Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3101

Migration virtueller Knoten in einer zeitvirtualisierten Emulationsumgebung

Sebastian Bartmann

Studiengang:	Informatik
Prüfer:	Prof. Dr. Kurt Rothermel
Betreuer:	Dipl.-Inf. Andreas Grau
begonnen am:	27. Juli 2010
beendet am:	26. Januar 2010
CR-Klassifikation:	H.2.4, I.6.7, C.4

Zusammenfassung

Ziel der Diplomarbeit ist die Erweiterung einer zeitvirtualisierten Emulationsumgebung namens TVEE (Time Virtualized Emulation Environment) um eine Möglichkeit zur dynamischen Neuplatzierung virtueller Knoten. TVEE wurde für Test und Evaluation verteilter Software und Netzwerkprotokolle entwickelt. Techniken wie Knoten und Zeitvirtualisierung ermöglichen eine Evalutation von Testszenarien mit tausenden von Knoten. Durch Knotenvirtualisierung wird die Ausnutzung bestehender Hardwareressourcen, durch Ausführung mehrerer Software Instanzen auf einem physikalischen Knoten, maximiert. Werden mehr Ressourcen benötigt als vorhanden, können diese mittels Zeitvirtualisierung auf Kosten der Experimentlaufzeit künstlich erhöht werden. Für die Akzeptanz eines Testsystems muss sich die Ausführungszeit eines Experiments in einem vertretbaren Rahmen bewegen. Für die Reduktion der Experimentlaufzeit werden aktuell in TVEE zwei Ansätze verfolgt: eine laufzeitoptimale initiale Platzierung und eine adaptive Anpassung der virtuellen Zeit. Durch Lastschwankungen oder falsche Annahmen kann die ermittelte initiale Platzierung allerdings suboptimal sein. Eine Anpassung der Platzierung virtueller Knoten während eines Experiments kann daher sinnvoll sein. Sind physikalische Knoten unterschiedlich stark ausgelastet kann durch Migration einzelner virtueller Knoten eine gleichmäßige Auslastung der Testumgebung erreicht werden. Ressourcenengpesse einzelner Rechner können dadurch vermieden werden. Jede Migration von Knoten ist allerdings mit Kosten verbunden.

In dieser Diplomarbeit werden zunächst Mechanismen zur transparenten Migration von virtuellen Knoten erarbeitet. Darauf aufbauend wird ein Modell vorgestellt mit dem sich erwartete Migrationskosten voraussagen lassen. Dann werden verschiedene Ansätze zur Optimierung einer Platzierung während eines Experiments vorgestellt und gegeneinander abgewogen. In Simulationen großer Szenarien zeigte sich, dass sich durch dynamische Neuplatzierung, die Experimentlaufzeit maßgeblich senken lässt.

Abstract

In this diploma thesis a time virtualized emulation environment called TVEE is extended by a technique named dynamic replacement of virtual nodes. TVEE was developed for test and evaluation of distributed software and network protocols. Techniques like node and time virtualization, allow test szenarios with thousands of nodes. Through node virtualization the hardware utilization is maximized by parallel execution of software instances on the same physical node. If more ressources are needed as provided, time virtualization is used to virtually increase ressources by slowing down the realtime by a factor called time dilation factor(TDF). For the acceptance of an emulation system the runtime of an experiment has to be short. To reduce the execution time TVEE currently uses two approaches: an adaptive virtual time and a runtime optimal initial placement. Due to load variation and wrong assumptions the initial placement can be suboptimal. Therefore an adaption of the placement during the experiment can be useful. In case of an unequal load of physical nodes of the system the load can be balanced by migrating several virtual nodes. Thereby ressource bottle necks can be avoided. However each migration causes costs that have to be considered.

First in this diploma thesis concepts for the transparent virtual node migration in TVEE are developed. Then a model for the prediction of migration costs is presented. After that different approaches to optimize a current placement are presented and discussed. Simulations of large szenarios show that the experiment runtime can be greatly reduced by dynamic replacement.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1 Einleitung

1.1 Motivation

Testen und Evaluieren sind wichtige Schritte in der Entwicklung neuer verteilter Anwendungen und Netzwerkprotokolle. Durch die Komplexität heutiger Anwendungen können diese beiden Schritte zu einer herausfordernden Aufgabe werden. Peer-to-Peer Anwendungen wie z.B. Gnutella [Gnu] bestehen meist aus tausenden von Knoten, die sich in einem Verbund aus heterogenen Netzen befinden. Nicht selten nehmen deswegen Test und Evaluierung einen großen Teil der Enwicklungszeit in Anspruch.

Aus der Literatur sind im wesentlich 3 Techniken für das Testen und Evaluieren neuer verteilter Anwendungen bekannt: Live Testing [CCR+03], Netzwerk Simulation [BTA+, Kes88, Ril03] und Netzwerk Emulation [CS03, Hem05, NSNK97].

Beim Live Testing wird die Software vor der Auslieferung unverändert unter realen oder fast realen Bedingungen getestet. Da realistische Testszenarien meist mehrere tausend Knoten umfassen, ist der Aufbau einer Live-Testumgebung allerdings mit sehr hohen Kosten verbunden. Zudem sind je nach eingesetzter Technologie z.B. Funk-LANs Messergebnisse nicht reproduzierbar. Außerdem ist die Testumgebung auf bereits existierende Technologien beschränkt. Live Testing ist also nur bedingt geeignet für das Testen und Evaluieren neuer verteilter Anwendungen.

Netzwerk Simulation [GRL05] stellt im Gegensatz zum Live Testing eine kostengünstige und kontrollierbare Alternative dar, die zu reproduzierbaren Messergebnissen führt. Bei der Simulation wird das reale Netzwerk auf ein parametrisierbares Modell abgebildet. Dieses Modell kann einen beliebigen Grad an Abstraktion aufweisen, weshalb Messergebnisse in Hinblick auf die reale Welt mit Vorsicht zu genießen sind. Meist ist außerdem eine Reimplementierung der zu testenden Software nötig.

Die dritte in der Literatur aufgeführte Technik ist die Netzwerk Emulation [GRL05]. Sie stellt einen hybriden Ansatz aus Live Testing und Simulation dar und vereint Vorteile beider Ansätze. In der Emulation werden reale Elemente, wie Ziel-Hosts und Protokolle, mit künstlichen, simulierten oder abstrahierten Elementen,wie Netzwerkverbindungen und Hintergrundverkehr, verbunden. Dadurch entsteht ein synthetische, parametrisierbare Netzwerkumgebung, in der Eigenschaften wie z.B. Bandbreite, Verzögerung und Verlustrate

festgelegt werden können. Dies ermöglicht die Erzeugung von reproduzierbare Messergergebnisse. Im Gegensatz zur Simulation muss hierbei die zu testende Software nicht angepasst werden. Getestet wird bei der Emulation meist in Realzeit [FTO, MI]. Die Emulation vereint Vorteile des Livetestings und der Simulation: Realitätsnähe und Kontrollierbarkeit .

In vielen Emulationsumgebungen wird pro Kommunikationsknoten ein physikalischer Computer eingesetzt. Realistische Testszenarien umfassen allerdings meist tausende Kommunikationsknoten, was einen hohen Hardwareaufwand bedeutet. Für große Szenarien werden daher besser skalierbare Ansätze benötigt. Aus der Literatur sind zwei Techniken bekannt, die eine skalierbare Emulationsumgebung ermöglichen: Knoten- und Zeitvirtualisierung.

Bei der Knotenvirtualisierung [AH06] werden mehrere Instanzen der zu testenden Software auf einem physikalischen Rechner genannt *physical Node*(PNode) ausgeführt. Jede Instanz wird dabei in ihrer eigenen Ausführungsumgebung, genannt *virtual node*(VNode), gestartet. Über diesen Ansatz kann eine bessere Ausnutzung der zu Verfügung stehenden Hardware gewährleistet werden. Werden mehrere physikalische Rechner z.B. über ein LAN verbunden ermöglicht der Knotenvirtualisierungsansatz die Segmentierung einer Netzwerk Topologie durch die Verteilung der einzelnen VNodes auf die vorhandenen physikalischen Rechner. Eine Emulationsumgebung, die aus mehreren physikalischen Rechnern besteht, wird verteilte Emulationsumgebung genannt. Durch Knotenvirtualisierung lässt sich die Größe eines Testszenarios, also die Anzahl der Knoten, massiv steigern. Durch den in [MHR] vorgestellten Knotenvirtualisierungsansatz gelang es z.B. Testszenarien um das 28 fache zu vergrößern. Die Testumgebung bestand aus einem Pentium 4 (2,4 Ghz) Cluster. Allerdings ist die Anzahl virtueller Knoten durch die zur Verfügung stehenden Ressourcen wie CPU, Speicher und Netzwerkbandbreite begrenzt.

Um die Anzahl der Knoten pro Test bei gleicher Hardware noch weiter erhöhen zu können , wird eine Technik namens Zeitvirtualisierung [GYM+06] eingesetzt. Dabei wird die Wahrnehmung der Zeit eines Betriebssystems und aller darin befindlichen Anwendungen verändert. Mittels eines Faktors, genannt *time dilation factor* (TDF), wird die Zeit verlangsamt. Das Betriebssystem nimmt dabei die real vergehende Zeit kürzer wahr. Dadurch stehen pro Zeiteinheit mehr Ressourcen zur Verfügung, was eine weitere Erhöhung der Knotenanzahl möglich macht. Diese Technik geht allerdings zu Lasten der Experimentlaufzeit. Sie erhöht sich proportional zum TDF.

Um die Experimentlaufzeit möglichst in einem vertretbaren Rahmen zu halten, können verschiedene Techniken eingesetzt werden. Sind mittlere Datenraten sowie mittlere Auslastungen der Knoten bekannt, kann ein Platzierungsalgorithmus [GHR] verwendet werden. Dieser bildet vor dem Test virtuelle Knoten eines Testszenarios auf physikalische Knoten derart ab, dass die erwartete Experimentlaufzeit minimal ist.

Eine andere Technik besteht in der adaptiven Anpassung des TDF Faktors [GHR09] an die aktuelle Last. Ist das System überlastet so wird der TDF Faktor erhöht. Befindet sich das System in einem Zustand mit ungenutzten Ressourcen wird der TDF Faktor erniedrigt.

Auf diese Weise kann eine hohe Auslastung der Testumgebung über das ganze Experiment erreicht werden.

Um keinen Knoten zu überlasten orientiert sich die Last des Systems an dem maximal ausgelasteten physikalischen Knoten. Dadurch kann es, trotz adaptiver TDF Anpassung, zu einer schlechten Auslastung einzelner Rechner kommen. Dies ist der Fall, wenn Knoten sehr unterschiedliche ausgelastet sind. In einer solchen Situation kann es sinnvoll sein, während des Experiments bestimmte virtuelle Knoten zu migrieren, z.B. virtuelle Knoten des am stärksten ausgelasteten physikalischen Knotens zu einem weniger ausgelasteten. Dadurch sinkt die Auslastung des Systems und der TDF Faktor kann erniedrigt werden, was zu einer Verkürzung der Experimentlaufzeit führt. Bei einer Migration entstehen allerdings Kosten, die berücksichtigt werden müssen.

1.2 Ziel der Arbeit

Im Rahmen dieser Diplomarbeit soll eine bestehende Emulationsumgebung, die Techniken wie Knoten- und Zeitvirtualisierung einsetzt, um die Möglichkeit zur dynamischen Neuplatzierung virtueller Knoten erweitert werden.

Bei unausgeglichener Last einzelner Rechner der Testumgebung sollen virtuelle Knoten von überlasteten Rechnern auf weniger ausgelastete Rechner verschoben werden. Dazu werden zunächst Mechanismen benötigt, die eine Migration virtueller Knoten erlauben. Diese müssen z.B. die Sicherung des Zustands, den Transfer und die Wiederherstellung des Zustands eines Knotens umfassen. Zudem gilt es, die Netzwerktopologie des Testszenarios an die neue Situation anzupassen. Um Messergebnisse nicht zu verfälschen, muss dies in einem sehr kurzen Zeitfenster geschehen.

Des weiteren wird ein Algorithmus benötigt, der bei ungünstiger Lastsituation, eine neue Plazierung virtueller Knoten bestimmt. Eine neue Platzierung wird dabei durch Migration einzelner Knoten erreicht. Durch die Migration entstehen Kosten, die bei der Auswahl einer neuen Platzierung zu berücksichtigen sind.

Zur Beurteilung unterschiedlicher Platzierungen ist es nötig, diese vorab abschätzen zu können. Es wird daher ein Modell gebraucht, mit dem sich Migrationskosten voraussagen lassen. Migrationskosten können als Investition angesehen werden, die sich nur lohnen, wenn die erwartete Kosteinsparungen höher sind als die Investition.

1.3 Outline

Die folgenden Kapitel der Diplomarbeit sind folgendermaßen gegliedert.

Im Kapitel 2 wird eine zeitvirtualisierte Emulationsumgebung namens TVEE (Time Virtualized Emulation Environment) vorgestellt. Diese bildet die Basis dieser Diplomarbeit.

Im darauf folgendem Kapitel 3 wird ein der dynamischen Neuplatzierung von virtuellen Knoten sehr ähnliches Problem präsentiert: Taskmigration.

Kapitel 4 geht, dann näher auf die dynamische Neuplatzierung von virtuellen Knoten ein. Unter anderem werden in diesem Kapitel unterschiedliche Lösungsansätze für Teilprobleme, wie die Optimierung einer aktuellen Platzierung, oder die Umsetzung einer Platzierung, vorgestellt und gegeneinander abgewogen.

In Kapitel 5 wird näher auf einen Prototype eingegangen, in dem Lösungsansätze und vorgestellte Konzepte umsetzt wurden.

Dieser Prototype wird im folgenden Kapitel 6 evaluiert.

Im letzten Kapitel werden schließlich Resultate zusammengefasst und auf offene Probleme eingegangen.

2 Zeitvirtualisierte Emulationsumgebung

Im Rahmen des NET (Network Emulation Testbed) [NP] Projekts wurde eine Emulationsumgebung namens Time Virtualized Emulation Environment (TVEE) zum Testen verteilter Anwendungen entwickelt. Sie besteht aktuell aus einem PC Cluster mit 20 Dual Core Xeons (2.13 Ghz) mit jeweils 24 GB Ram. Jeder Rechner des Clusters ist mittels Ethernet Karten an 2 Netzwerke angebunden: dem Kontrollnetzwerk und dem Emulationsnetzwerk. Kontroll- und Testdatenverkehr sind damit voneinander getrennt.

In TVEE kommen Techniken wie Knotenvirtualisierung und Zeitvirtualisierung zum Einsatz. Diese machen das System skalierbarer und ermöglichen die Emulation von Testszenarien mit tausenden von Knoten [GMHR08].

2.1 Architektur

Knoten- und Zeitvirtualisierung wird in TVEE durch einen geschachtelten Virtualisierungsansatz erreicht. Dieser ist in Abbildung 2.1 dargestellt.

Auf jedem physikalischen Knoten (pNode) , wird pro CPU eine virtuelle Maschine (VM) gestartet. Diese stellt eine virtuelle Zeit bereit, welche für das Betriebssystem innerhalb der virtuellen Maschine völlig transparent ist. Ressourcen der VM werden mittels Virtual Routing und Space Partitioning aufgeteilt (siehe Abschnitt 2.2). Pro Software Under Test(SuT) Instanz wird ein virtueller Knoten (vNode) innerhalb der VM erzeugt.

