasst werden.
Im Falle der Nutzung eines KVM Hypervisor, können in die KVM-VM-Images vor dem Deploy in eine neue VM, Dateien injected werden. So ist es beispielsweise möglich Public Keys zu hinterlegen oder Konfigurationsdateien anzupassen.

3.6 Bewertung der Private Compute Cloud Management Systeme

Über OpenNebula ist es einerseits möglich sehr komplexe Konfigurationen zu erstellen andererseits ist dafür ein unverhältnismäßig hoher Konfigurationsaufwand nötig. Die Dokumentation ist zwar nicht optimal Strukturiert aber sehr Umfangreich. Die Konfiguration und das Debuggen des Systems nimmt viel Zeit in Anspruch, was gegen einen produktiven Einsatz sprechen würde. Weiterhin ist es momentan in Verbindung mit dem ESXi 5 Hypervisor nicht möglich, OpenNebula mit einem ausreichenden Funktionsumfang zur Realisierung der verteilten Tests zu betreiben.
OpenStack bietet über die StackOps Distribution, die mit einer Default-Konfiguration ausgeliefert wird, einen schnellen Einstieg in das CMS. Das System ist gut Dokumentiert und verhältnismäßig schnell an die eigenen Bedürfnisse anzupassen. Auch hier ist der Betrieb in Kombination mit dem ESXi 5 Hypervisor nicht optimal, allerdings eine klare Alternative zu OpenNebula. Aufgrund der Unterstützung und Realisierbarkeit der notwendigen Funktionalitäten, ist das OpenStack System für den weiteren Einsatz im Rahmen dieser Arbeit geeignet.

3.7 Evaluierung von Cloud Abstraction APIs

Um eine Abhängigkeit des zu entwickelten Frameworks von einem bestimmten Cloud Provider zu vermeiden, sollte ein herstellerunabhängiger Zugang zur API des ausgewählten Cloud Systems, über eine Cloud Abstraction API, implementiert werden. Ein weiterer Grund hierfür besteht in der Gewährleistung des alternativen Einsatzes einer Private Cloud. Die Verwendung einer Cloud Abstraction API ermöglicht eine einheitliche Verwaltung der Cloud Ressourcen und sollte möglichst einen Großteil, der durch die API der Cloud Systeme angebotenen Funktionalitäten unterstützen. Nachfolgend werden die zum Zeitpunkt der Evaluierung vorliegenden IaaS Cloud Abstraction APIs, mit einer weit gestreuten Providerunterstützung Libcloud, Deltacloud und Jclouds näher untersucht.

3.7.1 Libcloud

Die in Python implementierte Cloud API bietet z.Z. die größte Anzahl an unterstützenden Cloud Providern, bzw. Cloud Systemen, für eine Cloud Abstraction API (Bluebox, Brightbox, CloudSigma, Dreamhost, Amazon EC2, enomaly CP, ElasticHosts, Eucalyptus, Gandi.net, GoGrid, IBM Cloud, Linode, Nimbus, OpenNebula (v1.4 & v3.0 API), OpenStack (v1.0 & v1.1 API), OpSource Cloud, Rackspace, RimuHosting, Slicehost, SoftLayer, Terremark, vCloud, Voxel, VPS.net, skalicloud, serverlove, Ninefold). Die von Libcloud unterstützten gemeinsamen Funktionen, werden für jede Provider-spezifische API, über Treiber zur Verfügung gestellt. Neben Funktionen zur VM-Verwaltung bietet die Libcloud API abstrahierten Zugriff auf Storage-, Loadbalancing- und DNS-Management-Lösungen in der Cloud. Die Funktionen zur VM-Verwaltung der API umfassen[12][13]:

- list_nodes
- list_images
- list_sizes
- list_locations
- create_node
- deploy_node
- reboot_node
- destroy_node
- _wait_until_running
- _ssh_client_connect
- _run_deployment_script

3.7.2 Deltacloud

Die mit dem Ruby Framework Sinatra entwickelte Deltacloud API wird von den meisten größeren Cloud Providern, bzw. populäreren Cloud Systemen unterstützt (Amazon EC2, Eucalyptus, IBM SBC, GoGrid, OpenNebula, Rackspace, RHEV-M, RimuHosting, Terremark). Der Zugriff auf die angebotenen Funktionen der Deltacloud API durch einen aufrufenden Client, erfolgt über einen Deltacloud Server, mit dem die Datenkommunikation über HTTP als Dienst-basierte REST (representational state transfer) Schnittstelle, abgewickelt wird. Hierfür werden ein CLI-Tool sowie Client-Bibliotheken in Ruby, C (Libdeltacloud) und Python angeboten. Der Vorteil dieses Ansatzes besteht in der Nutzung von bestehenden Standards wie HTTP und XML, wodurch eine Platform- und Programmiersprachenunabhängigkeit erreicht wird. Die Deltacloud API bietet neben Funktionen zur VM-Verwaltung einen abstrahierten Zugriff auf Storage- und Loadbalancing-Lösungen in der Cloud. Die Basis-Funktionen zur VM-Verwaltung der API unterstützen das Erstellen, Starten, Stoppen und Löschen von VM-Instanzen sowie die Abfrage von Hardware Profilen, Details zum Betriebssystem und Informationen über bestehende VM-Instanzen und Ressourcenstandorte.[14]

3.7.3 Jclouds

Die in java realisierte Cloud Abstraction API bietet ebenfalls eine Unterstützung für zahlreiche Cloud Systeme (aws-ec2, bluelock-vcloud-vcenterprise, bluelock-vcloud-zone01, cloudservers-uk, cloudservers-us, cloudsigma-zrh, elastichosts-lon-b, elastichosts-lon-p, elastichosts-sat-p, eucalyptus-partnercloud-ec2, gogrid, openhosting-east1, rackspace-cloudservers-uk, rackspace-cloudservers-us, rimuhosting, serverlove-z1-man, skalicloud-sdg-my, slicehost, stratogen-vcloud-mycloud, trmk-ecloud, trmk-vcloudexpress). Die API wird als Apache Maven Projekt angeboten und kann daher unkompliziert in eigene Maven Projekte, durch Angabe der entsprechenden Dependencies importiert werden. Jclouds bietet weiterhin den größten Umfang an unterstützten Funktionen. Neben dem Erstellen, Starten, Stoppen und Löschen von VMs werden Funktionen zur Datei-Injection, SSH-basierten Skriptausführung sowie für einen Suspend und Resume angeboten. Neben den Funktionen zur VM-Verwaltung können auch über jclouds Storage- und Loadbalancing-Systeme angesprochen werden. Aufgrund der Vorgabe, eine Realisierung in Java zu entwickeln, fällt die Wahl für die weiter zu verwendende Cloud Abstraction API auf jclouds. Diese bietet weiterhin den vergleichsweise größten Funktionsumfang und kann einfach über Maven angebunden werden.

3.7.4 Evaluierung der Interoperabilität mit VMware ESXi ohne Einbindung eines Cloud Management Systems

Vor der Evaluierung der Private Cloud Systeme wurde die Möglichkeit untersucht, mit einer Cloud Abstraction API, ohne die vermittelnde Zusatzkomponente CMS, die Steuerung des ESXi vorzunehmen. Diese Option ist mit der Unterstützung der ESXi / VSphere API, durch die Abstraction API Deltacloud gegeben. Die möglichen Funktionalitäten wurden zunächst, mit der in Ruby implementierten Deltacloud API untersucht. Zur Durchführung der Tests, erfolgte zunächst die Bereitstellung einer x64 Ubuntu 10.04 LTS Desktop Installation, auf einer dafür erzeugten VM des ESXi. Anschließend erfolgte die Auflösung erforderlicher Paket-Abhängigkeiten im Ubuntu System selbst sowie im Ruby Paketverwaltungssystem RubyGems, über das dann die Installation des Deltacloud-Daemon vorgenommen wurde. Dem Daemon muss beim Start, die zu abstrahierende Schnittstelle als Driver-Parameter mitgegeben werden.

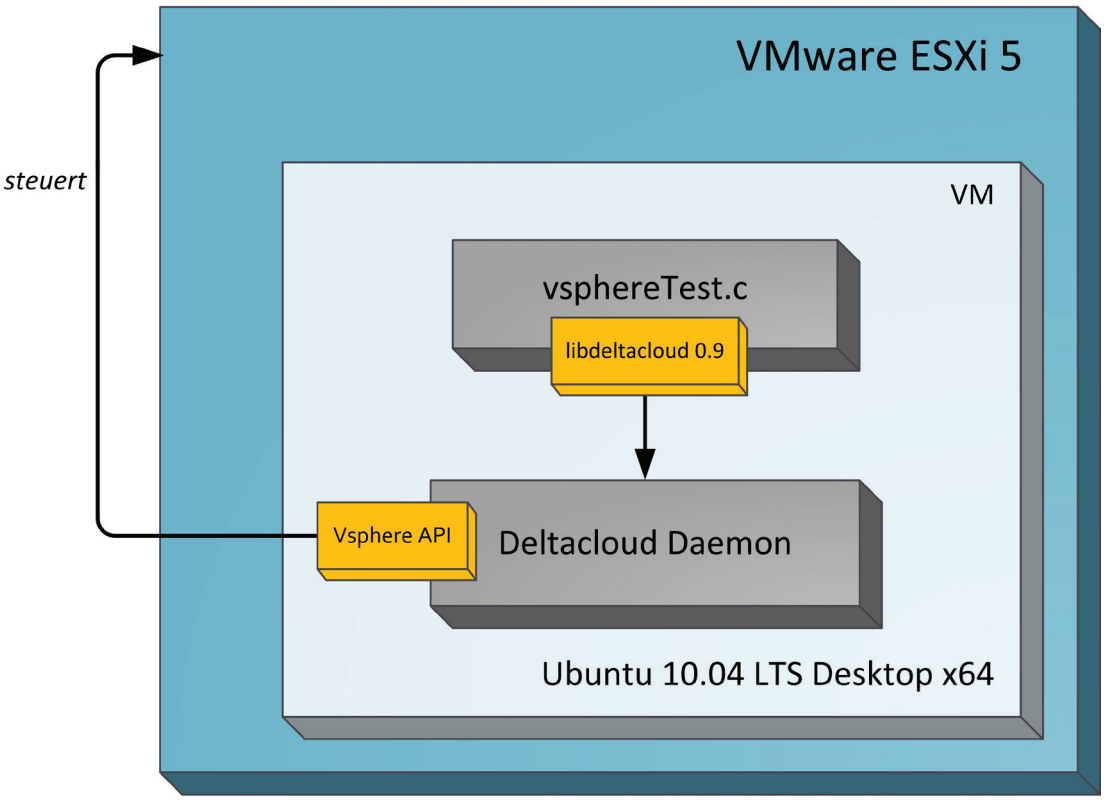

Abbildung 8: Steuerung des Hypervisors über die Abstraction API Deltacloud

Die durchzuführenden Tests erfolgten, wie in Abbildung 8 dargestellt, über die Implementierung einer Testanwendung in C, welche die von Deltacloud bereitgestellte C API libdeltacloud, für den Zugriff auf den Daemon verwendet. Der Daemon spricht die Funktionen des Hypervisors über dessen SOAP-Schnittstelle, mit Hilfe der von VMware bezogenen VSphere API, an. Der Test bezieht zunächst Informationen zu den auf dem ESXi laufenden VMs, mit den Daten zu Owner,

Image, Datastore und State, welche bis auf die Image ID sowie URL, korrekt zurückgeliefert werden. Weiterhin wurde versucht eine neue VM, über ein auf dem ESXi vorhandens Image zu erstellen. Dies schlug allerdings fehl, da auch hier über die Image ID, keine Image Datei aufgefunden werden konnte. Die Ursache dafür ist, dass der Image-Pfad auf dem Hypervisor beim Abrufen durch die VSphere API nicht korrekt zurückgegeben wird. Hier sollte eigentlich das Root-Directory, der zu einer VM zugehörigen Dateien übermittelt werden. Es stellte sich heraus, dass die Deltacloud VSphere Driver noch mit der VSphere API 4.1 entwickelt und entsprechend mit dem ESXi 4.1 getestet wurden. Im durchgeführten Test wurde hingegen mit der VSphere API 5 gearbeitet. Bis Version 4.1 bot VMWare den Hypervisor als Lizenzpflichtige ESX- und freie ESXi-Version an. Die ESXi-Version unterscheidet sich in verschiedenen Funktionen wie beispielsweise einer eingeschränkteren Service-Konsole sowie limitierten Konfigurationsgrößen. Ab Version 5 ist nur noch ein ESXi Produkt mit kostenfreier, oder verschiedenen kostenpflichtigen Lizenzen erhältlich. Die Unterschiede entsprechen denen der Vorgängerversionen zwischen ESX und ESXi.
Anschließend erfolgte noch eine Installation des ESXi 4.1, in einer VM auf dem ESXi 5 sowie die Bereitstellung der VSphere 4.1 API für den Deltacloud Daemon. Das Ergebnis war allerdings das Selbe.
Hieraus ergab sich, dass eine direkte Ansteuerung des ESXi über Deltacloud derzeit nicht praktikabel ist und ein CMS zur Cloud-Steuerung benötigt wird.

3.8 Vorstellung eines Frameworks für verteilte JUnit Tests

Während der Recherche wurden mehrere Möglichkeiten für die verteilte Ausführung von JUnit Tests, wie Beispielsweise mit GridGain gefunden, bei der zur Performance-Steigerung ein JUnit Test parallelisiert auf mehreren Systemen abgearbeitet wird. Ziel der Recherche war allerdings, evtl. ein Framework zu finden, das JUnit Tests von verteilten Anwendungen beherrscht. Hier viel schließlich das Open-Source-Projekt Pisces auf, welches eine Erweiterung des JUnit Frameworks darstellt und eine Testdurchführung in verteilten Testumgebungen ermöglicht. Dies ist durch TestSuites realisiert, bei denen die einzelnen Tests entfernt, auf verschiedenen Systemen, parallel oder sequenziell ausgeführt werden können. Diese entfernten Tests sind in normale JUnit-Tests verpackt und werden lokal ausgeführt, so dass hier gewöhnliche JUnit GUI Tools Verwendung finden können. Dies wird durch das eingeführte Wrapper-Objekt RemoteTestCase, eine Ableitung des TestCase Objekts, ermöglicht.

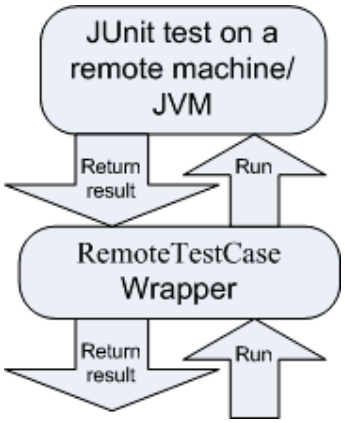

Abbildung 9: Ausführung eines RemoteTestCase, nach [6]

Auf jedem entfernten System muss ein Pisces Agent, zur Ausführung des eigentlichen JUnit-Tests und dem Zurückliefern der Testergebnisse an den lokalen Test, betrieben werden. Der Agent ist in Java implementiert und nimmt die Befehle des lokalen test runners entgegen. Jeder Agent führt ein Set von JUnit-Tests aus, wobei jeder Test seine Informationen in den default output schreibt, von welchem diese dann in die Konsole der lokalen TestSuite übertragen werden. Die Kommunikation zwischen Agents und lokaler TestSuite erfolgt in der Standardkonfiguration von Pisces über Multicast-Nachrichten. Nach der Installation einer Message Oriented Middleware, kann die Nachrichtenübertragen, auch über die Anbindung einer JMS-Implementierung erfolgen.[6]

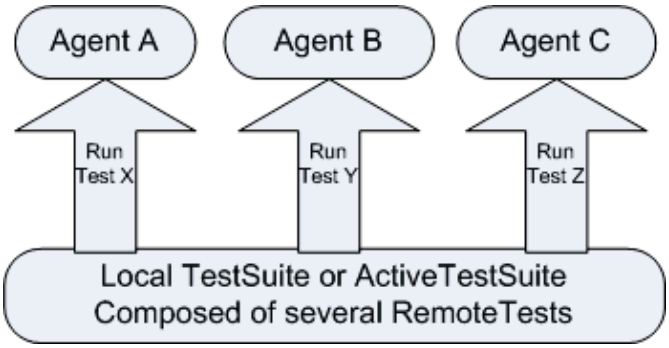

Abbildung 10: TestSuite mit mehreren entfernten Tests, nach [6]

3.9 Definition des Konzepts

Nachfolgend wird das für die Realisierung der automatisierten, verteilten Tests erarbeitete Konzept, aufbauend auf der in Kapitel 3.2 beschriebenen Grobstruktur erläutert. Die Darstellung erfolgt unterteilt in die vier Phasen VM Setup und Testinitiierung, Cloud Management, Configuration und Test Management sowie verteilte JUnit Tests.

3.9.1 VM Setup und Testinitiierung

Die erste Phase des Konzepts zum VM Setup und der Testinitiierung wird in Abbildung 11 dargestellt.
Nachdem ein Testcase entworfen wurde, ist für die Implementierung des Tests mit Hilfe der verteilte Test API, zunächst eine Konfiguration, für die bereitzustellenden VMs, zu erstellen. Hier sind Informationen, wie beispielsweise der zur Verfügung zu stellende Arbeitsspeicher und das zu verwendende Betriebssystem, in Form einer Konfigurationsdatei, anzugeben. Weiterhin wurde vorgesehen, die Erstellung von VM-Gruppen zu ermöglichen, um dadurch später im Test, effektiver Operationen ausführen zu können, die auf mehreren VMs identisch sind.
Anschließend kann die Initiierung des VM Setup, zur automatisierten Ausführung von Operationen bzgl. VM-Verwaltung, über das Webinterface des Jenkins CI Servers erfolgen. Das VM Setup ist eine eigenständige Komponente und wird durch die Interaktion des Entwicklers ausgeführt. Zunächst war geplant, das VM Setup beim Ausführen des JUnit Tests mit zu initiieren. So hätte der gesamte verteilte Test in einem Zuge, von der Konfiguration der VMs, über das hochfahren der Systeme, bis hin zur Testausführung sowie abschließendem Stoppen der VMs abgewickelt werden können. Da allerdings das Starten, bzw. Erzeugen der VMs einen zu großen Zeitaufwand darstellt, wurde dieser Vorgangausgelagert. So können zum einen neue VMs vor der eigentlichen Testausführung erzeugt sowie gestartet und evtl. vorhandene, laufende VMs direkt verwendet werden. Hier ist vorgesehen, eine Build-Konfiguration auf dem Jenkins Server zu erstellen, die mit Hilfe von anzugebenden Parametern, die gewünschte Operation an das VM Setup, über Umgebungsvariablen, übermittelt. Der testende Entwickler kann dann eine der Angebotenen Operationen (z.B. Starten, Stoppen von VMs) auswählen und eine Gruppe von VMs selektieren, auf die die Operation angewendet werden soll.
Zur Ausführung der JUnit Tests sowie des VM Setup über den Jenkins Server, ist auf dem Hypervisor ein Jenkins Build-Knoten bereitzustellen , der den über das Webinterface konfigurierten Build ausführt. Die Komponente des VM Setup steht im SVN Repository, für den Code Checkout auf dem Build-Knoten und der anschließenden Ausführung, bereit.
Während der Bereitstellung der zuvor definierten VMs durch CMS und Hypervisor, kann der Entwickler beginnen den JUnit Test zu schreiben. Dieser beinhaltet die eigentliche Testlogik und nutzt dabei u.A. Methoden zur Skriptausführung sowie SCP-Funktionen, der zu implementierenden verteilten Test API. Mit der verteilten Test API werden weiterhin notwendige Resource-Files, wie auszuführende Skripte und SSH Private Keys für die Public Key Authentifizierung, bereitgestellt. Der JUnit Test kann lokal auf dem Entwickler-System oder über den Jenkins Server, auf dem Build-Knoten ausgeführt werden, falls eine Zeitsteuerung oder wiederholte Ausführung gewünscht wird. Soll die Ausführung des JUnit Test auf dem Build-Knoten erfolgen, ist der entsprechende Code zunächst in das SVN Repository einzuchecken. Der Jenkins Server erkennt während der automa-

tischen Prüfung die Code-Änderung auf dem SVN Repository und veranlasst den Code Checkout sowie die Build-Initiierung und damit die Ausführung des JUnit Tests, auf dem Build-Knoten.

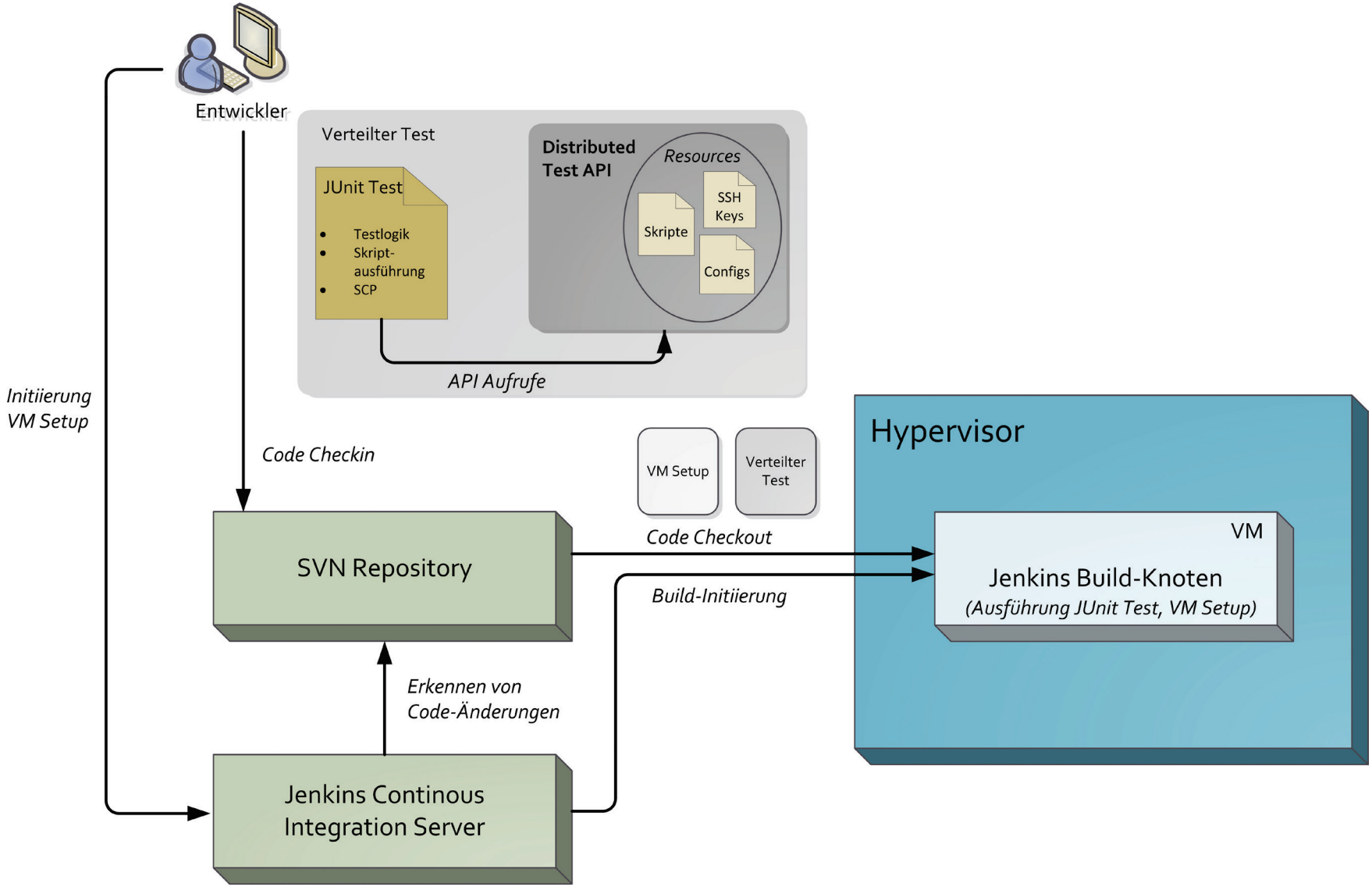

Abbildung 11: Konzept (1) - VM Setup und Testinitiierung

3.9.2 Cloud Management

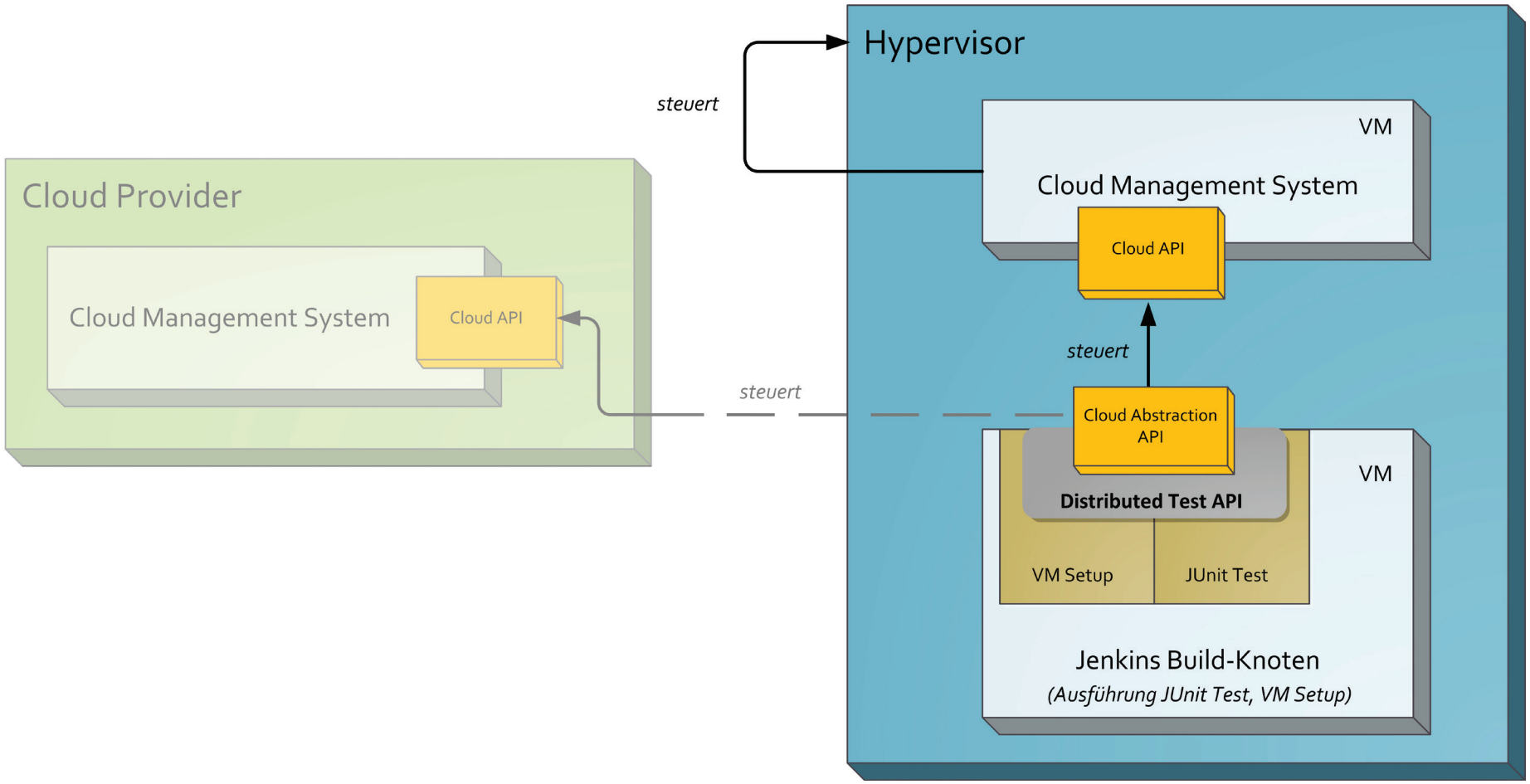

Abbildung 12: Konzept (2) - Cloud Management

Beim Ausführen des VM Setup auf dem Build-Knoten, wird die als Parameter übergebene Umgebungsvariable ausgewertet und die entsprechende Operation, auf die angegebene Gruppe von VMs, ausgeführt. Das VM Setup nutzt dazu die verteilte Test API, welche sich wiederum der Funktionen, der Cloud Abstraction API bedient, um die Cloud API des CMS anzusprechen. Das CMS führt die entsprechenden Operationen aus und interagiert dabei entsprechend mit dem Hypervisor.
Neben dem VM Setup kann auch der verteilte Test auf dem Build-Knoten ausgeführt werden. Dabei nutzt der JUnit Test ebenfalls die verteilte Test API, welche sich auch hier wiederum der Funktionen, der Cloud Abstraction API bedient, um die Cloud API des CMS anzusprechen.
Neben dem Private Cloud System, kann auch ein Public Cloud System über die Cloud Abstraction API angebunden werden. Dazu sind die spezifischen Daten zur Anbindung an das entsprechende System durch die Cloud Abstraction API, abzuändern und die korrekte Funktionsweise der verwendeten Methoden zu überprüfen.

3.9.3 Configuration und Test Management

Über die Ausführung des JUnit Test wird schließlich der verteilte Test gestartet. In Abbildung 13 ist wieder der Fall, der Ausführung des JUnit Test auf dem Jenkins Build-Knoten, dargestellt. Über die Anbindung des VM-Testnetzes an das DLR-Netz mittels IPCop Firewall wird ebenfalls gewährleistet, diesen auf dem System des Entwicklers auszuführen.
Im JUnit Test können über Methodenaufrufe der verteilten Test API Skriptausführungen sowie SCP Up- und Downloads, auf den über die VM Konfiguration erstellten VMs ausgeführt werden. So können dann die verteilten Anwendungen mit individuellen Konfigurationen auf die jeweiligen Testsysteme aufgebracht und die entsprechenden Tests ausgeführt werden. Über das CMS lassen sich neben einem gemeinsamen VM-Testnetz auch mehrere Subnetze erstellen, um die Durchführung erweiterter Testszenarien zu gewährleisten. Über beispielsweise zurück kopierte Log-Daten und deren Auswertung, erfolgt im JUnit Test schließlich die Einstufung des Ergebnisses als Erfolg oder Fehlschlag. Bei der Ausführung über den Jenkins Build-Knoten können diese Ergebnisse nach Abschluss des Tests in einer Übersicht der Testergebnisse betrachtet werden.