Damit beliebige SuT Instanzen miteinander kommunizieren können, werden Software Brücken eingesetzt. Diese verbinden virtuelle Netzwerkkarten der virtuellen Knoten, bzw. der virtuellen Maschinen miteinander. Um eine Kommunikation über die Grenzen eines Rechners hinweg zu ermöglichen, werden zudem Software Brücken an physikalische Netzwerkkarten angebunden.

Im folgenden wird nun näher auf die Umsetzung der Netzwerk Emulation, Knoten- und Zeitvirtualisierung eingegangen werden.

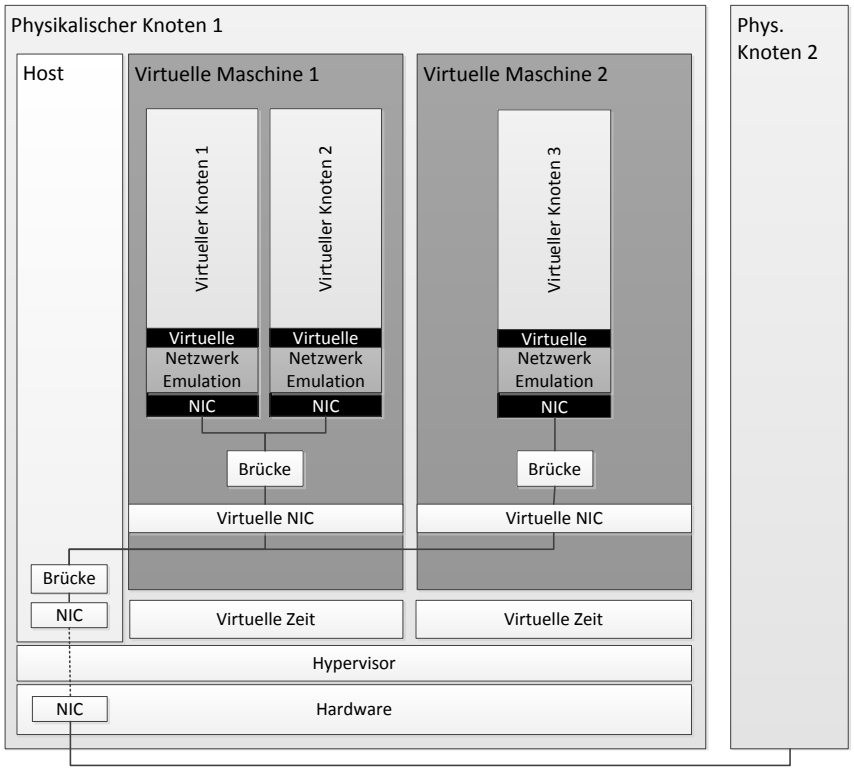


Abbildung 2.1: TVEE Architektur

2.2 Knoten Virtualisierung

Knotenvirtualisierung ermöglicht die Ausführung mehrerer zu testender Software Instanzen auf einem physikalischen Knoten. Jede Software Instanz wird dabei in ihrer eigenen virtuellen Umgebung ausgeführt. Aus der Literatur sind mehrere Virtualisierungsansätze bekannt.

Einen möglichen Ansatz stellt die Nutzung von virtuellen Maschinen dar. Dabei wird die zu testende Software in einem Betriebssystem ausgeführt, das keinen direkten Zugriff auf die Hardware hat. Der Zugriff erfolgt stattdessen über eine Software, die zwischen Hardware und Betriebssystem eingefügt wird. Diese wird *virtual machine monitor* (VMM) genannt. Sie koordiniert den Zugriff unterschiedlicher virtueller Maschinen und den darin befindlichen Betriebsystemen auf die Hardware. Jede SuT Instanz wird bei diesem Ansatz in einem eigenen Betriebsystem ausgeführt.

Da bei diesem Ansatz Betriebsysteme unverändert benutzt werden können, ist die Virtualisierung transparent für die zu testende Software. Allerdings sind für die Kommunikation zwischen Software Instanzen in unterschiedlichen virtuellen Maschinen teure Kontextwechsel nötig, was zu einem großen Virtualisierungsoverhead führt.

Einen günstigeren Ansatz stellt die Nutzung eines virtuellen Netzwerkstacks dar. Bei diesem Ansatz wird nur der Netzwerkstack virtualisiert. Im Gegensatz zum virtuellen Maschinen Ansatz werden alle SuT Instanzen im gleichen Betriebssystem ausgeführt.

SuT Instanzen können bei diesem Ansatz in sogenannten virtuellen Knoten voneinander separiert werden. Ein virtueller Knoten umfasst dabei

- eine bestimmte Anzahl von Prozessen (z.B. die SuT)
- einen Netzwerkstack, der mit Prozessen des virtuellen Knotens verbunden ist

Im Gegensatz zum virtuellen Maschinenansatz ist der Speicheroverhead deutlich geringer. Außerdem werden keine teuren Kontextwechsel für die Kommunikation zwischen SuT Instanzen benötigt. Allerdings ist die Nutzung von virtuellen Netzwerkstacks nicht so transparent wie der virtuelle Maschinen Ansatz.

In TVEE wurde sich für den Netzwerk Stack Ansatz entschieden. Dazu kommt OpenVZ zum Einsatz.

OpenVz ist ein leichtgewichtiges Virtualisierungssystem, das unabhängige, sicher und isolierte Container (virtuelle Knoten) auf einer physikalischen Maschine bereitstellt. Neben Netzwerkstack Virtualisierung wird in OpenVZ noch *Space Partitioning* eingesetzt. Dadurch erscheint jeder Container als einzelner Host mit eigenen Usern, autonomem Dateisystem und Speicher, unabhängiger Ip Adresse und eigenen Anwendungen.

Laut [GMHR08] stellt OpenVZ eine Virtualisierungslösung mit sehr geringem Speicheroverhead dar. Für jeden Container (VNode) werden nur zusätzlich 300 kbyte Speicher benötigt. Dies ermöglicht die Ausführung tausender virtueller Knoten auf einem einzelnen physikalischen Knoten.

2.3 Zeit Virtualisierung

Zeitvirtualisierung stellt einen Ansatz dar, mit dessen Hilfe sich zur Verfügung stehende Ressourcen künstlich erhöhen lassen. Sie kann unterschiedlich umgesetzt werden(siehe [GYM+06]). In TVEE wird eine angepasste virtuelle Maschine verwendet, um virtuelle Zeit bereitzustellen.

Aktuell kommt in TVEE Xen [xen] als Hypervisor zum Einsatz. Xen nutzt eine Technik namens Paravirtualisierung. Bei dieser Technik wird die Hardware nicht emuliert, sondern den virtuellen Maschinen wird ein direkter Zugriff auf vorhandene Ressourcen ermöglicht. Daher ist Xen effizient, erfordert aber eine Portierung der Betriebssysteme (üblicherweise Gastsysteme genannt), die in den virtuellen Maschinen laufen sollen.

In Xen werden Gastsysteme als Domains bezeichnet. Unter den Domains nimmt die Domain mit dem Index 0 (*dom*0) eine besondere Rolle ein. Diese wird beim Booten gestartet und

besitzt spezielle Rechte, wie z.B. die Befugnis zum Starten und Verwalten anderer Domains, meist domU genannt. Zur effizienten Unterstützung von Multicoresystemen wird in TVEE aktuell auf jedem physikalischen Knoten eine domo und gleich viele domUs wie verfügbare CPU gestartet([GHR]).

Standardmäßig unterstützt Xen keine virtuelle Zeit. Daher wurde die Schnittstelle zwischen Hypervisor und den virtuellen Maschinen um eine Funktion für die Einstellung der virtuellen Zeit erweitert. Über einen *time dilation factor* (TDF) kann die Geschwindigkeit, mit der die virtuelle Zeit voranschreitet, gesteuert werden.

2.4 Netzwerk Emulation

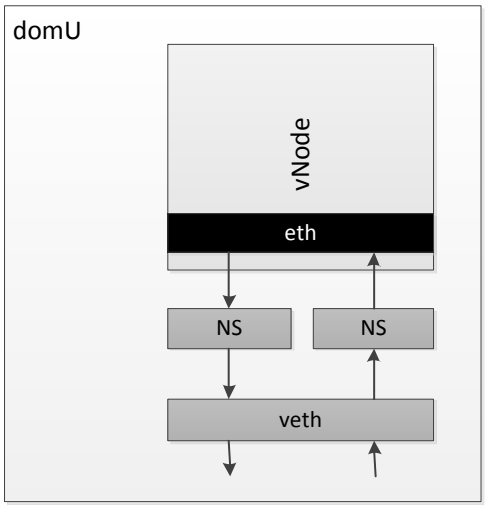


Abbildung 2.2: Netshaper

Netzwerk Emulation ermöglicht die Schaffung einer künstlichen Netwerkumgebung. In TVEE erfolgt die Umsetzung der Netzwerk Emulation durch die Integration eines Tools namens Netshaper in den Treiber der virtuellen Netzwerkkarte (veth). Dieser wird von OpenVZ bereitgestellt und kann von jedem virtuellen Knoten zur Kommunikation mit anderen Knoten oder dem Root Betriebsystem genutzt werden.

In OpenVz besteht der virtuelle Netzwerkkartentreiber aus zwei Komponenten. Einem virtuellen Netzwerkgerät, das nur innerhalb des virtuellen Knoten sichtbar ist und einem Netzwerkgerät, das nur für das außerhalb in der virtuellen Maschine laufenden Betriebssystem sichtbar ist. Typischerweise werden Instanzen der einen Komponente etho, eth1,... genannt und die der anderen veth<id>.0, veth<id>.1,... wobei <id> für die Id eines virtuellen

Knotens steht. Beide Geräte sind derart verbunden, dass Pakete, die an das eine Gerät geschickt werden, auch auf dem anderen Gerät sichtbar werden.

Wie in Abbildung 2.2 dargestellt, wird für die Netzwerkemulation in TVEE zwischen beide Netzerkgeräte eine zusätzliche Komponente eingefügt. Diese wird Netshaper genannt. Je nach gewünschtem Verhalten des Netzwerkes, werden im Netshaper z.B. Frames verzögert oder sogar verworfen. Das Verhalten des Netshapers richtet sich nach den Einstellungen, die über das Proc-Dateisystem vorgenommen werden können. Mögliche Einstellungen sind z.B. die Bandbreite, die Verlustrate und die Definition von Eigenschaften für bestimmte Verbindungen zwischen virtuellen Knoten. Diese Einstellungen können jeweils separat für die Sende- und Empfangsrichtung definiert werden.

Für beide Richtungen wird jeweils eine eigene Netshaper Instanz gestartet. Um Sendezeitpunkte von Frames anzupassen, besitzt jede Netshaper Instanz einen Puffer, in dem noch zu sendende Frames zwischengespeichert werden.

2.5 Techniken zur Experimentlaufzeitminimierung

Zur Minimierung der Experimentlaufzeit kommen in TVEE bisher zwei Techniken zum Einsatz: die Epochen basierte virtuelle Zeit und ein Platzierungsalgorithmus namens Netplace. Beide Techniken sollen im folgenden kurz vorgestellt werden.

2.5.1 Epochen basierte virtuelle Zeit

Während eines Experiments ist es möglich, dass die Auslastung der Testumgebung variiert. Ursache dafür kann z.B. eine sich ändernde Lastanforderung virtueller Knoten sein.

Zu keinem Zeitpunkt des Tests dürfen Rechner der Testumgebung überlastet werden. Ansonsten kann es zur Verfälschung von Messergebnissen kommen. Daher muss der TDF Faktor stets angemessen gewählt werden.

Um Überlasten zu verhindern und gleichzeitig das System möglichst gut auszulasten, ist es sinnvoll, den TDF adaptiv zu wählen. Um Testergebnisse nicht zu verfälschen, muss zudem die Anpassung des TDF auf allen physikalischen Knoten der Testumgebung nahezu gleichzeitig erfolgen. Dies macht eine gewisse Synchronisation der einzelnen Knoten nötig.

Im Rahmen der TVEE wurde ein Verfahren entworfen, das den TDF Faktor adaptiv an die aktuelle Lastsituation anpasst und dabei den Synchronisationsoverhead in einem vertretbaren Rahmen hält. Dem Verfahren liegt eine Epochen basierte virtuelle Zeit zugrunde. Die Laufzeit des Experiments wird dabei in Epochen unterschiedlicher Länge eingeteilt. Während einer Epoche ist der TDF Faktor konstant. Erst beim Übergang zu einer neuen Epoche wird dieser geändert und der geänderte Werte zeitgleich an alle virtuellen Maschinen der physikalischen

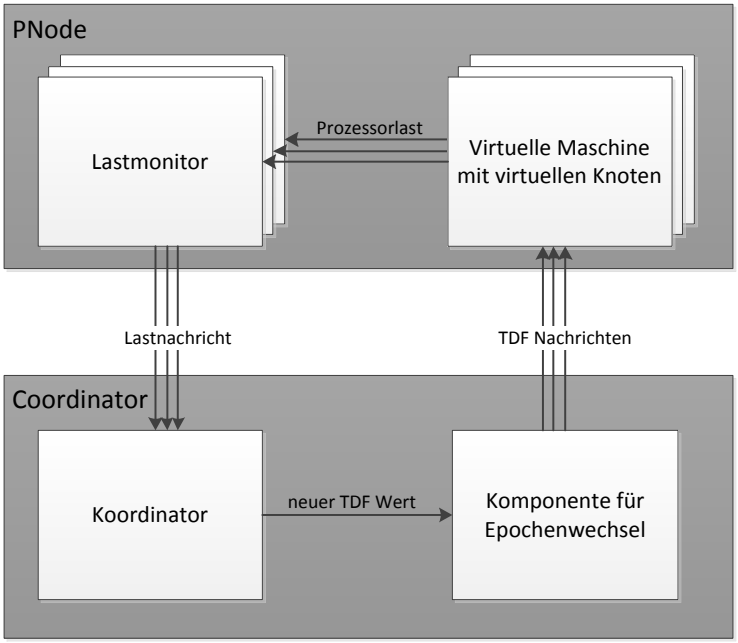


Abbildung 2.3: epochetime

Rechner weitergeleitet. Eine neue Epoche wird eingeleitet, falls die Last des Systems einen Überlastschwellwert übersteigt oder einen Unterlastschwellwert unterschreitet.

Zur adaptiven Anpassung des TDF kommt ein Regelkreis zum Einsatz. Dieser ist in **??** dargestellt. Eine zentrale Rolle im Regelkreis bildet der Koordinator. Er passt den TDF an die aktuelle Last an. Informationen über die aktuelle Last erhält er von einem verteilten Lastmonitor. Dieser zeichnet für jede virtuelle Maschine eines physikalischen Knoten Lasten auf und berechnet daraus die Last des physikalischen Knotens.

Da kein Knoten im System überlastet werden darf, bildet der Koordinator aus den einzelnen Lasten physikalischer Knoten das Maximum, um die Last des Systems zu bestimmen. Auf Basis dieser Last passt der Koordinator den TDF an und propagiert den geänderten TDF über die Komponente für den Epochenwechsel an die physikalischen Knoten. Je nach Über- oder Unterlastsituation wird der TDF schrittweise erhöht, bzw. erniedrigt.

2.5.2 NETplace

In TVEE wird die Laufzeit eines Experiments maßgeblich durch den TDF Faktor bestimmt. Da kein physikalischer Knoten überlastet werden darf, richtet sich dieser nach dem am stärksten ausgelasteten Knoten. Um eine möglichst hohe Auslastung des gesamten System zu

gewährleisten, ist es daher sinnvoll, alle physikalischen Knoten möglichst gleich auszulasten. Dies ermöglicht die Erhöhung der Geschwindigkeit mit der die virtuelle Zeit voranschreitet, was wiederum zu einer Verkürzung der Experimentlaufzeit führt.

Eine möglichst günstige Lastverteilung wird in TVEE durch die Verwendung eines automatischen Platzierungsalgorithmus, namens NetPlace erreicht.

Eine Platzierung ϕ wird dabei als Funktion verstanden, die virtuelle Knoten $n \in N$ auf virtuelle Maschinen $v \in V$ abbildet. Jede virtuelle Maschine kann durch den physikalischen Knoten $p \in P$ und die CPU $c \in C$, die der VM zugewiesen wurde, adressiert werden. Damit ergibt sich:$\phi : x \mapsto (p, c)$

Der Platzierungsalgorithmus verteilt virtuelle Knoten des Testszenarios auf physikalische Knoten der Testumgebung derart, dass die Gesamtlast und das Lastungleichgewicht zwischen physikalischen Rechnern möglichst klein sind.

Der Berechnung der Platzierung liegt ein Kostenmodell zugrunde. Einen wesentlichen Bestandteil dieses Modells bilden Kommunikationskosten. Diese werden auf Basis mittlerer Datenraten von Verbindungen zwischen virtuellen Knoten und der Art der Verbindung berechnet.

Im Kostenmodell werden 3 unterschiedliche Arten von Verbindungen zwischen virtuellen Knoten unterschieden:

- Intra-vm Links
- inter-vm Links
- inter pnode Links

Kommunizieren zwei virtuelle Knoten innerhalb einer virtuellen Maschine miteinander, so handelt es sich um einen *intra-vm* Link. Diese Kommunikatonsmethode verursacht die geringsten Kosten.

Tauschen 2 virtuelle Knoten Informationen aus, die sich in unterschiedlichen virtuellen Maschinen auf dem gleichem physikalischen Rechner befinden, so handelt es sich um einen *inter-vm* Link. Dieser verursacht mehr Kosten, ist allerdings günstiger als ein *inter-pnode* Link. Diese Art der Verbindung liegt vor wenn zwei virtuelle Maschinen miteinander kommunizieren, die sich auf verschiedenen physikalischen Rechnern befinden.

Neben Kommunikationskosten berücksichtigt der Algorithmus zudem noch mittlere Lasten der virtuellen Knoten.

Zur Platzierung der einzelnen Knoten kommt ein paralleler Algorithmus zum Einsatz. Dieser berechnet gleichzeitig eine initiale Platzierung mithilfe eines Greedy Ansatzes und eine Platzierung mithilfe eines Kantenschnitt basierten Ansatzes. Ergebnisse beider Ansätze werden mittels eines Hill Climbing Algorithmus optimiert. Das bessere Ergebnis bildet die Lösung des Algorithmus.

2.6 Konfiguration

Verteilte Software kann in der Regel sehr unterschiedliche Anforderungen haben: z.B. Skalierbarkeit, hohe Verfügbarkeit und Verlässlichkeit. Um sicher zu stellen, dass vorher definierte Anforderungen von der Software eingehalten werden, ist es nötig, sie unter verschiedenen Bedingungen zu testen.

Dazu müssen Testszenarien definiert werden. Diese können sich, z.B. in der verwendeten Hardware (z.B. 100 mbit Ethernet 1 gbit Ehternet), der Netztopologie, oder der Anzahl der physikalischen Rechner, auf denen die Software ausgeführt wird, unterscheiden. Ein einfaches Testszenario ist in Abbildung 2.4 dargestellt.

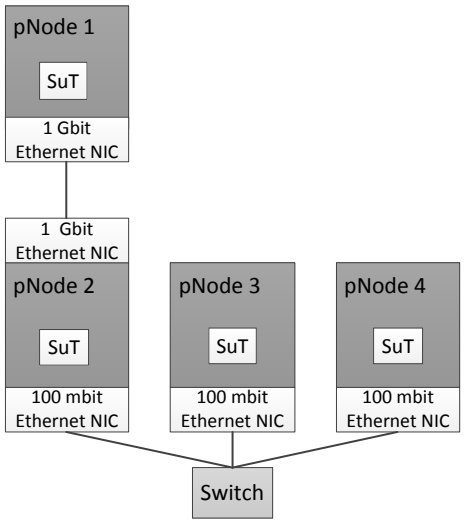


Abbildung 2.4: Beispiel für mögliches Testszenario

Um verteilte Software innerhalb der TVEE testen zu können, muss die Emulationsumgebung je nach Testszenario unterschiedlich konfiguriert werden können.

Dies geschieht mittels des TVEE Manager Frameworks. Es bietet die Möglichkeit, Testszenarien zu beschreiben und die Emulationsumgebung nach dieser Beschreibung automatisch zu konfigurieren. Eine manuelle, oft sehr aufwendige und fehleranfällige Konfiguration der Testumgebung soll dadurch vermieden werden.

Die Beschreibung des Testszenarios erfolgt über ein Ruby Skript. In diesem können Kommunikationsknoten, in denen die SUT ausgeführt werden soll, und Verbindungen zwischen ihnen definiert werden. Dazu stehen 2 Klassen zur Verfügung: *VNode* und *CollisonDomain*.

Über die *VNode* Klasse können virtuelle Knoten spezifiziert werden. Physikalische Knoten des Testszenarios müssen also zunächst auf virtuelle Knoten abgebildet werden. Die *VNode* Klasse bietet dabei unterschiedliche Konfigurationsmöglichkeiten. So können z.B. die Anzahl der verfügbaren Netwerkgeräte sowie CPU Limitierungen festgelegt werden.

Für jede *VNode* Instanz erzeugt das Framework später einen OpenVZ Container. Daher muss für jeden Container spezifiziert werden, in welcher virtuellen Maschine er gestartet werden soll. Eine Platzierung der Knoten kann z.B. vorab mittels NetPlace für das Testszenario bestimmt werden.

Für jedes Netzwerkgerät(Instanz der *VNic* Klasse) einer VNode Instanz wird ein virtuelles Netzwerkgerät(veth) angelegt. Für jedes Netzwerkgerät können dabei Parameter wie Bandbreite, Delay, Verlustrate eingestellt werden. Diese Parameter werden später an die Netshaper Instanzen weitergegeben.

Im TVEE Manager werden 3 Arten von Netzwerken unterstützt: Netzwerke, die auf Punkt zu Punkt Verbindungen beruhen, switch basierte Netzwerke und MANets(mobile adhoc networks).

Die Beschreibung aller Netze basiert beim TVEE Manager auf Instanzen der Klasse *CollisionDomain*. Mittels eines *CollisionDomain* Objekts lassen sich beliebig viele virtuelle Knoten miteinander verbinden. Dazu werden virtuelle Netzwerkkarten von Knoten, die miteinander kommunizieren möchten, an die gleiche *CollisionDomain* angebunden. Sind an einer Domain nur zwei Knoten angeschlossen, so handelt es sich um eine Punkt zu Punkt Verbindung; bei mehr Knoten, um ein switch basiertes Netzwerk. Eine Punkt zu Punkt Verbindung ist also als Spezialfall eines geswitchten Netzwerks modelliert.

MANet Szenarien werden ebenfalls über *CollisionDomains* realisiert. Dazu werden alle mobile Knoten an eine Collision Domain angebunden.

Für jedes *CollisionDomain* Objekt wird vom TVEE Manager später mindestens eine Linux Software Brücke erzeugt. Diese befindet sich innerhalb einer virtuellen Maschine. Werden *VNodes* der gleichen *CollisionDomain* auf unterschiedlichen virtuellen Maschinen platziert, so wird in jeder virtuellen Maschine, in der sich einer der *VNodes* befindet, eine Software Brücke erzeugt. Die unterschiedlichen Softwarebrücken werden dann mittels VLan verbunden. So kann sichergestellt werden, dass Nachrichten, trotz Aufteilung der Knoten, auf unterschiedliche virtuelle Maschinen nur von Netzwerkkarten, die an die gleiche *CollisionDomain* angeschlossen sind, empfangen werden.

Abbildung 2.5 zeigt wie eine konfigurierte Emulationsumgebung z.B. für das in Abbildung 2.4 dargestellte Testszenario aussehen könnte.

Für jeden physikalischen Knoten des Testszenarios wurde ein virtueller Knoten in der Testumgebung erzeugt. Zwei dieser Knoten wurden dabei auf der virtuellen Maschine *VM1*, die anderen auf der virtuellen Maschine *VM2*, platziert. Beide Maschinen befinden sich in diesem Fall auf dem gleichen physikalischen Knoten.

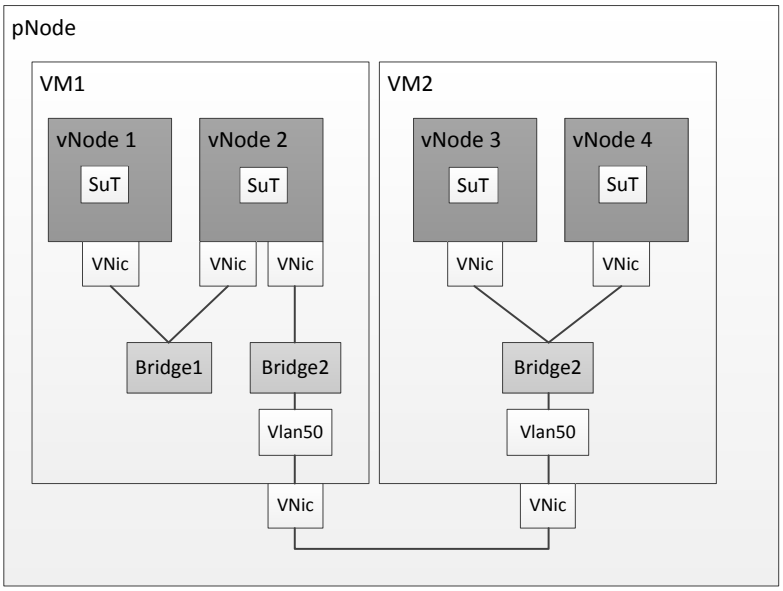


Abbildung 2.5: Beispiel für Konfiguration der TVEE

Für die Punkt zu Punkt Verbindung des Knotens *pNode 1* und *pNode 2* wurde eine Softwarebrücke angelegt und die passenden virtuellen Netzwerkkarten an diese Brücke gehängt. Für den geswitchten Teil des Netzwerks, also für die Verbindung der Knoten *pNode2*, *pNode3* und *pNode4*, wurde in beiden virtuellen Maschinen jeweils eine Brücke mit der Id 2 angelegt. Verbunden werden die Brücken über Netzwerkkarten beider virtuellen Maschinen. Damit bei den beiden Brücken nur Nachrichten der jeweils anderen Brücke ankommen, wurde auf beiden Seiten ein Vlan Gerät mit der Id 50 eingerichtet.

3 Related work

Für die Migration von virtuellen Knoten im Zusammenhang mit Load Balancing ließen sich keine passenden Quellen finden. Daher sollen hier ein ähnliche Probleme vorgestellt werden: Load Balancing in verteilten/parallelen Systemen

3.1 Load Balancing in verteilten/parallelen Systemen

Verteilte Systeme werden häufig zur Lösung von Problemen eingesetzt, die sich in Teilprobleme zerlegen lassen. Zur Lösung dieser werden Tasks erzeugt, die von Prozessoren des verteilten Systems verarbeitet werden. Durch die so erreichte Parallelisierung erhofft man sich eine Verkürzung der Rechenzeit, die für die Lösung eines Problems nötig ist. Entscheidend für die Performance dieses verteilten Ansatzes ist die Verteilung der einzelnen Tasks auf vorhandene Prozessoren. Um eine effiziente Nutzung der Ressourcen zu gewährleisten, muss die Last, die einzelne Tasks verursachen, möglichst gleichmäßig auf alle Prozessoren verteilt werden.

Zur gleichmäßigen Verteilung der Last sind aus der Literatur zwei Techniken bekannt: Statische - und dynamische Lastverteilung.

Bei der statischen Lastverteilung [SKS92] erfolgt die Zuweisung von Tasks zu Prozessoren zur Compilezeit. Mittels zur Verfügung stehendem a priori Wissen über Ressourcen(z.B. Anzahl Prozessoren, Größe des Speichers) und Tasks (z.B. mittlere Laufzeit) wird versucht, eine Platzierung der Tasks zu finden, die im Sinne der Lastverteilung optimal ist. Dazu kommen z.B. Techniken wie Graphpartitionierung und Simulated Annealing [Kir84] zum Einsatz.

Im Gegensatz zum statischen Verfahren erfolgt die Lastverteilung beim dynamischen Verfahren [SKS92] zur Laufzeit. Die Verteilung der Tasks orientiert sich dabei an dem aktuellen Systemzustand. Eine wichtige, in diesem Zusammenhang eingesetzte Technik bildet die Migration von Tasks. Dabei werden Tasks von einem Prozessor zu einem anderen verschoben. Auf diese Weise können Lastungleichgewichte im System beseitigt werden. Im Gegensatz zur statischen Lastverteilung erzeugt das dynamische Verfahren einen gewissen Overhead zur Laufzeit. So müssen z.B. für die Bildung des aktuellen Systemzustandes Lastnachrichten verschickt werden. Außerdem entstehen Kosten für den Transfer von Knoten.

Ziel beider Verfahren ist die Verkürzung der Task Laufzeiten, auch Antwortzeit genannt. Dies wird durch eine gleichmäßigere Auslastung des gesamten Systems erreicht.

3.1.1 Task Migration

Task Migration ist eine Technik, die bei der dynamischen Lastverteilung zum Einsatz kommt. Ziel der Task Migration ist eine gleichmäßigere Auslastung des gesamten Systems durch den Transfer einzelner Tasks. Sie kann präemptiv oder nicht präemtiv sein [SKS92].

Bei der präemptiven Task Migration werden Knoten zur Laufzeit verschoben. Da hierfür der aktuelle Status eines Task festgehalten werden muss, ist diese Methode mit zusätzlichen Kosten verbunden. Typischerweise enthält der aktuelle Status ein virtuelles Speicher Abbild , nicht gelesene I/O Puffer und Nachrichten sowie Zeiger auf geöffnete Dateien.

Im Gegensatz zur präemptiven Task Migration erfolgt der Transfer eines Tasks bei der nicht präemptiven Migration ausschließlich bevor der Task gestartet wurde. Ein Statusabbild ist bei dieser Methode also nicht nötig.

Task Migrationsalgorithmus

Im Folgenden sollen nun wesentliche Bestandteile eines Migrationsalgorithmus vorgestellt werden. Spätere Lösungsansätze für die dynamische Neuplatzierung werden sich an dem hier vorgestellten Schema orientieren.