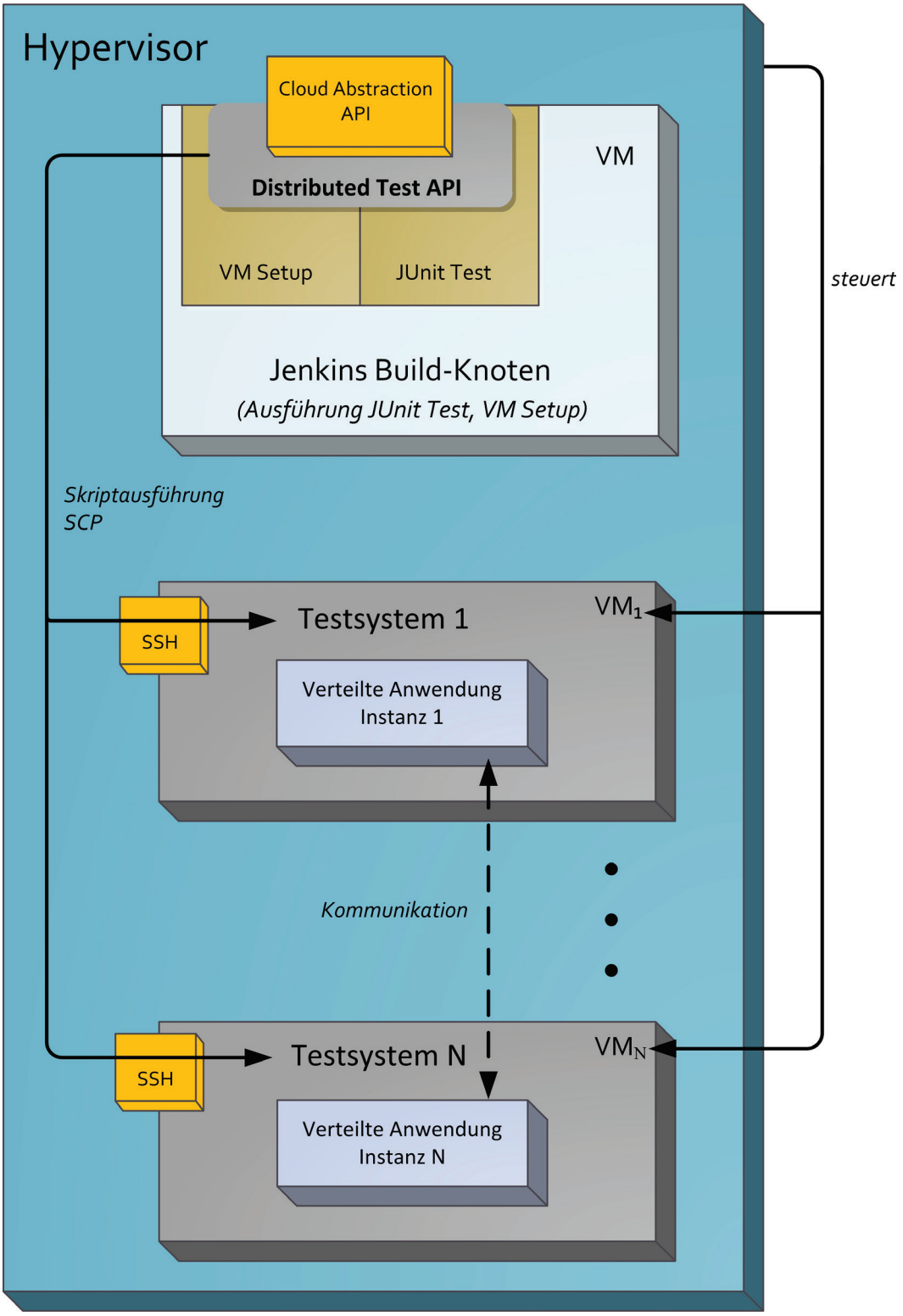

Abbildung 13: Konzept (3) - Configuration und Test Management

3.9.4 Verteilte JUnit Tests

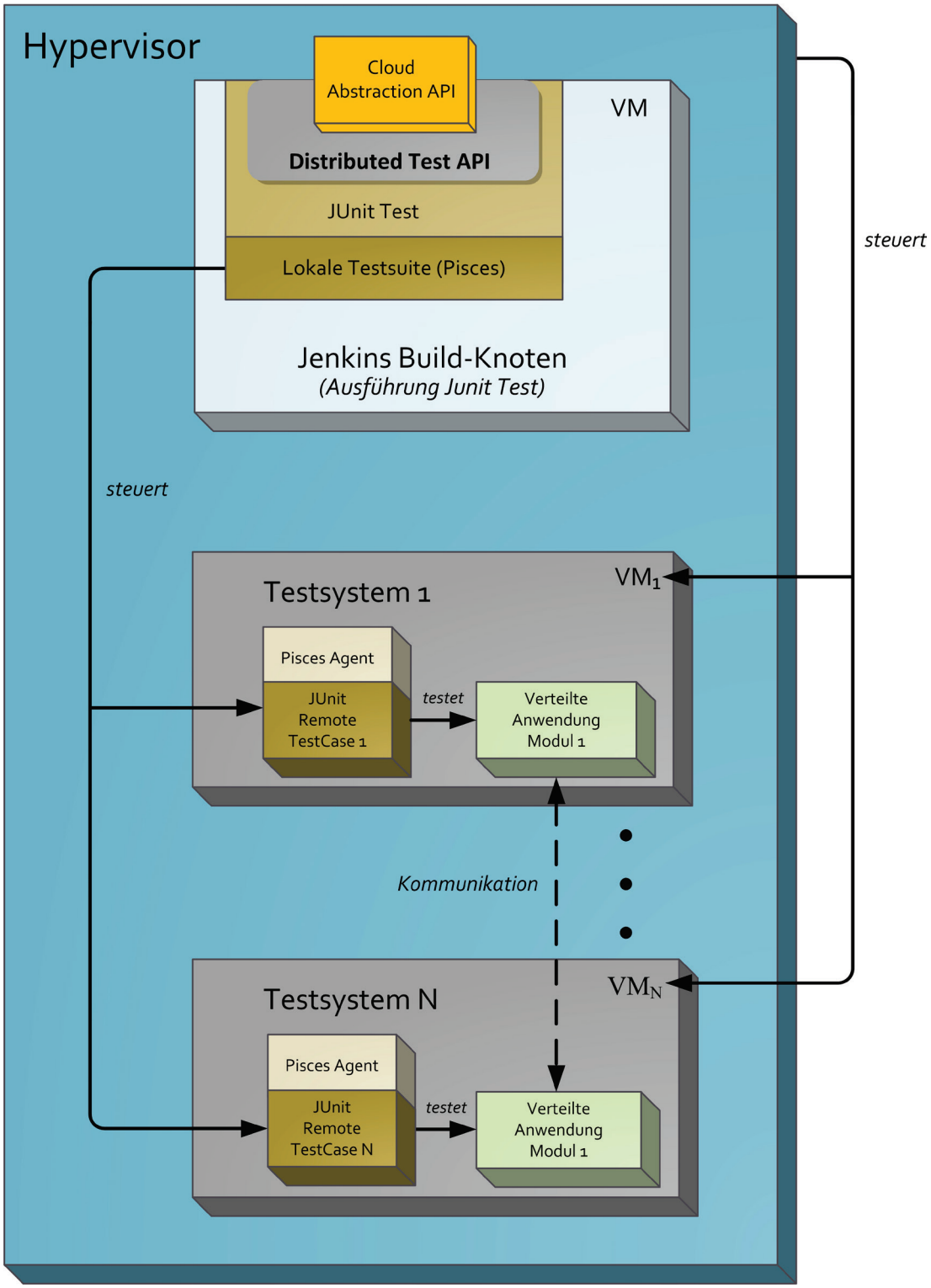

Abbildung 14: Konzept (4) - Verteilte JUnit Tests

Die Implementierung von verteilten JUnit Tests lässt sich über das in Kapitel 3.8 vorgestellte Pisces Framework realisieren. Zur Ausführung der lokalen Testsuite von Pisces kann weiterhin der JUnit Test verwendet werden, der zuvor die ei-

gentliche Testlogik enthielt. Die Installation des Pisces Agent erfolgt bei der Vorbereitung des einzusetzenden VM-Images.
Mit Hilfe der verteilten JUnit Tests können einzelne Module der verteilten Anwendung auf verschiedenen Systemen, beispielsweise während eines Kommunikationsvorgangs, parallel getestet werden. Hierzu sind vom Entwickler zunächst die entsprechenden JUnit Tests, für die zu testenden Module zu schreiben, welche dann mittels Pisces als Remote TestCases bereitgestellt werden. Bei der Testausführung wird anschließend der lokale testrunner von Pisces, die Remote TestCases über die Agents ausführen und die Testergebnisse mit Hilfe der Testsuite an den initialen JUnit Test zurückliefern.

3.10 Bewertung des Konzepts

Die in die Phase VM Setup und Testinitiierung integrierten Systeme des SVN Repository und Jenkins Server, stammen aus dem Einfluss der CI Entwicklungsumgebung der Abteilung SC. Da dies ein, in diesem Bereich weit verbreiteter Ansatz ist, wurden die Vorteile auch im Rahmen dieser Arbeit, sowie zur Anbindung an die vorhandene Entwicklungsumgebung genutzt. So ergeben sich durch den Jenkins Server Möglichkeiten zur Automatisierung sowie Steuerung zeitlicher Abläufe der Tests. Weiterhin stellt der Jenkins Server über die, mit geringem Aufwand zu konfigurierenden Build-Knoten, eine bereits an das System angebundene Ausführungsumgebung bereit.
In der Phase des Cloud Management ergab sich der Einsatz eines CMS durch die nicht Realisierbare Steuerung des Hypervisors durch die Cloud Abstraction API, welche wiederum aus der Anforderung bzgl. Providerunabhängigkeit und alternativen Einsatz eines Private Cloud Systems ihren Einsatz fand. Es ergibt sich zwar durch jede Abstraktion, eine Reduktion der Funktionalitäten am Übergang zwischen den verschiedenen APIs, der Gewinn an Flexibilität ist an dieser Stelle allerdings wichtiger.
Im Bereich Configuration und Test Management können bestehende Frameworks zur verteilten Testautomatisierung ausschließlich Aufgaben des Instanzmanagement einer zu testenden Software sowie der Testausführung- und Verwaltung erfüllen. Einen Vertreter dieses Frameworks, das einen großen Umfang an Funktionen in diesen Bereichen liefert ist das Software Testing Automation Framework (STAF).
Während der initialen Recherchen wurde eine Arbeit gesichtet, die einen Ähnlichen Anwendungsfall behandelt. Im Proposal der Bachelorarbeit an der Universität Göteborg, zum Thema „STAF-on-Eucalyptus: A Cloud Based Software Testing Environment for Distributed Systems“, wird eine Konzeptidee zur Realisierung von Lasttests für verteilte Systeme dargestellt. Der Ansatz hier ist, über eine Benutzerschnittstelle zunächst die notwendigen Cloud-Ressourcen im CMS Eucalyptus bereitzustellen und anschließend die Steuerung der System Tests über STAF abzuwickeln.[15] Diese wäre noch vergleichbar mit der, im Konzept beschriebenen Durchführung von VM Setup und Ausführung der JUnit Tests, welche

für die Realisierung der System Tests auf den VMs zuständig sind. STAF wird im dort beschriebenen Konzept allerdings nicht mit an das CMS angebunden, wodurch der Anwender Cloud- bzw. VM-System-spezifische Informationen manuell übertragen muss. Im Konzept dieser Arbeit wird die Interoperabilität von CMS, VM Setup und JUnit Test durch die verteilte Test API gewährleistet.
Das von IBM initiierte Open-Source-Projekt STAF stellt ein Framework zur Realisierung verteilter Testautomatisierung in einer Peer-to-Peer Umgebung bereit. Weiterhin liefert es mit STAX (STAF Execution Engine) eine Ausführungsumgebung sowie eine GUI zur Verwaltung und Darstellung der durchzuführenden Tests und deren Ergebnisse. Das Konzept von STAF sieht vor, dass auf den Systemen bzw. VMs, die als STAF-Knoten fungieren, das STAFProc, eine Art RPC-Agent läuft, der an einen TCP-Port gebunden ist und STAF-Kommandos empfängt. Die STAF-Knoten bieten so Dienste an, mit denen u.A. Dateisystemoperationen und konsolenbasierte Programmaufrufe ausgeführt sowie zeitliche Abläufe gesteuert werden können. Bei der Bereitstellung der Funktionalität zur Automatisierung mittels STAX werden die Auszuführenden Jobs über eine XML-basierte Sprache beschrieben und in der Execution Engine verarbeitet. STAX ist ebenfalls ein STAF Dienst und kann andere STAF Dienste nutzen. Die Jobs können zudem über verschiedene GUI-Komponenten ausgeführt und überwacht werden. Der „Job Monitor" zeigt beispielsweise aktive Prozesse der Jobs sowie Informationen zu einzelnen, ausgeführten Testfällen und bietet weiterhin Möglichkeiten zum Logging und Debugging.[16][17]
Mit STAF wäre es somit nur möglich, den Teil des Konzepts zum Configuration und Test Management zu realisieren. Dabei würde allerdings mehr Konfigurations-Overhead durch die Einrichtung des Frameworks und die Anbindung über die verfügbare API entstehen, als Nutzen aus den zusätzlichen Funktionen gezogen werden könnte. STAF dient zur Realisierung von System Tests, welchen im Rahmen dieser Arbeit, nur weniger Komplexe Funktionalitäten abverlangt werden. Weiterhin ergibt sich auch hier durch jede weitere Abstraktion, eine Reduktion der Funktionalitäten am Übergang zwischen den verschiedenen Systemen. So ist es hier beispielsweise effektiver, Befehle an eine VM, direkt über SSH abzusetzen, als dies, stellvertretend durch den entsprechenden STAF-Dienst, zu erledigen. Selbiges gilt auch für die Dateioperationen, die äquivalent mittels SCP ausführbar sind. Die erweiterten Funktionalitäten von STAF, wie beispielsweise die Steuerung der zeitlichen Abläufe beim Testen, wären zwar für die Realisierung der automatisierten, verteilten Tests ebenfalls relevant, stehen während der ersten Entwicklung in dieser Arbeit allerdings nicht im Vordergrund.
Da für die Realisierung der verteilten JUnit Tests bereits eine geeignete Lösung in Form des Pisces Framework existiert, wurde dieses in das Konzept mit eingebunden. Für diese Lösung sprachen die unkomplizierte Möglichkeit zur Anbindung an den, für die Testausführung zuständigen JUnit Test sowie das einfache Bereitstellen des Pisces Agent, im vorbereiteten VM-Image.