Im Wesentlichen lässt sich nach [WLR89] ein Migrationsalgorithmus in 4 Phasen aufteilen: Processor Load Evaluation, Load Balancing Profitability Determination, Task Migration Strategy und Task Selection Strategy.

- **Processor Load Evaluation** In dieser Phase wird die Last jedes physikalischen Knotens im System ermittelt. Die Lastdaten dienen als Input für die Load Balancing Profitability Phase.
- **Load Balancing Profitability** In dieser Phase wird der Grad des Lastungleichgewichts ermittelt. Er dient als Indikator für mögliche SpeedUps, die durch den Transfer von Tasks erreicht werden können. Ist eine Migration im aktuellen Zustand sinnvoll, sind also Ersparnisse höher als Migrationskosten, so wird zur nächsten Phase übergegangen.
- **Task Migration Strategy** In dieser Phase werden physikalische Knoten ausgewählt, die an der Migration teilnehmen sollen. Sie können entweder die Rolle der Quelle oder des Empfängers einnehmen: also entweder Tasks abgeben oder Tasks bekommen.
- **Task Selection Strategy** In der letzten Phase werden Tasks der Quellknoten für den Transfer ausgewählt und an die Empfängerknoten verschickt.

Beispiele

Im Folgenden sollen Beispiele von Task Migrationsalgorithmen vorgestellt werden. Diese lassen sich grob in verteilte und zentrale Algorithmen einteilen.

Zentrale Algorithmen sind eher wenig verbreitet. Eine mögliche Ursache besteht in der begrenzten Skalierbarkeit zentraler Ansätze. Der physikalische Knoten, auf dem der Migrationsalgorithmus ausgeführt wird, kann leicht zum Engpass werden. Dies liegt z.B. am Aufwand, der benötigt wird, um die Last des Systems zu bestimmen. Er steigt mit zunehmender Prozessoranzahl.

Meist werden deshalb verteilte Algorithmen eingesetzt. Ein verbreiteter Ansatz ist die Einteilung des gesamten Systems in sich überlappende Domänen. Eine Domäne kann dabei z.B. durch eine Nachbarschaftsrelation definiert sein, also z.B. einen Prozessor und seine direkten Nachbarn umfassen. Load Balancing findet dann nur innerhalb einer Domäne statt.

Rendezvous Algorithmus Der Rendezvous Algorithmus ist ein zentraler Migrationsalgorithmus. In einer zentralen Komponente, auch Koordinator genannt, werden Lastinformationen eines jeden physikalischen Knotens des Systems gesammelt. Sind zwei Knoten sehr unterschiedlich ausgelastet, arrangiert der Koordinator ein *Rendezvous* zwischen beiden. In diesem kann der stärker ausgelastete Rechner Tasks an den weniger ausgelasteten abgeben. Welche Tasks genau migriert werden, muss zwischen den Rendezvous Partnern ausgehandelt werden.

Tiling Algorithmus Der Tiling Algorithmus [CPJL98] ist ein verteilter Migrationsalgorithmus. Ein wesentlicher Bestandteil des Algorithmus ist die Unterteilung des verteilten Systems in kleine disjunkte Domänen, sogenannte *Windows*. In diesen *Windows* können physikalische Knoten Wissen über Lasten austauschen.

Durch Migration von Knoten innerhalb einer Domäne wird in diesem Algorithmus ein perfektes lokales Loadbalancing angestrebt.

Um ein globales Load Balancing zu erreichen, werden Windows verschoben. D.h. die Menge der physikalischen Knoten, die eine Domäne bilden, wird verändert.

Gradient model load balancing method Die gradient model load balancing method [LK87] gehört zur Klasse der verteilten Migrationsalgorithmen. Genau wie beim Tiling Algorithmus wird das verteilte System in Domänen unterteilt. Allerdings werden hier überlappende und nicht disjunkte Domänen verwendet. D.h. ein physikalischer Knoten ist bei dieser Methode in mehr als einer Domäne vertreten.

Die Definition einer Domäne basiert dabei auf Nachbarschaftsbeziehungen. Sie umfasst einen Knoten und alle direkten Nachbarn des Knotens. Wichtig ist, dass Wissen über Last und Tasks nur innerhalb einer Domäne ausgetauscht werden kann.

Die Gradient model load balancing method gehört zur Gruppe der Receiver Initiated Algorithmen. Eine Migration wird von Knoten angestoßen, die sich in einem Zustand geringer Last befinden, also bereit sind, Tasks zu empfangen.

Der Algorithmus basiert auf einer Gradientkarte, die die kürzesten Entfernungen (in hops) zu einem wenig ausgelasteten Knoten enthält. Diese wird verteilt auf den physikalischen Knoten des Systems gespeichert. Jeder Knoten hält dabei seine kürzeste Entfernung zu einem wenig ausgelasteten Rechner fest. Diese wird auf Basis von Entfernungsinformationen direkter Nachbarn bestimmt.

Zu migrierende Tasks werden entlang dieser Gradientenkarte verschoben. Tasks wandern also auf dem kürzesten Weg von stark ausgelasteten zu weniger ausgelasteten Knoten. Jeder Knoten routet dabei Tasks zu dem ihn bekannten Knoten mit der kürzesten Entfernung zu einem wenig ausgelasteten Knoten.

Durch sukzessive lokale Migration kann dadurch ein globales Load Balancing erreicht werden.

Sender Initiated Diffusion Bei Sender Initiated Diffusion [ELZ86] [LRCM95] Algorithmen handelt es sich um Migrationsalgorithmen, die wie bei der Gradienten Methode auf lokalen, sich überlappenden Domänen aufbauen.

Anders als bei der Gradienten Methode wird die Migration allerdings von einem überlasteten physikalischen Knoten angestoßen. Dieser fungiert als Sender und gibt Tasks an seine Nachbarn ab.

Bei Sender Initiated Diffusion Algorithmen schicken Knoten Nachrichten mit ihrer aktuellen Last an alle Nachbarn. Beim Erhalten einer Lastnachricht wird ein lokaler Load Balancing Algorithmus angestoßen.

Dieser berechnet zunächst die mittlere Auslastung der Domäne und die Lastabweichung zum Mittelwert des Knotens, der die Nachricht empfangen hat. Falls dieser Knoten mehr Last als seine Nachbarn aufweist, wird die Überlast durch Migration von Tasks auf die Nachbarn verteilt. Der Anteil der Last, den ein Nachbarknoten erhält, richtet sich dabei nach seiner Auslastung.

Die Überlast diffundiert bei diesem Ansatz von einem Prozessor zu seinen Nachbarn und gleicht so das Lastungleichgewicht aus. Durch die Überlappung der einzelnen Domänen, wird ein globales Load Balancing erreicht.

Random Algorithm Beim Random Algorithmus handelt es sich um einen sehr einfachen verteilten Migrationsalgorithmus. Jedes mal, wenn auf einem Rechner des System ein neuer Task erzeugt wird, wird dieser zufällig auf einen anderen Rechner des Systems migriert.

Im Gegensatz zu den vorher zuvor vorgestellten Algorithmen setzt der Random Algorithmus auf eine präemptive Migration. Im Mittel wird jeder physikalische Knoten des Systems gleich belastet, unabhängig davon wo er sich im verteilten System befindet.

4 Dynamische Neuplatzierung

4.1 Einführung

In Abschnitt 2.5.2 wurde ein Algorithmus namens NETplace vorgestellt. Dieser berechnet eine möglichst Laufzeit optimale initiale Platzierung virtueller Knoten. Das in Netplace verwendete Kommunikationskostenmodell beruht auf Annahmen über mittlere Datenraten von Verbindungen zwischen virtuellen Knoten und deren mittleren Lasten. Diese müssen allerdings nicht unbedingt zutreffend sein. Zudem können Lasten virtueller Knoten während des Experiments schwanken, wodurch sich Lastverhältnisse des Systems zeitweise ändern können. Dadurch kann die initiale Platzierung suboptimal sein.

Abbildung 4.1 zeigt z.B. einen möglichen Lastverlauf eines virtuellen Knotens.

Der tatsächliche Ressoucenbedarf einer virtuellen Maschine kann sich, durch wechselnde Bedürfnisse virtueller Knoten in ihr, während eines Experiments ändern. Dies kann Auswirkungen auf die Experimentlaufzeit haben. Droht eine virtuelle Maschine überlastet zu werden, so muss die virtuelle Zeit verlangsamt werden.

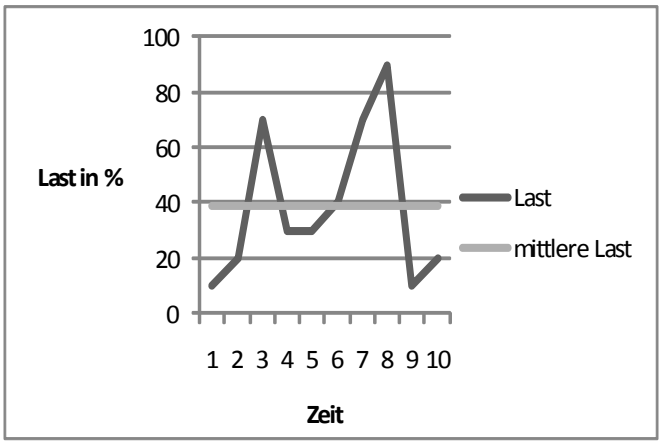


Abbildung 4.1: Beispiel für Lastverlauf eines virtuellen Knotens

Um eine Überlast zu vermeiden, kann es daher sinnvoll sein, virtuelle Knoten neu zu platzieren. Im Falle einer drohenden Überlast einer VM könnten z.B. virtuelle Knoten auf weniger belasteten VMs migriert werden.

Je nach aktuellem Ressourcenbedarf virtueller Knoten, kann eine andere Platzierung sinnvoll sein. Für eine gegebene Situation gilt es , eine möglichst optimale Platzierung zu finden. Dies ist eine Platzierung unter der die erwartete Experimentlaufzeit minimal ist.

Dabei ist zu beachten, dass für den Übergang von der aktuellen zu einer neuen Platzierung zunächst einige Kosten anfallen. So müssen beispielsweise virtuelle Knoten migriert sowie Netzwerktopologien angepasst werden. Dadurch erhöht sich die Experimentlaufzeit, was durch Einsparungen, die durch die neue Platzierung erreicht werden, zunächst ausgeglichen werden muss.

4.2 Architektur

In diesem Abschnitt sollen nun wesentliche Bestandteile der dynamischen Neuplatzierung von virtueller Knoten vorgestellt werden. Diese sind in Abbildung 4.2 dargestellt.

Zunächst werden die aktuellen Lastverhältnisse des Systems analysiert. Dies geschieht auf Basis aktueller Lasten physikalischer Knoten. Herrscht zwischen den Lasten ein großes Ungleichgewicht, so besteht ein hohes Optimierungspotential der aktuellen Platzierung.

Wurde von der Komponente *Beurteilung der Lastverhältnisse* ein Lastungleichgewicht festgestellt, wird der Neuplatzierungsalgorithmus angestoßen.

Dieser optimiert die aktuelle Platzierung in Hinblick auf die erwartete Experimentlaufzeit. Für die Abschätzung erwarteter Experimentlaufzeiten müssen nicht nur, wie in Netplace vorgestellt, Kommunikationskosten, sondern auch Rekonfigurationskosten, die für die Umsetzung einer Platzierung entstehen, betrachtet werden. Ein Modell für die Kommunikationskosten wird in 4.4 vorgestellt wird. Ein Modell zur Abschätzung erwarteter Rekonfigurationskosten enthält Abschnitt 4.5.

Das Ergebnis der Optimierung ist eine alternative Platzierung. Diese wird in einem nächsten Schritt in der Komponente *Beurteilung der Platzierung* bewertet. Senkt die alternative Platzierung die erwartete Experimentlaufzeit nicht, oder nur kaum, so wird sie nicht umgesetzt. Lohnt sich allerdings die Umsetzung, so muss die Emulationsumgebung rekonfiguriert werden. Dies geschieht durch die Komponente *Rekonfiguration* . Diese veranlasst u.a die Migration von virtuellen Knoten, deren Platzierung sich geändert hat. Darüber hinaus passt sie die Netzwerktopologie der Emulationsumgebung an die neue Situation an. In Abschnitt 4.3 wird näher auf die Rekonfiguration der TVEE eingegangen.

Wie man in 4.2 sehen kann, ergeben die Emulationsumgebung und die dynamische Neuplatzierung zusammen einen Regelkreis. Dabei nimmt die Emulationsumgebung die Rolle der

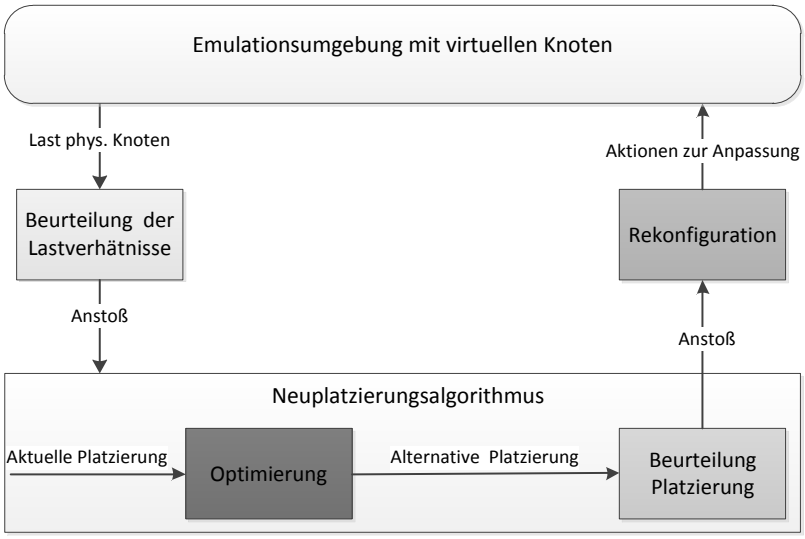


Abbildung 4.2: Architektur Neuplatzierung

Regelstrecke und die dynamische Neuplatzierung die des Reglers ein. Die Regelgröße ist die erwartete Experimentlaufzeit unter der aktuellen Platzierung. Diese soll möglichst klein sein.

In den folgenden Kapiteln wird nun näher auf die einzelnen Bestandteile der dynamischen Neuplatzierung eingegangen.

4.3 Rekonfiguration der TVEE

In diesem Abschnitt wird näher auf die Rekonfiguration der TVEE eingegangen. Sollen virtuelle Knoten neu platziert werden, muss die Emulationsumgebung angepasst werden. Es müssen z.B. virtuelle Knoten migriert und die Netzwerktopologie, welche die Kommunikation zwischen Knoten ermöglicht, angepasst werden.

In diesem Abschnitt wird auf wesentliche Bestandteile der Rekonfiguration eingegangen. Zunächst werden allerdings Anforderungen, die an die Rekonfiguration gestellt werden, vorgestellt.

4.3.1 Anforderungen

Zwei wesentliche Anforderungen der Rekonfiguration stellen Transparenz und geringe Kosten dar.

Transparenz Um ein laufendes Experiment nicht zu beeinflussen - denn dadurch könnte es zur Verfälschung von Messergebnissen kommen - ist es wichtig, dass die Rekonfiguration der TVEE für den virtuellen Knoten und die darin befindliche *Software under Test* (SuT) transparent ist.

Zwischen dem Ablauf des Experiments mit und ohne dynamische Neuplatzierung darf für die SuT kein Unterschied erkennbar sein. Es dürfen also z.B. während der Rekonfiguration keine Pakete verloren gehen.

Geringe Kosten Eine weitere Anforderung stellen geringe Kosten dar. Die Zeit, die für den Übergang von einer alten in eine neue Platzierung benötigt wird, muss gering sein. Sie verringert die Laufzeiteinsparung, die durch eine Neuplatzierung erreicht werden kann.

Daher sollten für die Neuplatzierung nötige Operationen möglichst effizient umgesetzt werden sowie die zur Verfügung stehenden Ressourcen möglichst gut ausgenutzt werden. Ein denkbarer Weg, um eine gute Ausnutzung bestehender Ressourcen zu erreichen, ist die parallele Ausführung von nötigen Operationen.

4.3.2 Operationen

In diesem Abschnitt sollen nun nötige Operationen zur Rekonfiguration der TVEE im Zuge einer neuen Platzierung vorgestellt werden. Als Einstieg betrachten wir zunächst Abbildung 4.3.

Sie zeigt eine Beispiel für eine Rekonfiguration der TVEE. In diesem wird die Position des virtuellen Knotens *vNode2* verändert. Er soll von der virtuellen Maschine *VM1* auf die virtuelle Maschine *VM2* umplatziert werden.

Im Zuge der neuen Platzierung muss die TVEE rekonfiguriert werden. Dazu wird der virtuelle Knoten *vNode2* sowie die Netshaper Instanzen (NS), die mit dem virtuellen Netzwergerät *veth2.0* verbunden sind, von der virtuellen Maschine *VM1* zur Maschine *VM2* migriert. Des Weiteren findet eine Anpassung der virtuellen Layer 2 Topologie, bestehend aus Softwarebrücken und Vlans, statt. Es wird ein Vlan Gerät mit er Id 50 in beiden virtuellen Maschinen und eine Softwarebrücke *bridge1* in der virtuellen Maschine *VM2* erzeugt. Die so entstehende neue Konfiguration der TVEE ist rechts in 4.3 dargestellt.

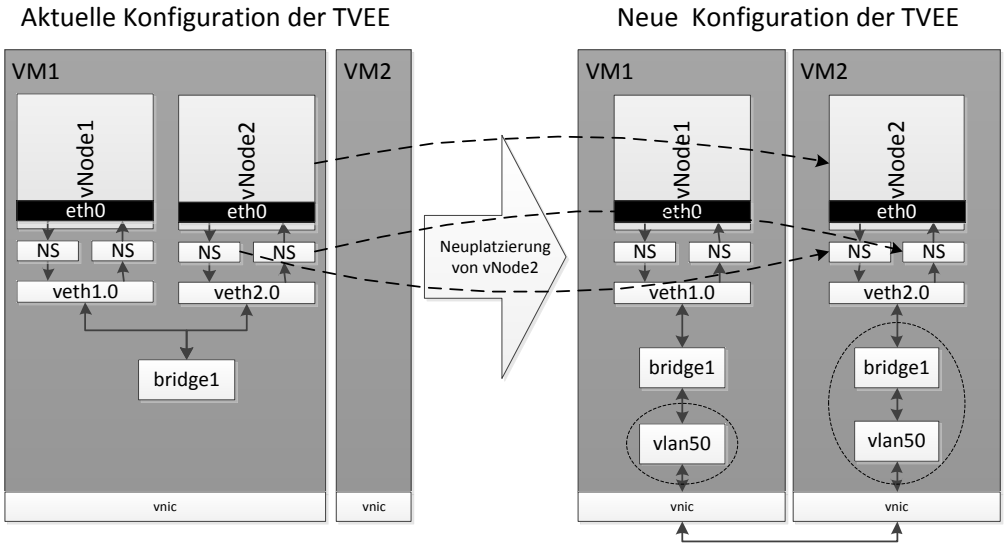


Abbildung 4.3: Beispiel für Neuplatzierung durch Migration

Allgemein sind folgende Operationen für die Rekonfiguration der TVEE im Zuge einer neuen Platzierung nötig.

- Migration virtueller Knoten
- Migration von Netshaper Instanzen
- Anpassung der virtuellen Layer 2 Topologie

Um der Forderung nach Transparenz nachzukommen, müssen darüber hinaus nachfolgende Operationen ausgeführt werden.

- Verlangsamung der globalen virtuellen Zeit
- Start/Stopp der Prozessausführung in virtuellen Knoten
- Zwischenspeichern von Paketen

Alle Operationen werden im folgenden nun näher vorgestellt. Dabei werden auch die für die Transparenz zusätzlich benötigten Operationen motiviert.

Migration virtueller Knoten

Die Migration virtueller Knoten bildet die Grundlage für den Übergang zu einer neuen Platzierung.

Für die Migration eines Knotens muss dazu zunächst dessen Zustand gesichert werden. Dabei muss der Zustand des Arbeitsspeichers,des Protokollstapels und des Dateisystem des virtuellen Knotens berücksichtigt werden. Die entstandene Sicherung, in der Regel eine Binärdatei, muss in einem nächsten Schritt zum Zielrechner transferiert werden. Dort ist der Knoten dann wiederherzustellen. Dieser Ablauf wird in Abbildung 4.4 dargestellt.

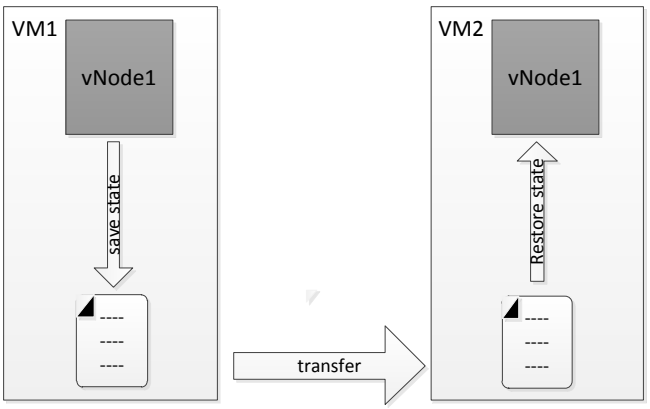


Abbildung 4.4: Migration eines virtuellen Knotens

In TVEE wird für die Knotenvirtualisierung OpenVZ eingesetzt. OpenVZ bietet bereits die Möglichkeit, virtuelle Knoten - in OpenVZ Container genannt- zu migrieren. Eine Sicherung (Dump) umfasst alle privaten Daten eines Containers. Dies sind z.B. der Addressraum, Registersätze, offene Dateien, offene Sockets, das aktuelle Arbeitsverzeichnis, Signal Handler, Timer, User und Prozessdaten.

Bei einem Dump werden all diese Daten gesammelt und in einer Image Datei abgelegt. Diese Image Datei kann dann zu einem anderen Rechner transferiert und dort als Container wiederhergestellt werden.

In der aktuellen Implementierung umfasst die Sicherung allerdings nicht das Dateisystem, also die „Festplatte“ auf dem der virtuelle Knoten arbeitet.

Aus diesem Grund müssen Maßnahmen getroffen werden, die sicherstellen, dass der Container auf dem Zielsystem das gleiche Dateisystem vorfindet. Je nach Schwere der Änderungen besteht ansonsten die Gefahr, dass Container nicht wiederhergestellt werden können, oder Prozesse innerhalb von Containern externe Veränderungen an offenen Dateien bemerken.

Das Dateisystem eines Containers befindet sich in OpenVZ im Ordner */vz/private/<ContainerID>* auf dem Host. Dieser Ordner muss auf dem Quell und ZielHost identisch sein.

Mögliche Maßnahmen identische Ordner zu erreichen sind:

1. Transfer des kompletten Ordners auf den Zielhost
2. Synchronisation der Ordner auf den Ziel- und Quellhost z.B. mittels rsync
3. Nutzung eines verteilten Dateisystemprotokolls auf dem Ziel- und Quellhost

Beim Transfer der Ordners auf den Zielhost werden eventuell bereits vorhandene Dateien überschrieben. So ist sichergestellt, dass alle Dateien auf dem Zielhost aktuell sind. Allerdings fallen bei dieser Methode bei jeder Migration hohe Kosten an. Diese sind außerdem nicht über mehrere Migrationen konstant. Im Laufe des Experiments können unter Umständen große Logdateien erzeugt werden, die den Transfer des kompletten Ordners gegen Ende des Experiments sehr teuer machen.

Weniger Daten müssen Übertragen werden, wenn beide Ordner mittels eines Synchronisationtools abgeglichen werden. Bei dieser Methode werden nur Änderungen beider Ordner übertragen. Um diese allerdings aufzuspüren, ist ein gewisser Overhead nötig. Beim rsync Protokoll z.B. werden die zu synchronisierenden Dateien zunächst in Blöcke aufgeteilt, von denen dann Prüfsummen berechnet werden. Dies geschieht sowohl auf dem Quell als auch auf dem Zielhost.

Wird ein verteiltes Dateisystemprotokoll benutzt, können beide Hosts über das Netzwerk auf den gleichen Ordner zugreifen. Dieser kann sich z.B. auf einem Fileserver befinden und über NFS angebunden werden. Bei der Migration müssen bei dieser Methode keine Daten übertragen werden. Allerdings ist der allgemeine Zugriff auf das Dateisystem langsamer aufgrund des Overhead eines Dateisystemprotokolls. Da sich Daten nicht auf lokalen Festplatten der Hosts befinden, müssen sie bei einem Zugriff über das Netzwerk übertragen werden. Dies führt zu einer höheren Belastung der Hosts. In Folge dessen steht weniger Rechenleistung für virtuelle Knoten (Container) zur Verfügung.

Falls virtuelle Knoten während eines Experimients in geringerem Umfang auf das Dateisystem zugreifen, empfiehlt sich die Verwendung des verteilten Dateisystemprotokolls. Werden aber z.B. sehr häufig neue Logeinträge in Logdateien erzeugt, so ist die Verwendung eines Synchronisationsprotokolls sinnvoller. Zwar sind die Kosten, die bei einer Migration für den Abgleich der Dateisystem benötigt werden dann höher, aber die Kosten für den Zugriff auf das Dateisystem sind deutlich niedriger. Zudem werden Migrationen sehr wahrscheinlich mit einer niedrigeren Frequenz auftreten als Dateisystem Zugriffe.

Migration von Netshaper Instanzen

Wird ein virtueller Knoten migriert, so müssen auch für den Knoten relevante Netshaper Instanzen migriert werden. Da diese, wie in 4.3 dargestellt, sich außerhalb des virtuellen Knotens befinden, werden sie bei der Migration des virtuellen Knotens nicht automatisch mit transferiert.

Daher muss für jede Instanz zunächst der aktuelle Zustand gesichert werden, um daraufhin zum Zielrechner übertragen und dort wiederhergestellt werden zu können.

Eine Sicherung muss dabei alle aktuellen Einstellungen der Instanz beinhalten. Dies sind z.B. Werte für Bandbreite, Verlustrate und Verzögerung. Außerdem müssen Pakete, die sich noch im Puffer des Emulationstools befinden, festgehalten werden.

Eine Möglichkeit, die Migration umzusetzen, besteht in der Nutzung des Proc Dateisystems. Dieses ermöglicht eine Kommunikation zwischen User und Kernelspace Programmen über spezielle Dateien.

Der aktuelle Zustand des Netshapers kann von einem Usertool über das Proc Dateisystem ausgelesen, transferiert und dann wiederhergestellt werden.

Anpassung der virtuellen Layer 2 Topologie

Für die Umsetzung einer neuen Platzierung muss die virtuelle Layer 2 Topologie der TVEE angepasst werden. Die Anpassung umfasst Komponenten wie Software Brücken und Vlans.

Wie in Kapitel 2.6 beschrieben, werden Netzwerktopologien von Testszenarien im TVEE Manager über *CollisionDomains* beschrieben. Für jede der Domänen wird eine Softwarebrücke innerhalb der virtuellen Maschinen erzeugt. Diese verbinden virtuelle Netzwerkkarten von Knoten, die sich in der gleichen *CollisonDomain* befinden miteinander. Sind Knoten der gleichen Domäne auf mehrere virtuelle Maschinen verteilt, so wird in jeder virtuellen Maschine, in der sich einer der Knoten befindet, eine Brücke und ein Vlan erzeugt.

Wird ein Knoten migriert, muss die Netzwerkkonfiguration so angepasst werden, dass der migrierte Knoten auch weiterhin mit anderen Knoten der gleichen Domäne kommunizieren kann.

Wie genau die TVEE dafür angepasst werden muss, soll nun im Folgenden beschrieben werden.

Zunächst muss für jede virtuelle Netzwerkkarte eines migrierten virtuellen Knotens bestimmt werden, an welche Domäne sie angebunden war. Die Anpassung der TVEE richtet sich dann danach, ob sich alle Knoten, die an diese Domäne angebunden sind, in der gleichen virtuellen Maschine befinden oder ob sie auf verschiedene Maschinen verteilt sind.

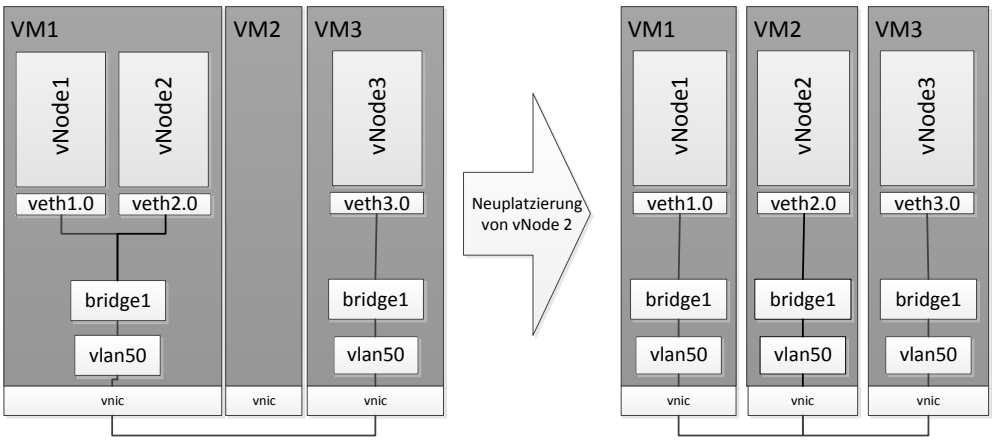


Abbildung 4.5: Anpassung von Netzwerkkomponenten - virtuelle Knoten verteilt

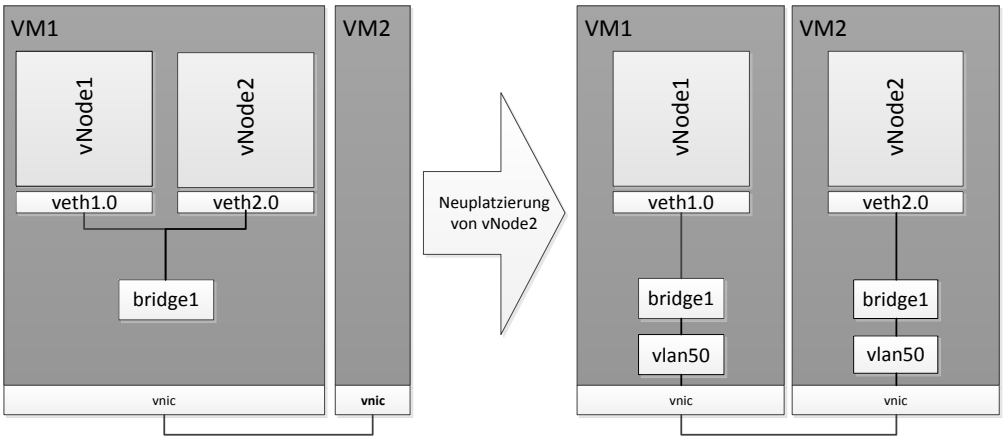


Abbildung 4.6: Anpassung von Netzwerkkomponenten - virtuelle Knoten zusammen

- **Knoten auf mehrere VMs verteilt:** In dieser Situation lassen sich zwei Fälle unterscheiden. Befindet sich auf der Ziel virtuellen Maschine schon ein Knoten der zur gleichen Domäne gehört, so muss die virtuelle Netzwerkkarte des migrierten Knotens nur an die passende Brücke der Domäne angebunden werden. Abbildung 4.7 zeigt ein Beispiel für diesen Fall

  Ist dies nicht der Fall, muss eine Software Brücke und ein Vlan mit passender Vlan Id erzeugt werden. Ein Beispiel ist in Abbildung 4.5 zu sehen.