4 Implementierung des Konzepts

Bei der Implementierung des Konzepts wurde auf den, während der Evaluierungen vorbereiteten Systemen des OpenStack CMS, ESXi sowie der IPCop Firewall aufgebaut und diese weiter konfiguriert.

4.1 Bereitstellung von VM-Images

Für die Tests während der Implementierung wurde zunächst ein Linux Image konfiguriert. Um beim Betrieb der verteilten Tests später auch ein Windows System zur Verfügung zu stellen, erfolgte anschließend die Untersuchung, wie sich Windows in das bestehende Konzept integrieren lässt.
Nach der Konfiguration der VM-Images über den VSphere Client auf dem ESXi, wurden die entsprechenden VMDK Image Files in das OpenStack System importiert und standen für die VM-Erzeugung zur Verfügung.

4.1.1 Linux VM

Als Betriebssystem wurde ein Ubuntu 12.04 x86 Server gewählt, da die aktuelle Ubuntu Version ein häufig eingesetztes Betriebssystem, in den Nutzerkreisen der zu testenden verteilten Anwendung RCE, darstellt. Aufgrund der Testausführung auf Konsole-Ebene mittels SSH ist es nicht notwendig, eine grafische Oberfläche zur Verfügung zu stellen. Dadurch ergibt sich hier eine entsprechende Ressourceneinsparung durch den Einsatz der Server-Version.
Zunächst wurde eine neue VM, mit den während der Anforderungsdefinition festgelegten Leistungsgrößen eingerichtet. Entsprechend diesen Vorgaben wurden zunächst eine 32 Bit Linux VM mit 2GB Arbeitsspeicher, 5,5 GB Festplattenspeicher, eine Single Core CPU sowie einer Netzwerkschnittstelle mit e1000 Treiberkompatibilität und der Zuordnung zur Testnetz-Portgruppe konfiguriert.
Anschließend erfolgte die Installation des Betriebssystems über das entsprechende ISO Image. Zur Bereitstellung der JVM, für die Ausführung von Java-Applikationen, insbesondere RCE sowie JUnit Tests, wurde das Paket openjdk-6-jre installiert. Die Wahl fiel hier auf das OpenJDK, da dieses von den meisten Anwendern eingesetzt wird. Weiterhin wurde ein neuer Benutzer „tester" in der Gruppe „users", zur Ausführung der Tests mittels SSH angelegt. Zur Public Key Authentifizierung wurde weiterhin ein Schlüsselpaar erstellt und der Public Key in die authorized_keys des Benutzers „tester" eingetragen.

4.1.2 Windows VM

Um auch die Realisierung, der in den Anforderungen beschriebenen Tests, unter Windows zu gewährleisten, wurde zunächst mit dem IT Manager der Abteilung, ein geeignet lizenziertes Windows Betriebssystem bestimmt. Dieser konnte ein Windows XP Professional 32 Bit SP3 in Form eines VMWare ESX Images, das für

Jenkins Windows Build-Knoten eingesetzt wird, bereitstellen. Das Image wurde in den ESXi 5 importiert und die Ressourcenkonfiguration der VM auf 2GB Arbeitsspeicher sowie eine Netzwerkschnittstelle im Testnetz angepasst. Die zugewiesene Festplattenkapazität von ca. 8GB wurde beibehalten, da ca. 2GB freier Festplattenspeicher gewährleistet waren. Im Betriebssystem wurde hier ebenfalls ein Benutzer „tester" zur entfernten Ausführung von Befehlen angelegt. Die Installation einer JVM war nicht erforderlich, da hier bereits ein Java SE 6 mit dem vorkonfigurierten VM-Image bereitgestellt wurde.
Aufgrund der im Rahmen der Testausführung benötigten Skriptausführung über SSH sowie erforderlichen SCP Funktionalitäten war es notwendig, einen geeigneten SSH Daemon unter Windows zur Verfügung zu stellen. Auch hier waren Open Source, bzw. kostenfreie Produkte bevorzugt auszuwählen. Im Bereich Freie Windows SSH Daemon (Frei, nicht nur auf eine personal/home Edition bezogen) konnte nach durchgeführten Recherchen allerdings nur der freeSSHd und eine ältere Version des OpenSSH Daemon von 2004 ermittelt werden. So wurde zunächst freeSSHd in der Version 1.2.4 getestet. Dieser läuft nach der Installation als eigenständiger Server und wird nicht über die Windows Dienstverwaltung zur Verfügung gestellt. Beim Testen der SSH-Verbindung mittels Putty sowie von einem Linux System per SSH-Kommando wurde festgestellt, dass während einer bestehenden Verbindung u.U. der SSH Daemon abstürzt. Einen weiteren Kritikpunkt stellt die fehlende SCP-Unterstützung dar, die für die Realisierung des Datentransfers zwischen JUnit Test und Test-VM benötigt wird.
Eine weitere Möglichkeit zur Bereitstellung eines SSH Daemon unter Windows, wäre der Einsatz des Apache MINA SSHD. Apache MINA (Multi-purpose Infrastructure for Network Applications) stellt ein Open Source Java Framework zur Entwicklung von Netzwerkanwendungen bereit und bietet über eine API auch die Möglichkeit zur Realisierung eines SSHD mit Funktionalitäten für SCP und Public Key Authentifizierung. Da hier allerdings ein zusätzlicher Entwicklungsaufwand entstehen würde, wäre eine Lösung mit einem installierbaren SSH Daemon effektiver.
Zur Überprüfung der Realisierbarkeit, eines in diesem Szenario korrekt funktionierenden Windows SSH Daemon mit Unterstützung von SCP- und Public Key Funktionalitäten, wurde ein für die gewerbliche Nutzung kostenpflichtiges Produkt getestet. Hier fiel die Wahl auf den Bitvise SSH Server in der Version 5.51, da dieser eine zeitbeschränkte Evaluierungsversion bereitstellt und die genannten Anforderungen unterstützt. Dieser wird ebenfalls als eigenständiger Server bereitgestellt und ermöglicht den Import von Public Keys mit der Zuordnung zu einem Windows Benutzerprofil. Die Tests mittels Putty und Linux-Konsole ausgeführten SSH-Sessions über Public Key Authentifizierung verliefen erfolgreich. Auch die Nutzung der SCP Funktionalitäten war Problemlos möglich.
Um noch eine mögliche Realisierung mit einem Open Source SSH Daemon zu evaluieren, wurde schließlich der OpenSSH Daemon für Windows in der Version 3.8p1-1 getestet. Dieser Konfiguriert während der Installation eine minimale Cygwin-Umgebung zum Betrieb des sshd unter Windows. Der OpenSSH Daemon

wird hier als Windows Dienst bereitgestellt während die Konfiguration über Konfigurationsdateien und entsprechende Kommandos erfolgt. Hier wurde zunächst der Windows Benutzer „tester" in die Cygwin-Umgebung überführt. Weiterhin erfolgte das Importieren des Public Key, wie auf Linux-Systemen, in die authorized_keys, welche sich hier im Windows Home-Verzeichnis ebenfalls unter .ssh befinden. Beim Testen war zunächst keine Verbindung über Public Key Authentifizierung möglich, da, wie später herausgefunden wurde, der OpenSSH Daemon keinen Zugriff auf die authorized_keys hatte. Diese wurden mit dem Benutzer „tester" angelegt und hatten keine Zugriffsberechtigungen für den System-Account mit dem der OpenSSH Daemon ausgeführt wird. Nach hinzufügen der entsprechenden Berechtigungen war nun der Login via Public Key Authentifizierung möglich. Auch Up- sowie Downoads über SCP ließen sich so problemlos durchführen. Damit war mit dem OpenSSH Daemon eine geeignete Lösung für die Realisierung des SSH-Zugangs unter Windows gefunden.
Damit dem Entwickler bei der Implementierung der JUnit Tests unter Windows und Linux eine annähernd gleiche Umgebung zur Befehlsausführung geboten werden kann, sollten auch hier die wichtigsten Unix-Befehle zur Verfügung gestellt werden. Weiterhin nutzt die in Kapitel 4.3.3.1 beschriebene JSchCommandLineExecutor Bibliothek, die zur Befehlsausführung über eine SSH-Verbindung dient, Syntax und Befehle der Shell-Umgebung. Diese werden hier zur Vorbereitung der Ausführungsumgebung innerhalb einer SSH-Session benötigt. Zur Bereitstellung der wichtigsten Unix-Befehle wie beispielsweise wget, grep oder tar wurden die GNU Utilities for Windows in der Version 0.6.3 installiert. Die Tests bei der entsprechenden Befehlsausführung über SSH verliefen Problemlos, bis auf die Kommandos, die auch unter Windows bekannt sind. Hierzu zählt beispielsweise mkdir, bei dem dann stets das Windows-Kommando ausgeführt wird, wodurch es zu Problemen mit den angegebenen Parametern kommen kann. Um dies zu vermeiden ist in solchen Fällen der Absolute Pfad zum Executable des Unix-Befehls anzugeben.
Mit dieser Konfiguration kann auch die Windows XP VM für die automatisierten verteilten Tests eingesetzt werden.

4.2 Konfiguration von Netzwerk und Firewall

Um den im Testnetz, sowie VM-Netz befindlichen Systemen den Zugriff auf Dienste im DLR-Netz, bzw. im Internet zu gewähren, wurden zunächst entsprechende Regeln für den ausgehenden Verkehr an der Netzwerkschnittstelle lan-1 konfiguriert. Es wurden hier die Ports zum Beziehen einer IP per DHCP sowie für die Namensauflösung freigeschaltet. Weiterhin wurden Verbindungen zu den Diensten HTTP, HTTPS und FTP, für den Download von zu testender Software auf den Test-VMs zugelassen. Schließlich war es noch notwendig die Kommunikation vom Jenkins Build-Knoten zum Jenkins Server zuzulassen, damit dieser Regelmäßig abzuarbeitende Aufträge abfragen kann.

Um den Zugriff durch die Cloud Abstraction API auf die OpenStack Nova EC2 API aus dem DLR-Netz zu gewährleisten musste ein entsprechender Portforward eingerichtet werden. Weiterhin wurde eine Lösung benötigt, wie Systeme im VM-Netz, aus dem DLR-Netz über SSH erreicht werden können. Der Weg über die Firewall sieht zunächst einen Portforward auf die Floating IP am OpenStack System vor, welches dann über die selbst mit iptables verwalteten Portforwards auf die Ziel IP und Port im VM-Netz weiterleitet. In Abbildung 15 ist ein Beispiel der Kommunikation, zwischen der Komponente zur Befehlsausführung via SSH sowie SCP Up- und Downloads und dem SSH Daemon auf der Test-VM dargestellt. Als Ziel der Verbindung muss hier die externe IP der Firewall mit dem, für den Portforward eingerichteten TCP Port 22130 angegeben werden. Auf dem OpenStack wurde für die Ziel-VM die IP 10.0.0.3 im VM-Netz konfiguriert und die Floating IP 192.168.1.130 im Testnetz zugewiesen. Der Portforward auf der Firewall leitet dann die Anfrage an die Floating IP 192.168.1.130 mit Port 22 weiter. Über den, beim Zuweisen der Floating IP automatisch via iptables eingerichteten Portforward auf die IP 10.0.0.3, erreicht die Verbindungsanfrage ihr Ziel.
Um diese Konfiguration in einen automatisierten Vorgang zu überführen, würde eine Liste mit auf der Firewall konfigurierten Portmappings benötigt, die der Distributed Test API zur Verfügung gestellt wird. So ist dort die Information bekannt, an welchen Floating IPs, der SSH Daemon über welchen Port erreichbar ist. Zunächst erfolgte die Konfiguration der SSH-Portforwards für die Floating IPs 192.168.1.130 - 192.168.1.139 mit den Ports 22130 – 22130. Mit diesen Informationen konnte dann innerhalb der Distributed Test API, die Erreichbarkeit der SSH Daemon der Test-VMs, auch von Außerhalb des Testnetzes realisiert werden.

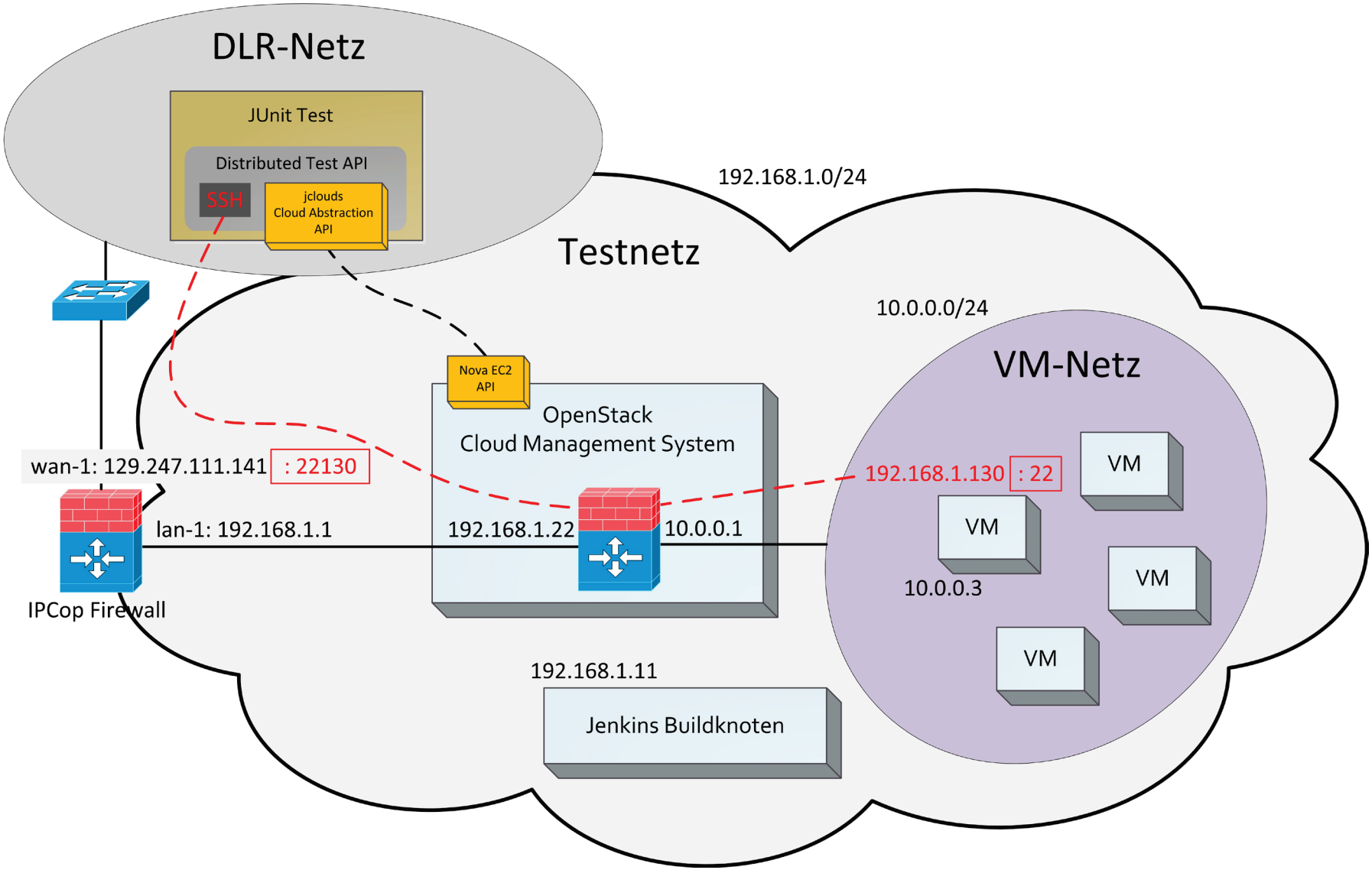

Abbildung 15: Konfiguration von Netzwerk und Firewall

4.3 Entwicklung der Distributed Test API

Die Distributed Test API enthält die Implementierung der Komponenten zur Steuerung der Cloud Infrastruktur sowie der durchzuführenden Tests.

4.3.1 Klassendiagramm

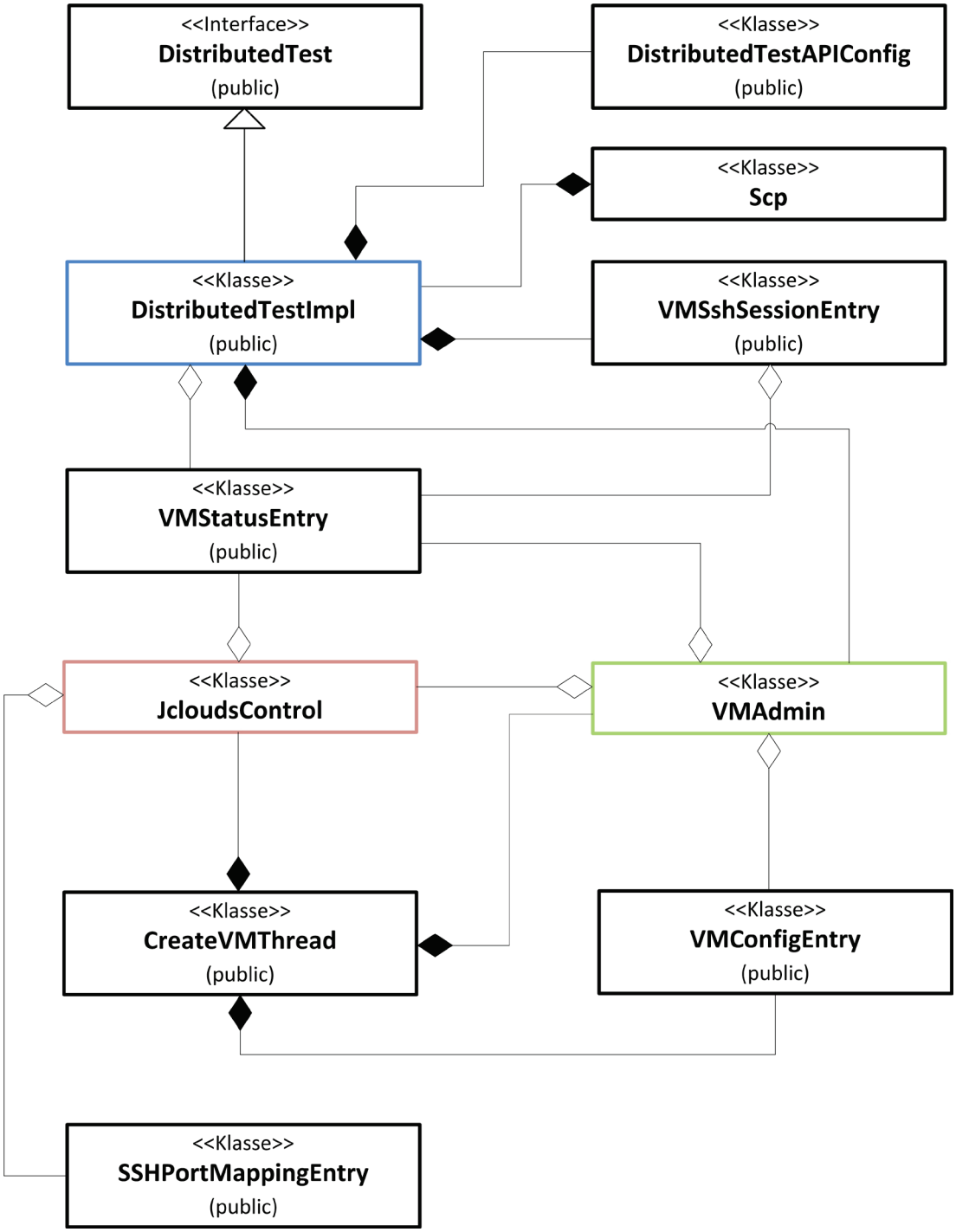

Abbildung 16: Klassendiagramm der Distributed Test API

Die Klassen JcloudsControl, VMAdmin und DistributedTestImpl stellen die Grundlegenden Funktionalitäten, zur Realisierung der im Konzept beschriebenen Vorgänge bereit. JcloudsControl steuert dabei mit Hilfe der OpenStack EC2 API die VM- und Netzverwaltung auf dem OpenStack. Die eigenständige Komponente VMAdmin dient zur Durchführung des VM Setup und nutzt die in JcloudsControl bereitgestellten Funktionen zur VM-Verwaltung. Dabei wird u.A. die, über die VM Config festgelegte Bereitstellung von entsprechend zu konfigurierenden VMs, Thread-gesteuert ausgeführt. Mit der Klasse DistributedTestAPIConfig wird die Objektstruktur zum einlesen der konstanten Konfigurationsparameter, wie Dateipfaden, IPs und URLs aus der Datei DistributedTestAPIConfig.json festgelegt. DistributedTestImpl stellt die API zur Implementierung von verteilten Tests durch den Entwickler bereit. Hier werden die im Interface DistributedTest deklarierten Funktionen zur Befehlsausführung über SSH sowie SCP Up- und Downloads implementiert. Für diese Operationen können allerdings nicht die Funktionen von jclouds genutzt werden, da diese ausschließlich die im OpenStack hinterlegten IPs, für den SSH-Zugriff verwenden kann. Da hier allerdings die Verbindung über den Portforward an der externen IP der Firewall erfolgen muss, können die vorhandenen IPs nicht verwendet werden. JcloudsControl wird daher eine Liste der Portmappings mit Floating IP und externem SSH-Port zur Verfügung gestellt, um die SSH Sessions den korrekten VMs zuzuweisen. Dabei wird zur Sicherstellung eines korrekten Testablaufs der Zustand, der über die Klasse DistributedTestImpl erzeugten SSH-Sessions verwaltet. Für eine koordinierte Zugriffsteuerung auf die VMs, wird weiterhin der Zustand der VMs, in einer Statusliste gepflegt, die allen Komponenten der Distributed Test API zur Verfügung steht. Die einzulesenden Konfigurationsdateien werden im Format der JavaScript Object Notation (JSON) abgelegt, die ein Standard zur Strukturierung Textbasierter Daten darstellt. Diese wurde zwar im Rahmen von JavaScript entwickelt, kann aber in zahlreichen Programmiersprachen wie u.A. C, C++ und Java über entsprechende Bibliotheken genutzt werden. Es können verschiedene Datentypen wie beispielsweise Number, String oder Boolean verwendet werden, um die Werte innerhalb der Textdatei zu verarbeiten. Weiterhin gibt es die Möglichkeit zur Strukturierung der Werte in Objekten oder Arrays. Zum Einsatz in der Distributed Test API wurde der Jackson Java JSON-processor verwendet, um das Lesen und Schreiben der JSON-Dateien innerhalb der Klasse VMAdmin zu realisieren. Dieser wird dort zum Serialisieren und Deserialisieren der Objekte aus den Konfigurationsdateien über einen ObjectMapper verwendet.
Die verwendeten json-Dateien, sowie für die Public Key Authentifizierung zu verwendenden Private Keys werden im entwickelten Package de.dlr.sc.utils.testing.distributed unter src/main/resources abgelegt. Diese können so dem Entwickler bei der Verwendung der Klasse DistributedTestImpl, bzw. dem Jenkins Build-Knoten, bei der Ausführung des VMAdmin über das SVN Repository zur Verfügung gestellt werden.

4.3.2 Steuerung der VM- und Netzkonfiguration

4.3.2.1 VMConfigEntry

Die Klasse VMConfigEntry definiert das Format des einzulesenden VM Config Objekts aus der, durch den Entwickler zu editierenden Konfigurationsdatei VMConfig.json. Diese stellt die Benutzerschnittstelle zur Eingabe der gewünschten VM-Konfigurationen dar und wird bei der Durchführung des VM Setup in der Klasse VMAdmin eingelesen. Mit Hilfe der hier definierten Setter Methoden realisiert der ObjectMapper des Jackson JSON Parser, die Serialisierung der Werte aus der Konfigurationsdatei in VMConfigEntry-Objekte. Das Attribut vmId gewährleistet die eindeutige Identifikation der VM innerhalb der Distributed Test API. Mit der Definition der group lassen sich bei der Testausführung über die Klasse DistributedTestImpl, gleiche Operationen, die auf mehreren VMs auszuführen sind, in einem Methodenaufruf erledigen. Über die imageId wird das für die VM zu verwendende OpenStack VM Image angegeben. Das Attribut ramInMB lässt den angegebenen Wert zur Arbeitsspeicherzuteilung der VM auf dem ESXi, über den OpenStack konfigurieren.

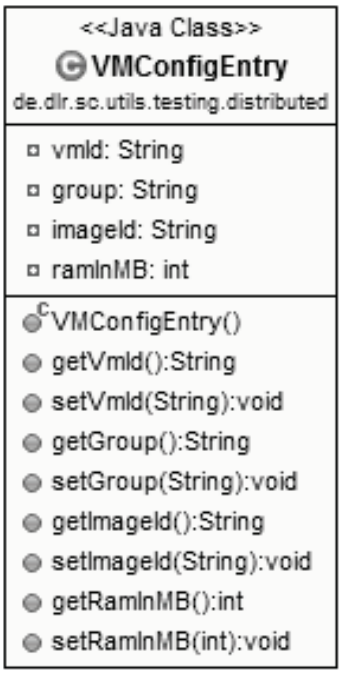

Abbildung 17: Klasse VMConfigEntry

4.3.2.2 VMStatusEntry

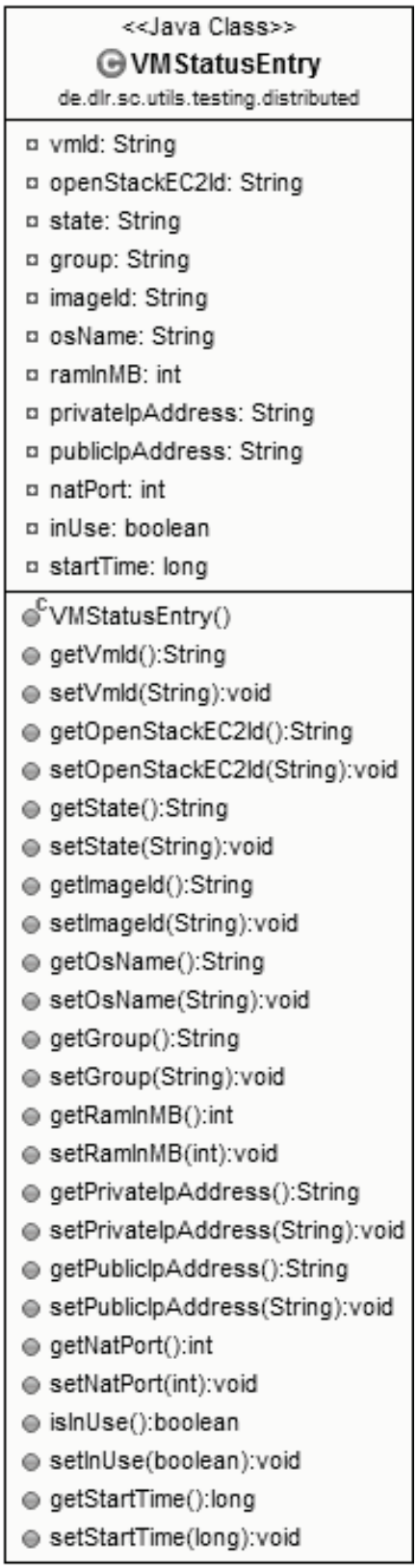

Abbildung 18: Klasse VMStatusEntry

Zur Koordination der Zugriffssteuerung auf die VMs, durch die einzelnen Komponenten der Distributed Test API, wurde eine Statusliste eingeführt. Diese ermöglicht, beim Setup durch die Klasse VMAdmin sowie zur Testausführung über die Klasse DistributedTestImpl, die Bereitstellung des Zustands der VM.
Auch hier werden mit Hilfe der Setter und Getter Methoden, die Serialisierung sowie Deserialisierung der Objekte aus der Status-Datei, über den ObjectMapper des Jackson JSON Parser ermöglicht. Weiterhin erfolgt das Festhalten aller Informationen einer VM, die für die Durchführung der verteilten Tests relevant sind. Die Behandlung des VM Status wird dann in den verwendenden Klassen näher erläutert.
Die OpenStackEC2Id repräsentiert die ID der VM im OpenStack System, die bei der Erstellung der VM zugewiesen wird. Um eine VM innerhalb der Distributed Test API eindeutig zu identifizieren, auch wenn diese gerade (noch) nicht auf dem OpenStack, bzw. ESXi existiert, wurde die vmId eingeführt. Weiterhin ermöglicht dies dem Entwickler, eine eigene Bezeichnung zur besseren Beschreibung bzw. Wiedererkennung der VM zu wählen.
Über das Attribut state können die Zustände ausgelesen werden, welche die OpenStack Nova EC2 API für eine VM vorsieht. Dieser nimmt den Zustand PENDING an, wenn die VM gerade durch den OpenStack bzw. ESXi bearbeitet wird (u.A. beim Starten und Stoppen). Während des Löschens der VM vom ESXi, befindet diese sich im Zustand TERMINATED. Weiterhin ist hier der Zustand SUSPENDED vorgesehen, der allerdings bei Verwendung des ESXi 5, beim Ausführen der entsprechenden suspend-Funktion dazu führt, dass die VM vom ESXi5 entfernt wird. Beim Aufruf der resume-Funktion erfolgt dann ein erneutes Importieren des VM-Images vom OpenStack. Seitens OpenStack wurde angegeben, dass dies in der verwendeten OpenStack Diablo Version in Verbindung mit dem ESXi Hypervisor nicht korrekt funktioniert, bei der Verwendung von XEN oder KVM als Hypervisor allerdings keine Probleme auftreten würden. Wenn die VM korrekt durch den ESXi gestartet werden konnte, geht diese in den Zustand RUNNING über. Beim Auftreten von Fehlern tritt der Zustand ERROR ein und falls dieser nicht ermittelt werden kann, wird er auf UNRECOGNIZED gesetzt.
Das Attribut osName beschreibt die, beim Import des VM-Images auf dem OpenStack zugewiesene Bezeichnung. Die privateIPAddress gibt die IP der VM im VM-Netz und publicIpAddress, die Floating IP im Test-Netz an. Der natPort legt den für die SSH-Verbingungen auf dieser VM zu verwendenden Port an der Floating IP fest. Schließlich beschreibt noch das Attribut inUse, ob sich eine VM gerade in Verwendung, durch eine Funktion der Klasse DistributedTestImpl (während einer Testdurchführung), befindet.
Damit die Statusdatei den unabhängig ausgeführten Komponenten DistributedTestImpl und VMAdmin zur Verfügung steht, erfolgt die Speicherung im Dateisystem des OpenStack. Bei jeder, die VMs betreffende, ausgeführte Aktion durch diese Klassen wird die Statusdatei mit den Änderungen aktualisiert. Die Struktur der Statusdatei entspricht dem in Kapitel 4.3.2.1 dargestellten Beispiel der VMConfig.json.