- **Alle Knoten in gleicher VM:** Befanden sich vor der Migration alle Knoten der gleichen Domäne in der gleichen virtuellen Maschine, so muss zunächst eine freie VlanId für

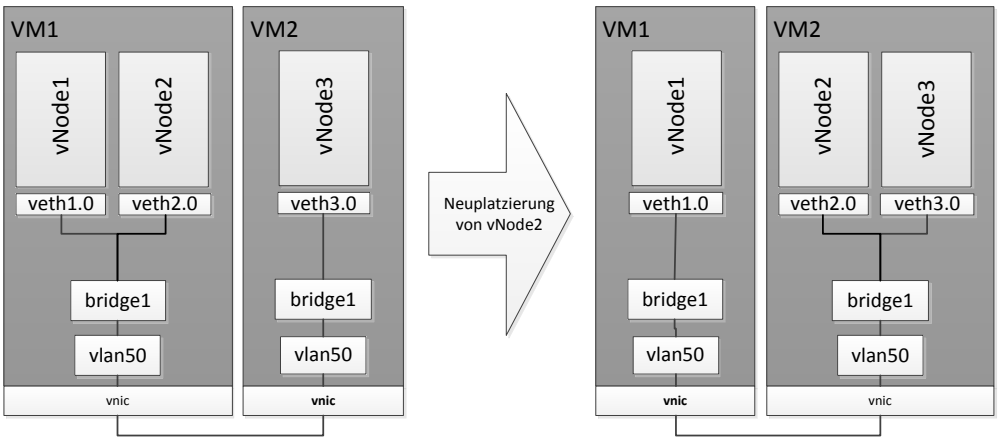


Abbildung 4.7: Anpassung von Netzwerkkomponenten - virtuelle Knoten verteilt

die Domäne gewählt werden. Danach ist in der Ziel VM eine Softwarebrücke und ein VLan zu erzeugen. Zusätzlich muss auf der Quelle VM ein VLan eingerichtet werden. Dies ist allerdings nur nötig, falls durch die Neuplatzierung nicht alle Knoten, deren virtuelle Netzwerkkarte sich an der Brücke befanden, migriert werden konnten. Abbildung 4.6 zeigt ein Beispiel für diesen Fall.

Wurden diese Änderungen für alle migrierten Knoten durchgeführt, so können in einem letzten Schritt Software Brücken gelöscht werden, an die keine virtuelle Netzwerkkarte mehr angebunden ist.

Zusammenfassend sind also für die Anpassung der Netzwerkkomponenten eine Auswahl folgender Aktionen nötig.

- Erzeugung einer Softwarebrücke
- Erzeugung eines Vlans
- Anbindung einer virtuellen Netzwerkkarte an eine Softwarebrücke
- Löschen einer Softwarebrücke
- Löschen eines Vlans

Diese sind lokal in den einzelnen virtuellen Maschinen auszuführen. Wer die nötige Auswahl der Aktionen erzeugt und auf welche Weise die Koordination der Operation abläuft, wird in Abschnitt 4.8.2 vorgestellt.

Verlangsamung der globalen virtuellen Zeit

Wird ein Knoten im Zuge der Rekonfiguration migriert, so ist er für eine bestimmte Zeit offline. Während dieser Zeit läuft die, von den Knoten zur Zeitmessung benutzte, virtuelle Zeit weiter. Timer, die von Anwendungen innerhalb des Containers genutzt werden, können daher auslaufen. Ohne gewisse Maßnahmen ist die Rekonfiguration daher nicht transparent.

Um eine transparente Rekonfiguration zu ermöglichen, muss diese in keiner oder in einem sehr kleinen virtuellen Zeitintervall durchgeführt werden. Dies kann durch die Wahl eines sehr hohen TDF Wertes beim Koordinator geschehen. Eine Rekonfiguration, die mehrere Sekunden in realer Zeit dauert, kann dadurch in wenigen Millisekunden virtueller, von den Knoten benutzter Zeit, durchgeführt werden. Dadurch kann der Auslauf von Timern in zu migrierenden Knoten vermieden werden.

Start/Stopp der Prozessausführung

Die Rekonfiguration beinhalten Operationen wie die Migration von Knoten und die Anpassung der Netzwerktopologie. Diese können die Ausführung der zu testenden Software beeinflussen.

So kann es z.B. zu Paketverlusten aufgrund der Migration kommen. Dies ist der Fall, wenn SuT Instanzen während der Rekonfiguration Pakete an Knoten schicken, deren Zustand gerade transferiert wird. Da während dieser Zeit der virtuelle Knoten nicht existiert, existiert im Netzwerk auch keine Neztwerkkarte mit der richtigen Zieladresse. Pakete, die an diesen Knoten geschickt werden, können daher nicht zugestellt werden.

Um eine transparente Rekonfiguration zu ermöglichen, sollte infolgedessen während der Migration von Knoten keine SuT mehr ausgeführt werden. Dies kann z.B. durch das Setzen aller virtuellen Knoten in einen Haltezustand vor der Migration erreicht werden. Beim Übergang in diesen Zustand werden alle Prozesse innerhalb der virtuellen Knotens gestoppt und der Protokollstapel angehalten. Am Ende der Rekonfiguration müssen die virtuellen Knoten dann wieder in einen ausführenden Zustand gesetzt werden, damit die Prozessausführung wieder aufgenommen werden kann.

OpenVZ bietet bereits die Möglichkeit, einen virtuellen Knoten in einen Haltezustand,in OpenVZ *suspend* genannt, zu setzen und daraus wieder zu lösen,in OpenVZ *resume* genannt.

Wird jeder virtuelle Knoten, in dem sich die zu testende Software befindet, in einen Haltezustand gebracht, hat dies einen positiven Nebeneffekt für die Rekonfiguration. Es stehen in diesem Fall mehr Ressourcen für die Rekonfiguration zur Verfügung.

Für den Übergang eines Knotens in den Haltezustand wird allerdings eine gewisse Rechenzeit benötigt. Da virtuellen Knoten, die sich in der gleichen virtuellen Maschine befinden, nur eine CPU zur Verfügung steht, bedeutet dies, dass nicht alle Knoten gleichzeitig diesen Zustandswechsel vollziehen können. Je mehr Knoten sich in einer virtuellen Maschine befinden (dies können mehrere Tausend sein), desto größer wird die Zeitspanne zwischen dem ersten und dem letzten Knoten, der in den Haltezustand wechselt. Demzufolge kann es vorkommen, dass Pakete an virtuelle Knoten geschickt werden, die sich schon im Haltezustand befinden und aufgrund ihres Zustandes die gesendeten Pakete nicht mehr entgegennehmen können. Der nächste Abschnitt befasst sich mit dieser Problematik.

Zwischenspeichern von Paketen

Wie im vorigen Abschnitt vorgestellt, ist es nicht möglich, alle Prozesse und Protokollstacks gleichzeitig anzuhalten. Daher kann es vorkommen, dass SuT Instanzen Pakete verschicken, die von anderen Instanzen nicht mehr entgegengenommen werden können. Dieses Phänomen tritt auf, wenn die Prozessausführung und der Protokollstack des Containers, in dem sie sich die Ziel SuT befindet, schon angehalten wurden.

In diesem Fall müssen Pakete außerhalb des Containers zwischengespeichert werden.

Dazu kann z.B. der Netshaper verwendet werden. Dieser speichert, falls aktiv, alle Pakete, die an Netzwerkgeräte virtueller Knoten ausgeliefert werden sollen, zwischen. Wurde ein virtueller Knoten in einen Haltezustand versetzt, so kann dies vom Netshaper detektiert und die Zustellung von Nachrichten ausgesetzt werden. Damit werden Nachrichten automatisch im Netshaper Puffer gesichert.

Wechselt der Knoten dann wieder in einen ausführenden Zustand, so kann mit der Zustellung fortgefahren werden. Da der Netshaper allerdings nur beim Empfang neuer Pakete aufgerufen wird, kann auch nur in diesem Fall ein Zustandswechsel festgestellt werden. Werden also keine weiteren Pakete an den Knoten gesendet, so wird der Netshaper die zwischengespeicherten Pakete nicht mehr ausliefern.

Diese Problematik kann auf verschiedene Weisen gelöst werden.

- Durch periodisches Nachfragen des Knotenzustandes
- Durch Hook in der Resume Methode des Knotens
- Durch Anstoß der Zustellung mitels Proc Dateizugriff

Wird periodisch nach dem aktuellen Zustand eines Knotens gefragt, so werden unnötig wichtige Ressourcen verbraucht, zumal für jede virtuelle Netzwerkkarte zwei Netshaper Instanzen angelegt werden und ein Testszenario mehrere hundert Tausend virtuelle Netzwerkkarten umfassen kann.

Sinnvoller ist da ein Hook in der Resume Methode des Knotens. In diesem Fall muss jedoch direkt in die OpenVZ Implementierung eingegriffen werden.

Alternativ kann nach der Wiederaufnahme der Prozessausführung eines virtuellen Knotens den angeschlossenen Netshaper Instanzen mitgeteilt werden, dass eine Auslieferung von Paketen nun wieder möglich ist. Dies kann z.B. durch ein Userspace Tool erfolgen, dass über das Proc Dateisystem mit dem Netshaper kommuniziert.

4.3.3 Reihenfolge der Operationen

Zwischen den Operationen, die für die Rekonfiguration benötigt werden, bestehen gewisse Abhängigkeiten. Diese sollen im Folgenden nochmal kurz aufgelistet werden. Aus ihnen lässt sich eine sinnvolle Reihenfolge der Operationen ableiten.

Es bestehen folgende Abhängigkeiten

- Operationen dürfen keine virtuelle Zeit kosten. Daher muss zu Beginn der TDF auf einen hohen Wert gesetzt und am Ende diese Einstellung wieder rückgängig gemacht werden.
- Während Knoten migriert werden, dürfen keine Pakete verschickt werden. Der Stopp der Prozessausführung muss also vor der Migration geschehen.
- Netshaper Instanzen können erst wiederhergestellt werden, wenn die mit ihnen verbundenen virtuellen Netzwerkkarten wiederhergestellt wurden.

Eine Reihenfolge, die die obigen Abhängigkeiten berücksichtigt sieht wie folgt aus.

1. Setzen eines hohen TDF
2. Stopp der Prozessausführung
3. Sicherung des Zustands zu migrierender Knoten
4. Sicherung des Zustands zu migrierender Netshaper Instanzen
5. Transfer aller gesicherten Daten
6. Wiederherstellung des Zustands zu migrierender Knoten
7. Wiederherstellung des Zustands zu migrierender Netshaper Instanzen
8. Anpassung der Layer 2 Topologie
9. Start der Prozessauführung
10. Rücksetzen der TDF Änderung

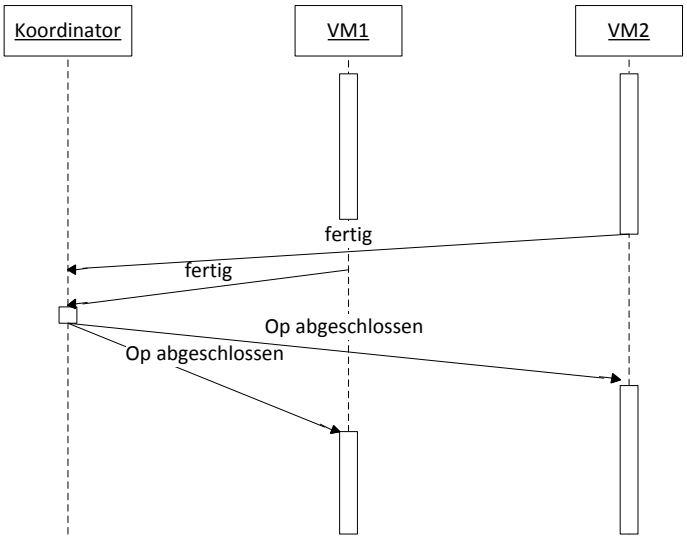


Abbildung 4.8: Synchronisation - zentraler Ansatz

4.3.4 Synchronisation einer verteilten Operation

Im letzten Kapitel wurde eine Reihenfolge nötiger Rekonfigurationsoperationen festgelegt. Diese beruhte auf Abhängigkeiten einzelner Operationen.

Unabhängig davon wie Operationen ausgeführt werden, muss diese Reihenfolge eingehalten werden. In den meisten Fällen ist es möglich, eine Operation parallel auf mehreren virtuellen Maschinen auszuführen

Betrachten man z.B. die Operation *Stopp der Prozessausführung*, so lässt sich diese sehr einfach parallel ausführen. Das Ziel der Operation besteht im Setzen aller virtueller Knoten in den Haltezustand. Dazu müssen in jeder virtuellen Maschine alle virtuellen Knoten „suspended“ werden, was gleichzeitig in allen VMs erfolgen kann.

Bei der verteilten Ausführung einer Operation, benötigen die beteiligten CPUs unter Umständen eine unterschiedlich lange Rechenzeit. z.B. weil für die Operation unterschiedliche Aktionen in den einzelnen VMs ausgeführt werden müssen. Bei der Anpassung der Layer 2 Topologie kann es beispielsweise vorkommen, dass in einer VM mehr Brücken als in einer anderen erzeugt/gelöscht werden müssen. Daher ist es für eine VM schwierig festzustellen, wann eine Operation abgearbeitet ist.