4.3.2.3 SSHPortMappingEntry

Diese Klasse beschreibt das SSHPortMappingEntry-Objekt zum einlesen der Einträge aus der Datei SSHPortMapping.json. Diese Datei enthält je Objekt eine Zuordnung des Attributs ipAddress, das mit einer Floating IP belegt ist und dem natPort, der den entsprechenden SSH-Port spezifiziert. Die auf der Firewall konfigurierte Zuordnung von Floating IP und SSH-Port wurde in Kapitel 4.2 beschrieben.

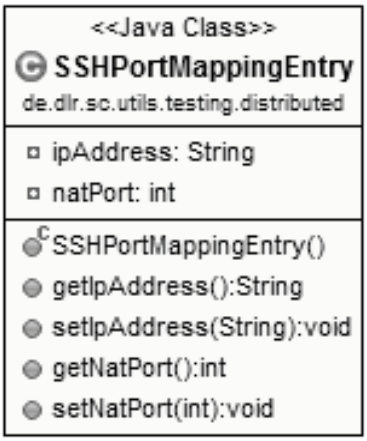

Abbildung 19: Klasse SSHPortMappingEntry

4.3.2.4 JcloudsControl

Die Klasse JcloudsControl stellt die Schnittstelle zur Steuerung der VM- und Netzverwaltung auf dem OpenStack dar. Die Anbindung der Distributed Test API an das OpenStack System erfolgt hier über die Cloud Abstraction API jclouds. In dieser werden die abstrahierten Methoden zum Ansprechen der OpenStack Nova EC2 API verwendet.
Zunächst wird beim Erzeugen eines JcloudsControl Objekts über den Konstruktor, stets ein Verbindungskontext zur Nova EC2 API, mit Hilfe der Funktion initConnection() aufgebaut. Alle weiteren jclouds-Methodenaufrufe werden dann über das hierbei initialisierte ComputeServiceContext-Objekt context abgewickelt. Dieses kann mit Hilfe der Methode getContext() abgefragt werden.
Über die Funktion createVM() wird die im VMConfigEntry-Objekt übergebene VM angelegt. Dabei werden die entsprechenden Parameter, zusammengefasst in einem jclouds-templateBuilder-Objekt an die Methode createNodesInGroup() zur Erstellung der VM weitergereicht. Diese weist der VM noch eine OpenStack-interne Gruppe zu, die in der globalen Variablen group hinterlegt ist. Diese hat für die VM-Verwaltung innerhalb der Distributed Test API keine Bedeutung.
Während des Erstellungsprozesses der VM, wird automatisch durch den OpenStack, eine IP-Adresse aus dem Adressbereich des VM-Netzes und eine aus dem freigegebenen Bereich der Floating IPs zugewiesen. Hierbei tritt das Prob-

lem auf, dass beide IP-Adressen durch jclouds im VM-Objekt der Liste der Privaten IP-Adressen zugewiesen werden. Daher muss beim Übertragen der IP-Adressen in die VMStatus-Datei zunächst eine Unterscheidung anhand der Zuweisungsreihenfolge, in private und öffentliche IP-Adresse erfolgen. Im letzten Parameter bekommt der createVM-Aufruf noch die Information übermittelt, ob für die VM bereits ein Statuseintrag existiert, oder nicht. Entsprechend wird dann der Eintrag mit den entsprechenden VM-Daten neu angelegt oder aktualisiert.

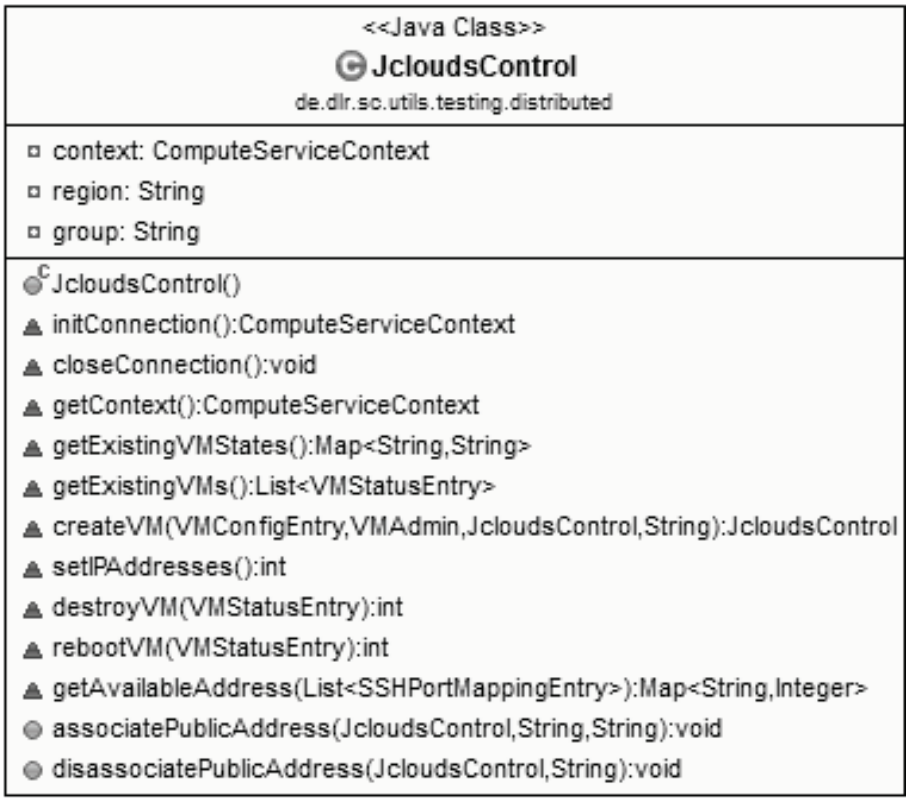

Abbildung 20: Klasse JcloudsControl

Über die Methode setIPAddresses erfolgt anschließend die Zuweisung eines freien Floating IP – SSH-Port Paares aus der Datei SSHPortMapping.json. Dabei wird zunächst, im Falle einer automatisch zugewiesenen Floating IP, aus der Zuweisung zur VM gelöst. Dies ist über die Methode disassociatePublicAddress() gelöst, welche hierzu eine Methode der OpenStack Nova EC2 proprietären API nutzt. Nachdem ein freies Floating IP – SSH-Port Paar über die Funktion getAvailableAddress() ermittelt wurde, kann zunächst die Zuweisung der entsprechenden Floating IP, über die Methode associatePublicAddress() erfolgen.
Diese verwendet hierfür ebenfalls eine Funktion der OpenStack Nova EC2 proprietären API. Zusammen mit dem SSH-Port wird die Floating IP anschließend zum Eintrag der entsprechenden VM in der Status-Datei hinzugefügt. Die feien Floating IP – SSH-Port Paare werden über das Abgleichen der VMStatus-List und der SSHPortMapping-Liste ermittelt.

Über die Methode getExistingVMStates kann eine Liste mit den aktuellen Zuständen, der über den OpenStack konfigurierten VMs abgefragt werden. Die Methode getExistingVMs passt die Einträge im VMStatus, entsprechend der aktuell auf der OpenStack bzw. ESXi vorliegenden VM-Konfiguration an. Die Methode rebootVM dient zum Neustart einer VM und destroyVM entfernt diese vom ESXi und OpenStack. Beim reboot ist zu beachten, dass dieser ein hartes Aus- und Anschalten der VM bewirkt, da auf dem VM-Image keine VMware Tools installiert sind, die einen Zugriff auf die Betriebssystemfunktion shutdown ermöglichen. Über den Aufruf closeConnection() wird der Verbindungskontext zwischen jclouds und der OpenStack Nova EC2 API geschlossen.

4.3.2.5 VMAdmin

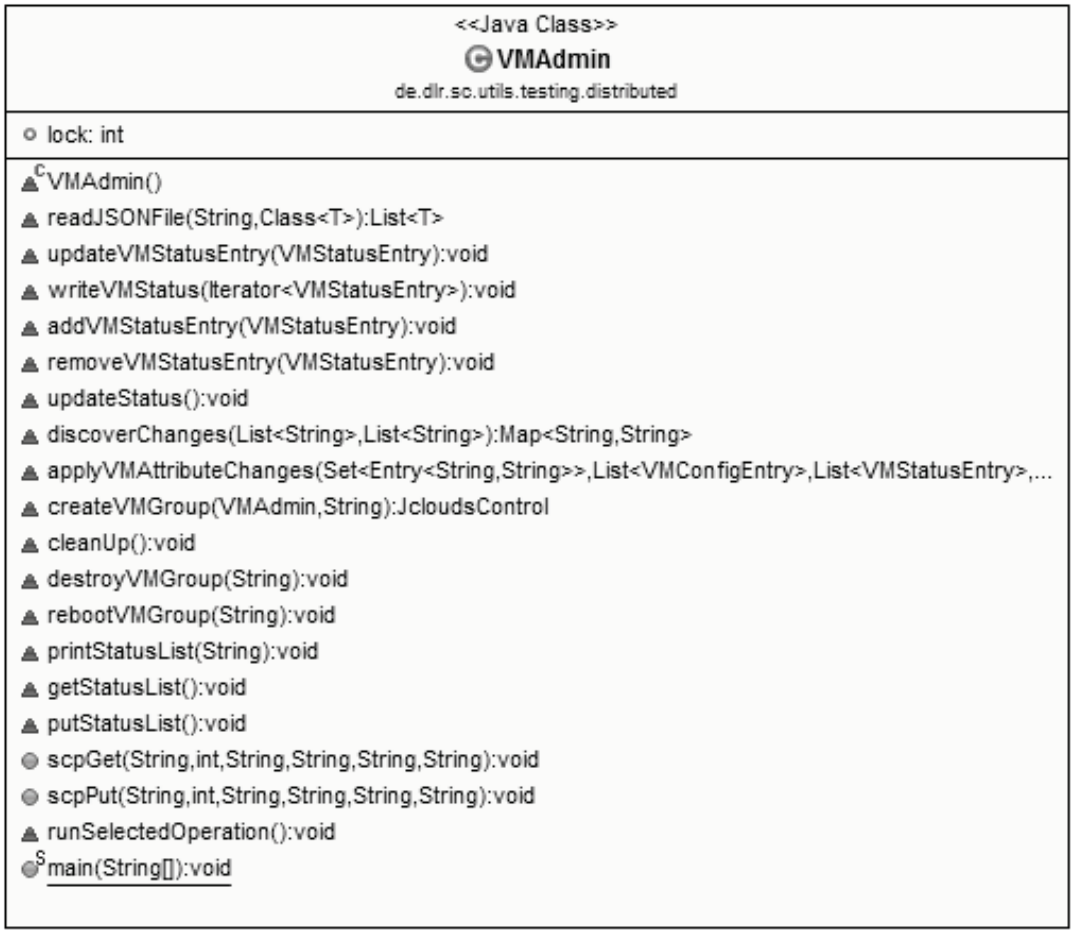

Abbildung 21: Klasse VMAdmin

Die Klasse VMAdmin dient zur Initialisierung und Steuerung des VM Setup. Beim Aufruf werden, wie in Kapitel 3.9.1 beschrieben, durch den Jenkins-Build-Job angegebene Umgebungsvariablen ausgewertet, um dann die entsprechende Operation auf eine Gruppe von VMs anzuwenden. Für die Umgebungsvariable OPERATION können die Werte start, stop, reboot, cleanup und status ausgewertet und die entsprechenden Methoden aufgerufen werden. Zur Ausführung der

Operation wird den entsprechenden Methoden noch der Wert der Umgebungsvariablen VM_GROUP übergeben.
Über die Operation start wird das Anlegen neuer VMs initialisiert und die Methode createVMGroup() aufgerufen. Hier wird zunächst die VM Config eingelesen und überprüft, ob alle Werte der anzulegenden VMs korrekt angegeben wurden (vmId, group vorhanden? Doppele vmIds?). Anschließend erfolgt über die Methode discoverChanges, der Abgleich zwischen der Liste vorhandener VMs aus der VMStatus-Datei und der VM Config. Der Rückgabewert liefert eine Map mit vmIds und einem zugeordneten Marker, der die Werte NEW, REMOVED oder EXISTING annehmen kann. Besitzt dieser den Wert EXISTING und es haben sich die VM Attribute RAM oder imageId geändert, erfolgen für die VMs über die Methode applyVMAttributeChanges() dieselben Operationen wie beim manuellen Aufruf von start und stop einer VM-Gruppe. Dabei werden die geänderten Werte für die entsprechenden VMs übernommen.
Besitzt der Marker den Wert NEW, erfolgt das neue Anlegen der VM über den Thread CreateVMThread. Das Anlegen der VMs einer Gruppe wird auf mehrere Threads verteilt, um die Zeit des Erstellungsprozesses effektiver Nutzen zu können. Hier dauert beispielsweise das Anlegen von drei VMs parallel nur ca. 70% länger als eine VM. Des Weiteren können so parallel auch andere Operationen des VMAdmin verwendet werden. Beim Starten des Threads wird über die globale Variable lock ein neuer Thread registriert. Über diese wird in einem Intervall von 5 Sekunden überprüft, ob bereits alle Threads abgearbeitet sind, und anschließend, der für die VM-Erstellung benötigte Verbindungskontext (context) der Klasse JcloudsControl, geschlossen. Anschließend erfolgt noch das Setzen, des für den SSH-Zugriff notwendigen, korrekten Floating IP – SSH-Port Paares für die neu angelegten VMs, durch die JcloudsControl-Methode setIPAddresses().
Besitzt die Umgebungsvariable den Wert stop, wird die Methode destroyVMGroup() aufgerufen und die entsprechenden VMs über Aufrufe der JcloudsControl-Methode destroyVM() vom OpenStack sowie ESXi entfernt. Weiterhin wird die VM auch aus der VMStatus-Datei entfernt. Über den Wert reboot erfolgt der Aufruf der Methode rebootVMGroup() und die entsprechenden VMs werden über Aufrufe der JcloudsControl-Methode rebootVM(), durch den OpenStack auf dem ESXi neugestartet. Ist der Umgebungsvariable der Wert cleanup zugewiesen, wird die gleichnamige Methode aufgerufen und zunächst über die JcloudsControl-Methode getExistingVMStates(), ein Set mit vmIds und zugeordnetem Status ermittelt. Verwaiste VMs, die nicht in der VMStatus-Datei aufzufinden sind und sich auch nicht im Zustand PENDING (siehe Kapitel 4.3.2.2) befinden werden gelöscht.
Im Anschluss an jede dieser Operationen erfolgt das Aktualisieren der VMStatus-Datei über die Funktion updateStatus(). Dabei werden zunächst die auf dem OpenStack exisitierenden VMs mit den, in der VMStatus-Datei Eingetragenen, abgeglichen. Hier erfolgt wieder eine Überprüfung der vorliegenden Änderungen über die Methode discoverChanges(). Daraufhin werden im Fall von, auf dem OpenStack geänderten oder gelöschten VMs, die entsprechenden Einträge in der