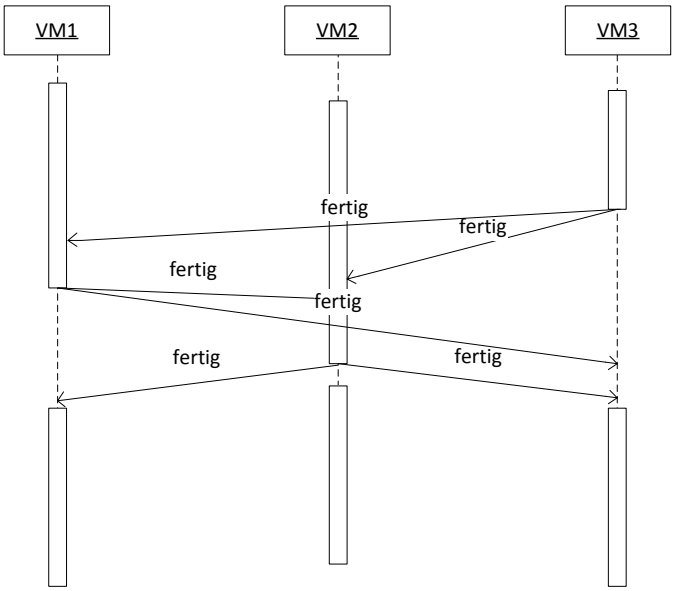


Abbildung 4.9: Synchronisation - verteilter Ansatz

Da aber Operationen in einer bestimmten Reihenfolge abgearbeitet werden müssen, wird ein Mechanismus benötigt, der sicherstellt, dass folgende Operationen erst gestartet werden können, wenn alle verteilten Arbeiten zur aktuellen Operation abgeschlossen sind.

Dazu müssen virtuelle Maschinen, die an der Operation beteiligt sind, synchronisiert werden. Dies kann auf zwei Arten erfolgen.

- Zentral über einen Koordinator
- Verteilt

Der Zentrale Ansatz ist in Abbildung 4.8 dargestellt. Dabei läuft die Synchronisation über einen Koordinator ab. Dieser signalisiert den VMs erst, dass die Folge-Operationen ausgeführt werden kann, wenn er eine Nachricht von allen virtuellen Maschinen erhalten hat, die an der verteilten Ausführung der aktuellen Operation beteiligt waren. Pro verteilter Operation werden bei diesem Ansatz $2 * |VM|$ Nachrichten benötigt.

Beim verteilten Ansatz wird die Ausführung der Operationen nicht durch einen Koordinator gesteuert. Stattdessen kann eine VM eine neue Operation starten, wenn sie Nachrichten aller VMs, die an der vorangegangenen Operation beteiligt waren, erhalten hat. Bei diesem Ansatz werden pro verteilter Operation $(|VM| - 1) * |VM| \approx \|VM|^2$ Nachrichten benötigt. Der verteilte Ansatz ist in 4.9 dargestellt.

Bei beiden Ansätzen wurde davon ausgegangen, dass eine VM genau weiß an welchen Operationen sie teilnimmt und welche Aktionen sie durchzuführen hat. Woher sie dieses Wissen hat, wird in Abschnitt 4.8.2 erläutert. In diesem Kapitel wird näher auf die Koordination der Rekonfiguration eingegangen.

Beim zentralen Ansatz werden weniger Nachrichten benötigt als beim verteilten Ansatz. Dieser ist also dem verteilten Ansatz vorzuziehen. Zumal die Belastung des Koordinators nur unwesentlich höher ist als die im verteilten Ansatz. Also der Koordinator nicht zum „bottle neck“ werden kann.

4.4 Kostenmodell Kommunikation

In 2.5.2 wurde ein Kostenmodell angedeutet mit dessen Hilfe sich die Laufzeit eines Experiments bei gegebener Platzierung virtueller Knoten voraussagen lässt. Für die Berechnung der Laufzeit werden Prognosen über mittlere Datenraten von Verbindungen zwischen virtuellen Knoten und Auslastungen virtueller Knoten benötigt. Dieses Modell soll im nächsten Abschnitt im Detail vorgestellt werden.

Die TVEE besteht aus einer Reihe von physikalischen Rechnern $p \in P$. Jeder Rechner besitzt $|C_p|$ Prozessoren, wobei einem Prozessor $c \in C$ eine virtuelle Maschine (VM) zugewiesen werden kann. Jede VM wird durch ein Tupel (p, c) identifiziert. In allen virtuellen Maschinen können virtuelle Knoten $i \in N$ ausgeführt werden.

Jede Verbindung zwischen virtuellen Knoten verursacht eine bestimmte Last. Diese wird aufgespalten in

- Last in der VM, in der sich der empfangende Knoten befindet. Im folgenden VM_{tx} genannt
- Last in der VM,in der sich der sendende Knoten befindet. Im folgenden VM_{rx} genannt.
- Last im Host-Os beider VMs. Im folgenden HOST-OS genannt.

Die Gesamtlast für eine Verbindung ergibt sich damit zu $L = VM_{tx} + VM_{rx} + 2 * \text{HOST-OS}$. Sie wird in CPU Zyklen pro Zeiteinheit angegeben. Maßgeblich für die Höhe der verursachten Last ist die Art der Verbindung der Knoten. In der TVEE lassen sich 3 Verbindungsarten unterscheiden:

- **intra-vm** Kommunizieren virtuelle Knoten in der gleichen VM über eine Softwarebrücke miteinander, so handelt es sich um eine intra-vm Verbindung. Bei dieser Verbindung wird nur die virtuelle Maschine, in der sich beide befinden, belastet.

- **inter-vm** Befinden sich die kommunizierenden Knoten in unterschiedlichen VMs, aber auf dem gleichen physikalischen Rechner, so handelt es sich um eine inter-vm Verbindung. Dabei werden beide virtuellen Maschinen sowie das HOST-OS belastet. Nach [GHR] ist diese Arte der Verbindung ungefähr 10 mal teurer als die intra-vm Verbindung.
- **inter-pnode** Befinden sich die Kommunikationspatner auf unterschiedlichen physikalischen Rechnern, so handelt es sich um eine inter-pnode Verbindung. Bei dieser Art der Verbindung fallen die höchsten Kosten an. Laut [GHR] ist sie 2 mal teurer als eine inter-vm Verbindung.

Sei nun eine Platzierung ϕ (Abbildung von virtuellen Knoten auf VMs), mittlere Datenraten β_{ij} von Verbindungen zwischen virtuellen Knoten und Lasten λ_i aller virtuellen Knoten $i \in N$ gegeben , dann wird ein HOST-OS durch Verbindungen von virtuellen Knoten wie in Formel (4.1) dargestellt, belastet.

$$\Lambda_p^{host-os} = \sum_{\substack{i,j \in N \\ \phi(i)=(p,c) \wedge \phi(j)=(p',c') \\ \vee \phi(i)=(p',c') \wedge \phi(j)=(p,c)}} \beta_{ij} * \begin{cases} \kappa_{intra-vm,Host-OS} & \text{falls } p = p' \wedge c = c' \\ \kappa_{inter-vm,Host-OS} & \text{falls } p = p' \wedge c \neq c' \\ \kappa_{inter-pNode,Host-OS} & \text{falls } p \neq p' \wedge c \neq c' \end{cases} \tag{4.1}$$

Die Last des Host-OS entspricht der Summe der Lasten die durch Verbindungen verursacht werden, bei denen sich ein Kommunikationspartner auf dem gleichen physikalischen Rechner befindet wie das Host-OS. Die Art der Verbindung bestimmt dabei den Faktor κ.

Die Formeln zur Berechnung von Lasten in virtuellen Maschinen für ausgehende und eingehende Verbindungen lassen sich analog zur Formel des Host-OS definieren. Diese sind in (4.2) und (4.3) dargestellt.

$$\Lambda_{p,c}^{VM_{rx}} = \sum_{\substack{i,j \in N \\ \phi(i)=(p',c') \wedge \phi(j)=(p,c)}} \beta_{ij} * \begin{cases} \kappa_{intra-vm,VM_{rx}} & \text{falls } p = p' \wedge c = c' \\ \kappa_{inter-vm,VM_{rx}} & \text{falls } p = p' \wedge c \neq c' \\ \kappa_{inter-pNode,VM_{rx}} & \text{falls } p \neq p' \wedge c \neq c' \end{cases} \tag{4.2}$$

$$\Lambda_{p,c}^{VM_{tx}} = \sum_{\substack{i,j \in N \\ \phi(i)=(p,c) \wedge \phi(j)=(p',c')}} \beta_{ij} * \begin{cases} \kappa_{intra-vm,VM_{tx}} & \text{falls } p = p' \wedge c = c' \\ \kappa_{inter-vm,VM_{tx}} & \text{falls } p = p' \wedge c \neq c' \\ \kappa_{inter-pNode,VM_{tx}} & \text{falls } p \neq p' \wedge c \neq c' \end{cases} \tag{4.3}$$

Die Last einer virtuellen Maschine wird durch die Last für eingehende und ausgehende Verbindungen zwischen virtuellen Knoten sowie durch die Last von virtuellen Knoten, die sich in ihr befinden, bestimmt. Dies ist in Formel (4.4) dargestellt.

$$\Lambda_{p,c}^{vm} = \Lambda_{p,c}^{VM_{tx}} + \Lambda_{p,c}^{VM_{tr}} + \sum_{i \in N \wedge \phi(i) = (p,c)} \lambda_i \tag{4.4}$$

Mit (4.5) lässt sich nun die Auslastung einer CPU bestimmen. Da virtuelle CPUs der VMs physikalischen CPUs zugewiesen werden erfährt jede CPU mindestens die Last der virtuellen Maschine. Zusätzlich wird aber noch die Last des Host-OS auf alle verfügbaren CPUs eines physikalischen Rechners verteilt. Jede CPU erfährt daher noch einen gewissen Anteil der Host-OS Last.

$$\Lambda_{p,c} = \max(\Lambda_{p,c}^{vm}, (\frac{1}{|C_p|} * (\Lambda_p^{host-os} + \sum_{c' \in C_p} \Lambda_{p,c'}^{vm}))) \tag{4.5}$$

Auf Basis errechneter Lasten einzelner Prozessoren lässt sich durch Formel (4.6) die erwartete Experimentlaufzeit für ein bestimmtes virtuelles Zeitintervall $\theta_{virtual}$ berechnen. ν_{CPU} steht dabei für die Geschwindigkeit der CPU. Diese wird in Zyklen pro Zeiteinheit angegeben.

$$\theta_{real} = \frac{\max\limits_{p \in P, c \in C_p} (\Lambda_{p,c})}{\nu_{CPU}} * \theta_{virtual} = TDF * \theta_{virtual} \tag{4.6}$$

Werden mehr Zyklen benötigt als der Prozessor pro Zeiteinheit zur Verfügung stellt, ist also $\frac{\max\limits_{p \in P, c \in C_p} (\Lambda_{p,c})}{\nu_{CPU}} > 0$, so muss das Experiment verlangsamt werden. Dabei gibt der Quotient den Faktor an, mit dem die Zeit skaliert werden muss. Dieser wird TDF (Time Dilation Factor) genannt.

4.5 Kostenmodell Rekonfiguration

Bei der Rekonfiguration der TVEE entstehen Kosten. Diese verlängern die Experimentlaufzeit und müssen daher bei der Beurteilung von möglichen neuen Platzierungen berücksichtigt werden.

In diesem Kapitel soll ein Modell vorgestellt werden mit dem sich erwartete Kosten für die Umsetzung einer neuen Platzierung ermitteln lassen. Es orientiert sich an den für die Rekonfiguration benötigten Operationen. Diese wurden in Kapitel 4.3 vorgestellt. Sie sind:

- Setzen/Rücksetzen des TDF
- Start/Stopp der Prozessausführung in virtuellen Knoten (suspend/resume)
- Migration von virtuellen Knoten und Netshaper Instanzen (migration)
- Anpassung der Layer 2 Topology (layer2adaption)

Die Gesamtkosten der Rekonfiguration ergeben sich aus der Addition der Zeit, die für die einzelnen Operationen benötigten wird, da diese, wie in 4.3.3 beschrieben, hintereinander auszuführen sind. Dies ist in Formel (4.7) dargestellt. Das Setzen und Rücksetzen des TDF ist mit konstanten, sehr geringen Kosten verbunden. Daher werden diese hier nicht aufgeführt. In der Formel bezeichnet ϕ die aktuelle Platzierung und ϕ' die durch die Rekonfiguration zu erzeugende neue Platzierung.

$$\theta^{\phi,\phi'}_{reconfiguration} = \theta_{suspend} + \theta_{migration} + \theta_{layer2adaption} + \theta_{resume} \tag{4.7}$$

Im Folgenden wird nun näher auf die Kosten einzelner Operationen eingegangen.

4.5.1 Start/Stopp der Prozessausführung

Zu Beginn der Rekonfiguration muss die Prozessausführung in allen Knoten gestoppt werden. Dazu muss jedem Prozess eines virtuellen Knotens ein Stopp Signal geschickt werden. Dies führt zu einem TaskStopped Eintrag in der Scheduler Tabelle, wodurch der Prozess keine Rechenzeit mehr erhält.

Die Verarbeitung des Stopp Signals kann je nach Reaktionsverhalten eines Prozesses unterschiedlich lange dauern. Um sicherzugehen, dass alle Prozesse eines virtuellen Knotens schließlich gestoppt sind, muss daher periodisch deren Zustand abgefragt werden. In OpenVZ kann über das VZ Control Tool ein *Suspend* Befehl an einen Knoten geschickt werden. Daraufhin werden alle Prozesse in diesem Knoten gestoppt und die virtuellen Netzwerkgeräte abgeschaltet.

Der Aufwand, der für das „suspenden“ aller Knoten benötigt wird, richtet sich nach der Anzahl der virtuellen Knoten und den sich darin befindlichen Prozessen. Die Kosten steigen in etwa linear mit der Anzahl der Knoten und mit der Anzahl der sich darin befindlichen Prozessen. Die Kosten erhöhen sich allerdings nur gering mit zunehmender Anzahl von Prozessen innerhalb eines Knotens.

Da sich in TVEE in einem virtuellen Knoten nur wenig Prozesse befinden (im Normalfall nur der Hauptprozess und die der SuT), wird die Anzahl hier als konstant angesehen.

(4.8) $\theta_{suspend} = \max_{vm \in VM}(n_{vm}) * \kappa_{suspend}$

Jede virtuelle Maschine hat eine eigene CPU. Virtuelle Knoten unterschiedlicher VMs können parallel in den Haltezustand gebracht werden. Die Gesamtkosten für den kompletten Stopp der Prozessausführung im gesamten System werden daher durch die VM mit den meisten virtuellen Knoten bestimmt. Dies ist in Formel (4.9) dargestellt. $\kappa_{suspend}$ stellt dabei eine Konstante dar, die stark von der benutzten Hardware und Software abhängt. n_v bezeichnet die Anzahl der virtuellen Knoten in der virtuellen Maschine vm. Also $n_{vm} = \mid \{n \in N | \phi(n) = vm\}$

Die Einheit von $\theta_{suspend}$ ist Millisekunden.

(4.9) $\theta_{resume} = \max_{vm \in VM}(n_{vm}) * \kappa_{resume}$

Die Zeit, die für die Wiederaufnahme der Prozessausführung benötigt wird, ergibt sich analog zur obigen Formel. Genau wie bei der komplementären Operation wird hier ein Signal an den Prozess geschickt: das CONT Signal. Die Formel zur Berechnung ist in (4.9) dargestellt. n_{vm} steht für die Anzahl der Knoten, die sich in der neuen Platzierung auf der virtuellen Maschine vm befinden. Also $n_{vm} = \mid \{n \in N | \phi'(n) = vm\}$

4.5.2 Migration von virtuellen Knoten und Netshaper Instanzen

Die Migration von virtuellen Knoten und Netshaper Instanzen lässt sich in 4 Teiloperationen unterteilen.