VMStatus-Datei angepasst. Die VMStatus-Datei ist, wie in Kapitel 4.3.2.2 beschrieben, zentral auf dem OpenStack abgelegt. Die Änderungsvorgänge bezüglich des Status werden stets erst auf dem lokalen System, auf dem der VMAdmin betrieben wird, übernommen. Hierbei kommen die Methoden addVMStatusEntry, updateVMStatusEntry und removeVMStatusEntry zum Einsatz. Zum Ausführen dieser Aktionen wird jeweils die VMStatusDatei über die Funktion readJSONFile() eingelesen, die Änderungen an der Liste von VMStatusEntry-Objekten übernommen und mit der Funktion writeVMStatus() eine neue VMStatus-Datei geschrieben. Über die Methoden putStatusList() und getStatusList() erfolgt das Importieren bzw. Exportieren mit der VMStatus-Datei im Dateisystem des OpenStack. Mit den darin enthaltenen Funktionsaufrufen von scpGet() bzw. scpPut() wird dann die VMStatus-Datei mit Hilfe der SCP-Funktion des JSchCommandLineExecutor übertragen. So wird die VMStatus-Datei nach jeder Änderung in das Dateisystem des OpenStack kopiert.
Die Operation status führt zum Aufruf der Methode printStatusList(). Besitzt die Umgebungsvariable VM_GROUP den Wert all, werden alle Einträge der VMStatus-Datei ausgegeben. In allen anderen Fällen werden nur die, mit der entsprechend eingetragenen Gruppenbezeichnung dargestellt.
Es kann vorkommen, dass beim Start einer VM, der Erstellungsprozess auf dem OpenStack hängen bleibt. Dies kann daran erkannt werden, das bei einem normalen Verlauf einer zu erstellenden VM, nach spätestens 2 Minuten eine IP-Adresse aus dem VM-Netz zugewiesen wird. Ist dies nicht der Fall, kann davon ausgegangen werden, dass der Erstellungsprozess hängt. Mit Hilfe dieser Erkenntnis wurde versucht, in diesem Fall die VM zu entfernen und nochmal neu anzulegen. Da diese Lösung allerdings noch nicht in 100% der Fälle funktioniert und damit noch nicht sicher eingesetzt werden konnte, muss das Entfernen derzeit noch manuell über die OpenStack Konsole erfolgen.

4.3.2.6 CreateVMThread

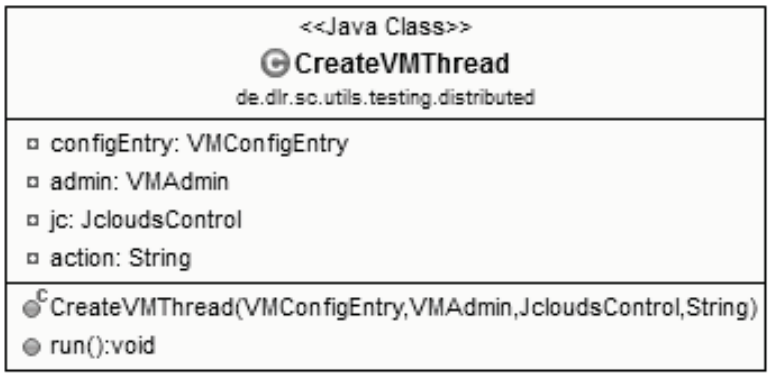

Abbildung 22: Klasse CreateVMThread

Die von Thread abgeleitete Klasse dient zur parallelen Ausführung der JcloudsControl-Methode createVM(). Nachdem diese abgearbeitet ist, wird die

globale Variable lock der Klasse VMAdmin um den abgearbeiteten CreateVMThread reduziert.

4.3.3 API zur Erstellung verteilter Tests

4.3.3.1 VMSshSessionEntry

Die Klasse VMSshSessionEntry definiert die Objekte einer Liste, welche die relevanten Daten, der zu den VMs aufgebauten SSH-Sessions beinhaltet. Diese wird in der Klasse DistributedTestImpl eingesetzt, um während der Befehlsausführung verwaiste SSH-Sessions zu beenden. Die Befehlsausführung und SCP-Funktionalitäten werden über eine, in der Abteilung SC entwickelten Bibliothek bereitgestellt. Diese verwendet die java SSH-Schnittstelle Jsch zur Realisierung der entsprechenden Funktionen. Die Attribute sshSession, executor und sshLogger werden mit Objekten aus dieser Bibliothek belegt. In der sshSession sind dabei die Daten der zugehörigen Session gespeichert. Die Variable sshLogger beschreibt den, zur Initialisierung der Sessions notwendigen Apache-Commons-Logger und der JSchCommandLineExecutor executor liefert die Informationen zur, in der Session durchgeführten Befehlsausführungen bzw. SCP-Operationen. Das Attribut running dient zur Beschreibung des Zustands der Session.

Abbildung 23: Klasse VMSshSessionEntry

4.3.3.2 DistributedTestImpl und Scp

Die Klasse DistributedTestImpl stellt die API zur Realisierung von verteilten Tests durch den Entwickler bereit. Hier werden die im Interface DistributedTest deklarierten Funktionen implementiert.
Im Konstruktor werden zunächst die aus der DistributedTestAPI Config benötigten Informationen ausgelesen und der Klasse zur Verfügung gestellt. Weiterhin erfolgt das Herunterladen der VMStatus-Datei über die Methoden der Klasse VMAdmin. Um Befehle über eine SSH-Session auf einer einzelnen VM ausführen zu können, wurde die Methode runScriptOnVM() zur Verfügung gestellt. Als Übergabewerte werden eine vmId sowie die mit einem Linebreak getrennten Befehle in Form eines Strings erwartet. So ist es auch möglich, ein in String konvertiertes Shell-Skript zu übergeben.

Abbildung 24: Klasse DistributedTestImpl

Zunächst erfolgt über die Methode checkVMIsInUse() die Kontrolle, ob die entsprechende VM evtl. gerade durch einen anderen Test in Verwendung ist. Dies

wird anhand der Variable isInUse des VMStatus bestimmt. Ist dieser Wert auf false gesetzt, erfolgt nun die Änderung auf true, um keinem anderen Vorgang den Zugriff auf diese VM zu gewähren. Anschließend erfolgt der Aufbau der SSH-Session über die Methode createVMSshSession(). Auch hierbei werden Methoden, der zur Verfügung gestellten Bibliothek zur Bereitstellung der SSH-Operationen, genutzt. Zum Aufbau der SSH-Verbindung werden die externe IP der Firewall, zusammen mit dem entsprechenden SSH-Port, des zugehörigen VMStatusEntry sowie der Benutzername tester und dessen Private Key angegeben. Über die aufgebaute Session erfolgt dann die Initialisierung der, aus der zur Verfügung gestellten Bibliothek zur Bereitstellung der SSH-Operationen stammenden Klasse JSchCommandLineExecutor. Diese Bietet Methoden zur Befehlsausführung (ein- und mehrzeilig) sowie SCP Up- und Downloads an. Über einen InputStream wird es ermöglicht die Konsoleausgabe des StdOut einer Befehlsausführung zu erhalten.
Nachdem in der Methode runScriptOnVM() dann der, an die Session gebundene JSchCommandLineExecutor zur Verfügung steht erfolgt die Ausführung der übergebenen Befehle. Zur Auswertung der Ausführung wird anschließend der JSchCommandLineExecutor zurückgeliefert.
Die Methode createCommandLineExecutorForGroup() ermöglicht die Erzeugung von JSchCommandLineExecutor für die VMs einer, als Übergabewert anzugebenden Gruppe. Der Ablauf ist hier zunächst mit runScriptOnVM() identisch. Es wird dann für jede VM der Gruppe, eine SSH-Session Aufgebaut und der JSchCommandLineExecutor erzeugt. Über eine Map erfolgt die Rückgabe, der jeweils einer vmId zugeordneten JSchCommandLineExecutor. Für die Befehlsausführung auf mehreren VMs wird in der Regel eine parallelisierte Abarbeitung benötigt. Um diese für den jeweiligen Test optimal zu verwalten, erfolgt hier nicht direkt die Skriptausführung. Der Testschreiber erhält so die Möglichkeit diese selbst zu verwalten.
Der SCP Upload ist über die Funktion scpToVM() und der Download über die Funktion scpFromVM() realisiert. Hier wird zunächst auch wie in der Methode runScriptOnVM() verfahren, um einen JSchCommandLineExecutor, über die SSH-Session zur entsprechenden VM zu initialisieren. Der eigentliche SCP-Vorgang wird dann im Thread Scp ausgeführt um eine Parallelisierung der Kopiervorgänge zu ermöglichen.

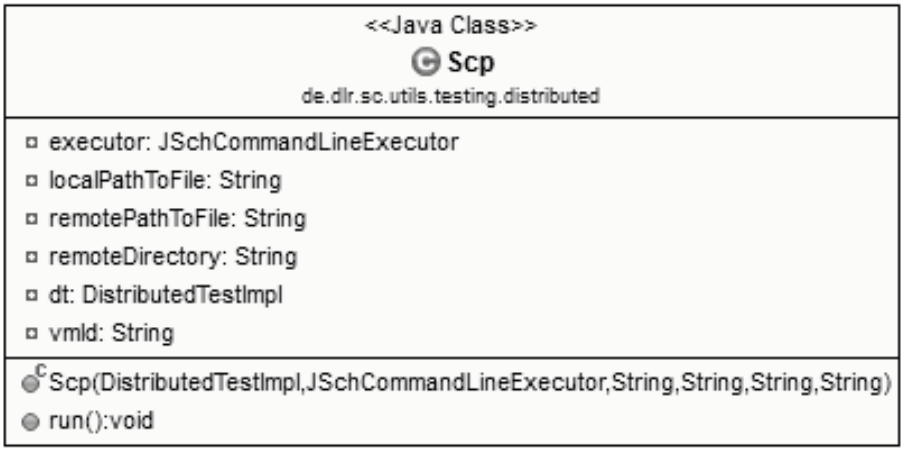

Abbildung 25: Klasse Scp

Anhand der Übergabeparameter erfolgt dort dann die Unterscheidung, ob ein Up- oder Download durchgeführt werden soll. Mit Hilfe des zuvor initialisierten JSchCommandLineExecutor, wird das Kopieren der angegebenen Datei aus dem lokalen Dateisystem, in ein Verzeichnis auf dem entfernten System sowie vice versa möglich. Beim SCP Download von der VM wird dem Dateinamen vor dem Schreiben in das lokale Dateisystem noch die „Private IP" der VM angehangen.
Um einen SCP Up- bzw. Download auf mehrere VMs vorzunehmen, können die Methoden scpToGroup() und scpFromGroup eingesetzt werden. Hierbei erfolgt der Aufruf der Methoden scpToVM(), bzw. scpFromVM() für jede VM, der als Übergabeparameter angegebenen Gruppe.
Die Möglichkeit zur Bereitstellung der Attribute aus der VMStatus-Datei ist über die Funktion getVMAttributesByGroup() gegeben.
Die Verwaltung der SSH-Sessions wird über die Methode closeSessions() bereitgestellt. Hierzu werden in der globalen Variable sessionList alle aufgebauten SSH-Sessions über VMSshSessionEntry-Objekte festgehalten. Es wird hier nun alle zwei Sekunden überprüft, ob noch laufende SSH-Sessions vorhanden sind. Besteht im Falle einer laufenden Ausführung, einer SCP Operation noch die entsprechende SSH-Session, wird gewartet bis das Attribut isRunning der sessionList auf false steht. Dies geschieht nach Abarbeitung einer SCP Operation durch den Thread Scp. Besitzt isRunning den Wert true, wird die entsprechende SSH-Session beendet.

4.3.4 Systemtests der Distributed Test API

4.3.4.1 Test der Funktionalitäten der Klasse VMAdmin

Um die korrekte Funktionsweise der im Rahmen des VM Setup durchgeführten Operationen zu gewährleisten, werden nachfolgende Systemtests durchgeführt.
Da die Komponente des VMAdmin derzeit noch nicht auf dem Jenkins Build-Knoten läuft, erfolgen die Tests aus Eclipse heraus. Diese werden über eine 64Bit JRE 1.6.0_31, auf einem Windows 7 SP1 System im DLR-Netz ausgeführt. Die Umgebungsvariablen werden über die entsprechende Run Configuration zur Verfügung gestellt.

4.3.4.1.1 Operation start, status

In der Datei src/main/resources/VMConfig.json wird zunächst folgende VM Config vorgenommen.

```
{
"vmId" : "abcd",
"group" : "gruppe2",
"imageId" : "nova/ami-00000002",
"ramInMB" : "1740"
}
{
"vmId" : "12",
"group" : "gruppe3",
"imageId" : "nova/ami-00000002",
"ramInMB" : "1740"
}
{
"vmId" : "34",
"group" : "gruppe3",
"imageId" : "nova/ami-00000002",
"ramInMB" : "1740"
}
```

Abbildung 26: VM Config

Werden die Umgebungsvariablen OPERATION=start und VM_GROUP=gruppe3 gesetzt erfolgt zunächst der Start der in beiden, in gruppe3 befindlichen VMs. Da zunächst keine vmIds angegeben wurden, bricht der Start mit der entsprechenden Fehlermeldung ab. Erfolgt die Ausführung des VMAdmin auf dem Jenkins Build-Knoten, sind diese Meldungen über das Anzeigen der Konsolenausgaben erhältlich. Nach einer erneuten Initiierung des Starts beginnt nun das Anlegen der VMs auf dem ESXi. Eine mit ca. 2 Minuten Verzögerung, parallel ausgeführte Statusausgabe über ein Starten des VMAdmin mit den Umgebungsvariablen OPERATION=status und VM_GROUP=all erzeugt die in Abbildung 27 gezeigte Ausgabe. Die vmId, group und natPort sind während des Anlegens der VM noch

null, da diese erst anschließend aus der VM Config, bzw. aus dem SSHPortMapping übertragen werden. Der state PENDING weist darauf hin, dass die VMs gerade durch den ESXi in Bearbeitung sind. Das Attribut publicIpAddress ist ebenfalls noch null, da die Floating IP auch erst anschließend konfiguriert wird. Nach der erfolgreich durchgeführten VM-Erstellung und dem weiteren Anlegen der VM abcd ergibt sich die in Abbildung 28 dargestellte Ausgabe. Das Anlegen der VMs aus gruppe3 war nach ca. 17min. abgeschlossen. Nun sind vmId und group zugewiesen. Weiterhin wurden jeweils erfolgreich ein korrektes Paar Floating IP – SSH-Port konfiguriert.

```
getting StatusList ...


vmId: null
openStackEC2Id: nova/i-000000a9
state: PENDING
group: null
imageId: nova/ami-00000002
osName: Ubuntu-12.04-Server-x86
ramInMB: 1740
privateIpAddress: 10.0.0.2
publicIpAddress: null
natPort: 0
inUse: false


vmId: null
openStackEC2Id: nova/i-000000a8
state: PENDING
group: null
imageId: nova/ami-00000002
osName: Ubuntu-12.04-Server-x86
ramInMB: 1740
privateIpAddress: 10.0.0.3
publicIpAddress: null
natPort: 0
inUse: false
```

Abbildung 27: Ausgabe der Statusliste während VM-Erstellung

Die Attribute state besitzten den Wert RUNNING und zeigen damit an, das die VMs betriebsbereit sind.

```
vmId: 12
openStackEC2Id: nova/i-000000a9
state: RUNNING
group: gruppe3
imageId: nova/ami-00000002
osName: Ubuntu-12.04-Server-x86
ramInMB: 1740
privateIpAddress: 10.0.0.2
publicIpAddress: 192.168.1.131
natPort: 22131
inUse: false
```

```
vmId: 34
openStackEC2Id: nova/i-000000a8
state: RUNNING
group: gruppe3
imageId: nova/ami-00000002
osName: Ubuntu-12.04-Server-x86
ramInMB: 1740
privateIpAddress: 10.0.0.3
publicIpAddress: 192.168.1.130
natPort: 22130
inUse: false


vmId: abcd
openStackEC2Id: nova/i-00000097
state: RUNNING
group: gruppe2
imageId: nova/ami-00000002
osName: Ubuntu-12.04-Server-x86
ramInMB: 1740
privateIpAddress: 10.0.0.4
publicIpAddress: 192.168.1.132
natPort: 22132
inUse: false
```

Abbildung 28: Ausgabe der Statusliste nach VM-Erstellung

Der Login über SSH, mit dem Benutzername tester und dem entsprechenden Private Key konnte auf den VMs über 129.247.111.141:22130, 129.247.111.141:22131 und 129.247.111.141:22132 erfolgreich getestet werden. Mittels ifconfig war es hier auch jeweils möglich, die „Privaten IPs" erfolgreich zu verifiziert werden.

4.3.4.1.2 Operation reboot

Durch Setzen der Umgebungsvariablen OPERATION=reboot und VM_GROUP= gruppe2 mit anschließender Ausführung des VMAdmin, wurde der Neustart der VM abcd initialisiert. Nach ca. 5 Sekunden konnte auf dem ESXi das erneute Hochfahren der VM beobachtet werden. Der Zustand REBOOT wird nicht vom OpenStack gepflegt und das Attribut state bleibt daher, bei einer Ausgabe der Statusliste auf RUNNING.

4.3.4.1.3 Operation cleanup

Durch das Anlegen einer VM über die OpenStack Konsole wurde zunächst eine verwaiste VM simuliert. Diese zeichnet sich dadurch aus, dass sie im OpenStack, aber nicht in der Statusliste auftaucht. Durch Setzen der Umgebungsvariablen OPERATION=cleanup mit anschließender Ausführung des VMAdmin, wurde diese VM erfolgreich entfernt.

4.3.4.1.4 Operation destroy

Um die VM 12 aus der gruppe3 zu löschen, wurde dieser in der VM Config zunächst das Attribut group auf den Wert gruppe4 geändert. Durch Setzen der Umgebungsvariablen OPERATION=reboot und VM_GROUP=gruppe4 mit anschließender Ausführung des VMAdmin, erfolgte das Entfernen der VM. Es konnte erfolgreich verifiziert werden, dass die VM vom ESXi und OpenStack entfernt und aus der Statusliste gelöscht wurde.

4.3.4.2 Test der Funktionalitäten der Klasse DistributedTestImpl

Um die öffentlichen Methoden der Klasse DistributedTestImpl zu testen wurde eine Klasse DistributedTestTest geschrieben und in zwei Funktionen Beispiele für Skriptausführung und SCP Operationen implementiert. Da hierbei das Testen der Funktionalitäten im Vordergrund steht und keine Testlogik mit entsprechender Testfall-Bewertung benötigt wird, erfolgte die Realisierung in einer Java Klasse.

4.3.4.2.1 comparePrivateIPs()

Der erste Test dient zum Vergleich der über das OpenStack System ermittelten „Privaten IP" einer VM, mit der tatsächlich auf dem System Vorliegenden. Zunächst wurde über die Methode createCommandLineExecutorForGroup("gruppe3") die Map, mit der Zuordnung von vmIds und JSchCommandLineExecutor bezogen. Anschließend erfolgte iterativ über den jeweiligen JSchCommandLineExecutor der VMs, die Befehlsausführung eines ifconfig-Kommandos.
Der Methodenaufruf getVMAttributesByGroup("gruppe3") lieferte die Map, mit der Zuordnung von vmIds und VMStatusEntry-Objekten der VMs mit dieser Gruppenzugehörigkeit.
Schließlich konnte über die Ausgabe der InputStreams der Ifconfig-Befehlsausführung und den aus den VMStatusEntry-Objekten ermittelten „Privaten IPs" die Äquivalenz der Einträge aus verschiedenen Systemen verifizieren.
Des Weiteren weist dies die korrekte Funktionsfähigkeit der verwendeten Methoden nach.

4.3.4.2.2 RCEwfExecution()

Die Ausführung von Workflows zur automatisierten Durchführung von Berechnungen stellt eine der zentralen Funktionen von RCE dar. Dies über die Distributed Test API durchzuführen, ist Aufgabe der Funktion RCEwfExecution(). Der nachfolgend beschriebene Test wurde auf der VM mit der vmId 12 durchgeführt.
Zunächst erfolgte über die Methode runScriptOnVM() das Herunterladen und Entpacken der Headless RCE Version Standard-2.3.2-linux.gtk.x86, welche ohne GUI auf der Konsole ausführbar ist. Ein Aufruf der Methode scpToVM() kopierte

die Workflowkonfiguration in das root-Verzeichnis von RCE, welches im Rahmen der Skriptausführung im home-Verzeichnis des Benutzers tester angelegt wurde.
Anschließend erfolgte nochmals eine Skriptausführung über die Methode runcScriptOnVM(), um die Ausführungsrechte des rce-Executables zu setzen, die RCE-Workflow-Ausführung zu starten und schließlich um nach der Ausführung, die während der Workflow-Ausführung generierten Log-Dateien in ein komprimiertes tar-Archiv zu packen.
Durch zwei scpFromVM()-Aufrufe wurden das rce.log sowie die gepackten Workflow-Logs von der VM in das lokale Dateisystem heruntergeladen.
Während der Ausführung des Tests wurde über den VMAdmin die Statusliste abgefragt und das inUse-Attribut der VM 12 kontrolliert. Dieses war korrekt auf true gesetzt. In dieser Zeit war es auch nicht möglich, den Test noch einmal zu starten, da die entsprechende VM bereits belegt war. Ein Entfernen der VM 12 über die stop-Operation des VMAdmin war ebenfalls nicht möglich. Nach Abschluss der Testausführung zeigte das inUse-Attribut, ebenfalls richtig, false an.
Anhand der zurückgelieferten Log-Dateien konnte die korrekte Ausführung des RCE-Workflow verifiziert werden. Des Weiteren wurden die Vorgänge über eine SSH-Konsole mit dem ps-Befehl überwacht. Die Vorgänge mit nicht allzu kurzer Abarbeitungsdauer ließen sich so zusätzlich erfolgreich bestätigen.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Ziel der Arbeit war es, eine Java API für automatisierte Tests von verteilt laufendem Java-Code zu entwickeln und dabei die Dienste einer Cloud, zur Bereitstellung der benötigten Rechenressourcen zu nutzen.
Um ein Verständnis für die eingesetzten Cloud Technologien zu vermitteln wurden zunächst entsprechende Grundlagen auf diesem Gebiet behandelt. Weiterhin galt es, dass als zentrales Testobjekt verwendete verteilte System RCE näher zu erläutern. Vor dem Beginn der Konzeptionierung wurden aus gegebenen Anwendungsszenarien für die automatisierten, verteilten Tests Anforderungen ermittelt, die in Allen Phasen der Arbeit zu berücksichtigen waren. Als weiteren Input für das zu erstellende Konzept, erfolgte Anschließend die Evaluierung von Public Compute Cloud Systemen, um eine Übersicht von Funktionalitäten und Preisen zu erhalten. Über eine Bewertung konnte festgestellt werden, welches System sich am ehesten, für die Verwendung in der geplanten Implementierung eignet. Zur Evaluierung von Cloud Management Systemen erfolgte zunächst die Vorbereitung der Testumgebung in Form der entsprechenden ESXi-Konfiguration. Weiterhin mussten verschiedene Rahmenbedingungen bzgl. Der Netzwerkanbindung geklärt werden. Über die Evaluierung der Cloud Management Systeme kristallisierte sich das Potential des OpenStack Systems, als geeignete Komponente zur VM-Verwaltung heraus. In der nachfolgenden Untersuchung wurde jclouds als zu Verwendende Cloud Abstraction API bestimmt. Nachdem noch Pisces als geeignetes Framework zur Realisierung der verteilten JUnit Tests ermittelt wurde, konnte die Konzeptionierung beginnen. Hier erfolgte, in vier Phasen des Testablaufs unterteilt, die Kombination von verschiedenen Verfahren zur Cloud- und Test-Steuerung. Die Anschließende Implementierung des Konzepts sah zunächst vor, VM-Images, Netzwerk und Firewall zu konfigurieren, um die infrastrukturellen Voraussetzungen zu gewährleisten. Über die initiale Entwicklung eines Klassendiagramms erfolgte die Festlegung der Struktur der Distributed Test API. Es wurden dann schrittweise die Komponenten zur Steuerung der VM- und Netzkonfiguration implementiert. Weiterhin folgte die Entwicklung der API zur Erstellung verteilter Tests. Abschließend wurden noch Systemtests, der durch den Testentwickler zugreifbaren Klassen VMAdmin und DistributedTestImpl durchgeführt.

5.2 Ausblick

In der momentanen Implementierung konnten noch keine Subnets, zur Ausführung erweiterter Netzwerktests realisiert werden. Dies wäre zum einen über die Konfiguration von VLANs oder die Verwendung der Multinic-Funktionalität auf dem OpenStack umsetzbar. Hierzu wäre es auch empfehlenswert, ein Update auf das aktuelle OpenStack Release Folsom durchzuführen, um eine unkompliziertere Konfiguration zu ermöglichen. Dies könnte auch helfen, das u.U. vorkommende Aufhängen der VM-Erstellung auf dem OpenStack zu beheben. Weiterhin wären Tests mit, im OpenStack hinzugefügten IPv6 Netz möglich. Des Weiteren wurde derzeit die Windows VM nur für den Einsatz in der DistributedTestAPI vorbereitet. Hier wäre eine Einbindung in die API, durch Anpassung der Methoden der Klasse JSchCommandLineExecutor wünschenswert.

Literaturverzeichnis

[1] Mell, P.; Grance, T.: The NIST Definition of Cloud Computing. Gaithersburg 2011
http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[2] Badger, L.; Grance, T.: Cloud Computing Synopsis and Recommendations. Gaithersburg 2012
http://csrc.nist.gov/publications/drafts/800-146/Draft-NIST-SP800-146.pdf

[3] Amazon Web Services (Ed.): Amazon Virtual Private Cloud - User Guide. Seattle 2012
http://awsdocs.s3.amazonaws.com/VPC/latest/vpc-ug.pdf

[4] Amazon Web Services (Ed.): Amazon Elastic Compute Cloud - User Guide.
Seattle 2012
http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-ug.pdf

[5] Rackspace (Ed.): Cloud Servers Developer Guide. San Antonio 2012
http://docs.rackspace.com/servers/api/v1.0/cs-devguide-20120921.pdf

[6] OpenStack: Conceptual Architecute – OpenStack Compute Administration Manual – Essex (2012.1) [Online; Stand 20.10.2012]
http://docs.openstack.org/essex/openstack-compute/admin/content/conceptual-architecture.html

[7] OpenStack (Ed.): OpenStack Compute Starter Guide. 2011
http://docs.openstack.org/diablo/openstack-compute/starter/openstack-starter-guide-diablo.pdf

[8] OpenStack (Ed.): OpenStack Compute Administration Manual. 2011
http://docs.openstack.org/diablo/openstack-compute/admin/os-compute-adminguide-diablo.pdf

[9] Krämer-Fuhrmann, O.; Klein, J.: RCE - Remote Component Environment.
Sankt Augustin 2009
https://wiki.sistec.dlr.de/RCE?action=AttachFile&do=get&target=NAFEMSMagazinNovember2009.pdf

[10] C12G Labs (Ed.): Designing and Installing your Cloud Infrastructure OpenNebula 3.2. 2011
http://www.c12g.com/Portals/155284/docs/ONE3.2%20-%20Designing%20and%20Installing%20your%20Cloud%20Infrastructure.pdf

[11] C12G Labs (Ed.): Designing and Installing your Cloud Infrastructure OpenNebula 3.6. 2012
http://www.c12g.com/Portals/155284/docs/ONE3.6%20-%20Designing%20and%20Installing%20your%20Cloud%20Infrastructure.pdf

[12] Apache: Libcloud Documentation [Online; Stand 20.12.2011]
http://libcloud.apache.org/docs/

[13] Apache: Libcloud, Supported Providers and Features [Online; Stand 20.12.2011]
http://libcloud.apache.org/supported_providers.html

[14] Apache: Deltacloud API [Online; Stand 18.12.2011]
http://deltacloud.apache.org/

[15] Igugu, J., O.; Biltoria, P: STAF-on-Eucalyptus: A Cloud Based Software Testing Environment for Distributed Systems. Göteborg 2011
https://gupea.ub.gu.se/bitstream/2077/27858/1/gupea_2077_27858_1.pdf

[16] Rentschler, M.; Zobel, J.: Verteilte Testautomatisierung mit STAF/STAX. Sindelfingen 2010
http://wwwlehre.dhbw-stuttgart.de/~rentschler/Publications/ESE_2010_Artikel_STAF_STAX.pdf

[17] IBM: STAF V3 User's Guide [Online; Stand 18.12.2011]
http://staf.sourceforge.net/current/STAFUG.htm