- Sichern aktueller Zustände (dump)
- Transfer der gesicherten Daten (transfer)
- Entfernen virtueller Knoten in Quell VM (killVNodes)
- Wiederherstellen der Zustände (undump)

Die Gesamtkosten für die Migration sind durch (4.10) gegeben.

(4.10) $\theta_{migration} = \theta_{dumpNs} + \theta_{dumpVNodes} + \theta_{killVNodes} + \theta_{transferData} + \theta_{undumpNs} + \theta_{undumpVNode}$

Auf die einzelnen Anteile der Kosten wird nun näher eingegangen.

Sichern der Zustands virtueller Knoten

Der Zustand eines virtuellen Knotens umfasst im Wesentlichen Folgendes: Den Adressraum, genutzte Registersätze, File und Signalhandler, Timer, User und Prozess Identitäten sowie den von Prozessen genutzter Speicher.

Die Zeit, die für die Erstellung einer Sicherung (in Form einer Datei) benötigt wird, steigt linear mit der Größe der Datei. Die Größe kann gut durch den gerade genutzten Arbeitsspeicher abgeschätzt werden.

Formel (4.11) zeigt die Kosten, die für die Sicherung der Zustände aller zu migrierenden Knoten entstehen. Jeder Knoten wird einzeln über das OpenVZ Control Tool gesichert. Jeder Aufruf verursacht dabei gewisse Grundkosten $c_{dumpVnode}$. Virtuelle Knoten, die sich in unterschiedlichen virtuellen Maschinen befinden, können zeitgleich gesichert werden. Wodurch die Gesamtlaufzeit für die Erstellung der Sicherungsdateien durch die virtuelle Maschine mit den höchsten Kosten gegeben ist.

M steht für die Menge der zu migrierenden virtuellen Knoten ($M = \{n \in N | \phi(n) \neq \phi'(n)\}$)und ς^m_{mem} kennzeichnet die Größe des vom Knoten m genutzten Arbeitsspeichers.

$$\theta_{dumpVNodes} = \max_{vm \in VM} \left(\sum_{m \in M \ ,\phi(m)=vm} \left(\kappa_{dumpMem} * \varsigma^m_{mem} + c_{dumpVnode} \right) \right) \tag{4.11}$$

Sichern der Zustände von Netshaper Instanz

Die Sicherung des Netshaper Zustands umfasst folgende Daten: Einstellungen wie Bandbreite, Verlustrate und Verzögerung, eine Parameterliste, die Verbindungseigenschaften virtueller Knoten definiert und Frames, die sich derzeit noch im Netshaper Puffer befinden. Maßgeblich für die Zeit, die für die Sicherung benötigt wird, ist die Gesamtgröße der Frames, die sich noch im Puffer befinden sowie die Größe der Parameterliste.

Jeder Netshaper besteht aus 2 Instanzen: eine für die Sende- und eine für die Empfangsrichtung. In der Netshaper Instanz der Senderichtung befindet sich in der Regel im Puffer nur maximal ein Frame. Muss ein Frame verzögert ausgeliefert werden, so wird höheren Schichten mitgeteilt, dass gerade keine Ressourcen zur Verfügung stehen. Diese stellen daraufhin das Senden ein.

In der anderen Instanz können sich sehr viele Frames befinden. Dies hängt im Wesentlichen von der Senderate der virtuellen Knoten und der Dauer der Suspend Operation ab. Sind höhere Schichten nicht verfügbar, z.B. weil sich der virtuelle Knoten im Haltezustand befindet, so werden eingehende Frames im Netshaper zwischengespeichert.

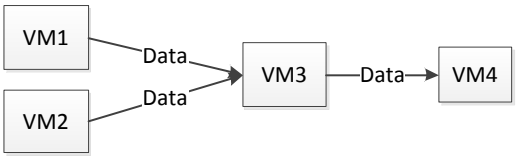


Abbildung 4.10: Beispiel Kosten Datentransfer

Formel (4.12) zeigt die Kosten, die für die Sicherung aller Netshaper Instanzen zu migrierender Knoten entstehen. Dabei steht NS für die Menge aller Netshaper Instanzen und das Prädikat $belongsTo$ gibt an, ob eine Netshaper Instanz ns mit einer virtuellen Netzwerkkarte eines Knotens m verbunden ist.

$$\theta_{dumpNs} = \max_{vm \in VM} \left(\sum_{\substack{ns \in NS \\ belongsTo(ns,m) \\ \phi(m)=vm}} \left(\kappa_{dumpMac} * \varsigma^{ns}_{macs} + \kappa_{dumpFrame} * \varsigma^{ns}_{frames} + c_{dumpNs} \right) \right) \tag{4.12}$$

Die Kosten für die Sicherung einer Netshaper Instanz ergeben sich aus den Kosten für das Speichern der Parameterliste ($\kappa_{dumpMac} * \varsigma^{ns}_{macs}$) und dem Sichern der zwischengespeicherten Frames $\kappa_{dumpFrame} * \varsigma^{ns}_{frames}$. Außerdem ist jeder Aufruf für die Sicherung einer Netshaper Instanz noch mit einem gewissen Grunkosten verbunden ($c_{dumpNetshaper}$).

Transfer der Daten

Nachdem der Zustand virtueller Knoten und Netshaper Instanzen gesichert wurde, müssen die erzeugten Daten zu den Ziel-VMs übertragen werden. Die Zeit, die dabei benötigt wird, steigt linear mit der Größe der Daten. Jede virtuelle Maschine kann Daten empfangen und Daten senden.

Die Menge an Daten, die eine virtuelle Maschine $vm1$ an eine andere virtuelle Maschine $vm2$ überträgt, ergibt sich durch die Formel (4.13).

$$\varsigma^{vm1,vm2} = \sum_{\substack{ns \in NS, m \in M \\ belongsTo(ns,m) \\ \phi(m)=vm1, \phi'(m)=vm2}} \varsigma^{ns}_{dumpNetshaper} + \sum_{\substack{m \in M \\ \phi(m)=vm1, \phi'(m)=vm2}} \varsigma^{m}_{dumpVNode} \tag{4.13}$$

Dabei ist $\varsigma^{ns}_{dumpNetshaper} \approx \varsigma^{ns}_{frames} + \varsigma^{ns}_{macs}$ und $\varsigma^{m}_{dumpVNode} \approx \varsigma^{m}_{mem}$.

Das zu übertragende Datenvolumen umfasst also alle Sicherungsdateien von Netshapern und virtuellen Knoten, die sich in der neuen Platzierung ϕ' nun auf der virtuellen Maschine $vm2$ befinden.

Ist die Größe der auszutauschenden Daten zwischen zwei virtuellen Maschinen gegeben, so wird die Zeit, die für die Übertragung benötigt wird, anhand der Formel (4.14) berechnet.

$$\theta^{vm1,vm2}_{transferData} = \varsigma^{vm1,vm2} * \kappa + c_{transferData} \tag{4.14}$$

Der Wert von κ ist dabei von der Art der Verbindung zwischen den virtuellen Maschinen $vm1$ und $vm2$ abhängig. Befinden sich beide VMs auf dem gleichen physikalischen Knoten, so werden Daten über das Host-Os ausgetauscht. Befinden sie sich allerdings auf unterschiedlichen physikalischen Rechnern, so müssen die Daten über ein physikalisches Netzwerkgerät verschickt werden. Dadurch entstehen höhere Kosten.

Die Gesamtkosten für den Transfer aller Daten ist durch die Formel (4.15) gegeben.

$$\theta_{transferData} = \max_{vm \in VM} \left(\sum_{vm=vm1 \vee vm=vm2} \theta^{vm1,vm2}_{transferData} \right) \tag{4.15}$$

Für jede VM werden die Kosten für den Austausch von Daten mit anderen VMs aufsummiert. Die virtuelle Maschine mit dem höchsten Kosten legt dabei die benötigte Zeit fest.

Abbildung 4.10 zeigt ein Beispiel in dem virtuelle Maschinen Daten austauschen. Die Zeit, die dabei die virtuelle Maschine $VM3$ für die Kommunikation aller Daten benötigt, ergibt sich aus der Zeit für den Empfang von Daten der Maschine $VM1$, der Zeit für den Empfang von Daten von $VM2$ und der Zeit für das Senden von Daten an die Maschine $VM4$.

Dabei hat die Anzahl der virtuellen Maschinen, mit der die virtuelle Maschie $VM3$ gleichzeitig Daten austauscht, keinen Einfluss auf die Gesamtlaufzeit. Bestehen gleichzeitig mehrere Datenverbindungen zu unterschiedlichen VMs, so teilen sich alle die zur Verfügung stehenden Ressourcen.

Wiederherstellen der Zustände virtueller Knoten

Formel (4.16) zeigt die Kosten für die Wiederherstellung aller migrierten virtuellen Knoten. Sie ist analog zu Formel (4.11).

$$\theta_{undumpVNode} = \max_{vm \in VM} \left(\sum_{m \in M \,,\phi'(m)=vm} \left(\kappa_{undumpMem} * \varsigma^{m}_{mem} + c_{undumpVnode} \right) \right) \tag{4.16}$$

Wiederherstellen der Zustände von Netshaper Instanzen

Formel (4.17) zeigt die Kosten für den Undump aller migrierten Netshaper Instanzen. Sie ist analog zu Formel (4.12).

(4.17)

$$\theta_{undumpNs} = \max_{vm \in VM} \left(\sum_{\substack{ns \in NS, m \in M \\ belongsTo(ns,m) \\ \phi'(m)=vm}} \left(\kappa_{undumpMac} * \varsigma^{ns}_{macs} + \kappa_{undumpFrame} * \varsigma^{ns}_{frames} + c_{undumpNs} \right) \right)$$

Entfernen virtueller Knoten in Quell VMs

Wurde der Zustand eines virtuellen Knotens zum Zielrechner transferiert, wird dieser auf dem Quellrechner nicht mehr benötigt und kann beendet werden. Die Zeit, die für diese Operation benötigt wird, ist von der Anzahl der zu beendenden virtuellen Knoten abhängig und von der Zahl der virtuellen Netzwerkgeräten eines Knotens (überraschenderweise). Da ein virtueller Knoten in der Regel nur eine konstante, sehr geringe Anzahl an Netzwerkgeräten hat, werden Netzwerkgeräte hier vernachlässigt.

In Formel (4.18) sind die Kosten für das Entfernen eines Knotens gegeben.

(4.18) $\theta_{killVNodes} = \max_{vm \in VM} (m^{vm}) * \kappa_{killVNode} + c_{kill}$

Dabei steht m^{vm} für die Knoten, die sich in der neuen Platzierung ϕ' nicht mehr auf der virtuellen Maschine vm befinden. Also $m^{vm} = |\{n \in N | \phi(n) = vm \wedge \phi'(n) \neq vm\}|$.

4.5.3 Anpassung Layer 2 Topologie

Für die Anpassung der Layer 2 Topologie können folgende Aktionen nötig sein.

- Erzeugung einer Softwarebrücke und Einrichtung eines Vlans
- Löschen einer Softwarebrücke und Entfernen des Vlans
- Anbindung einer virtuellen Netzwerkkarte an eine Softwarebrücke

Werden virtuelle Knoten migriert so müssen ihre virtuellen Netzwerkkarten in der Ziel VM an die richtigen Brücken angebunden werden. Sind diese noch nicht vorhanden, müssen sie erstellt werden. Das Anhängen einer Netzwerkkarte an eine Brücke ist mit geringen Kosten verbunden. Da ein virtueller Knoten in der Regel nur über wenige Netzwerkkarten verfügt, werden diese Kosten hier nicht betrachtet.

Das Löschen und Erzeugen von Softwarebrücken sind im Wesentlichen zeitkonstante Operationen. Die Zeit, die für die Anpassung der Layer 2 Topologie benötigt wird, ergibt sich durch Formel (4.19)

$$\theta_{layer2adaption} = \max_{vm \in VM} \left(n^{vm}_{create} * \kappa_{createBridge} + n^{vm}_{destroy} * \kappa_{destroyBridge} \right) \tag{4.19}$$

Dabei steht n^{vm}_{create} für die Anzahl der zu erzeugenden Brücken in der virtuellen Maschine vm und $n^{vm}_{destroy}$ für die Anzahl der zu löschenden Brücken in der virtuellen Maschine vm. Die Kosten für das Einrichten und Entfernen eines Vlans sind in den Kosten für das Erstellen und Zerstören einer Softwarebrücke enthalten.

4.6 Optimierung der Platzierung

Einen wesentlichen Bestandteil der dynamischen Neuplatzierung stellt die Ermittlung einer günstigeren Platzierung dar. An diese werden bestimmte Anforderungen gestellt. Sie muss einen möglichst hohen Nutzen haben (also die Experimentlaufzeit verkürzen), darf aber für die Umsetzung nicht zu viele Kosten verursachen. Es gilt eine Platzierung zu finden, die die in Abschnitt 4.6.1 vorgestellte Zielfunktion maximiert.

Von einer optimalen Platzierung spricht man, wenn deren Zielfunktionswert maximal ist. Falls möglich, gilt es, diese optimale Platzierung zu finden.

Ein Problem diese optimale Lösung zu finden stellt der sehr große Suchraum dar. Dieser steigt exponentiell mit der Anzahl der Knoten $O(|VM|^{|N|})$. Der Rechenaufwand, der für die Ermittlung der optimalen Platzierung benötigt wird, ist daher sehr hoch. Aufgrund des dynamischen Verhaltens des Testsystems steht allerdings nur begrenzt Rechenzeit zur Verfügung. Nehmen Berechnungen zu viel Zeit in Anspruch, können sie wertlos werden, da sich die aktuelle Lastsituation bereits geändert haben kann.

Es wird daher ein Optimierungsverfahren benötigt, dass mit geringem Rechenaufwand eine möglichst günstige Platzierung findet. Aus der Literatur sind z.B. folgende Optimierungsverfahren bekannt:

- Simulated Annealing
- Evolutionäre Algorithmen
- Hill Climbing

Beim Simulated Annealing handelt es sich um eine Optimierungsverfahren, das z.B. für das Floor Planning beim Entwurf von Chips eingesetzt wird. Es gehört zu der Gruppe der naturanalgonen Optimierungsverfahren und beruht auf der Nachbildung des aus der Metallurgie bekannten Abkühlungsprozesses. Durch kontrolliertes Abkühlen soll, z.B. beim Glühen, eine Maximierung der Kristallgröße erreicht werden.

Beim Simulated Annealing Ansatz wird mit einer beliebigen Lösung gestartet. Von dieser ausgehend wird zufällig eine ähnliche Lösung bestimmt und bewertet. Ist diese besser als die vorige Lösung, wird mit ihr fortgefahren. Ist sie schlechter , wird sie dennoch mit einer bestimmten Wahrscheinlichkeit weiterverfolgt.

Die Wahrscheinlichkeit, mit der beim Simulated Annealing bergab gegangen wird, wird durch den Kontrollparameter T beeinflusst. Dieser ist als Analogon zur Temperatur beim Abkühlungsprozess zu sehen. Je kleiner der Kontrollparameter, desto unwahrscheinlicher ist es, dass schlechtere Lösungen weiterverfolgt werden. Genau wie die Temperatur im Abkühlungsprozess wird der Wert des Kontrollparameters mit fortschreitender Zeit verringert.

Die anfängliche Verfolgung vermeintlich schlechter Lösungen birgt eine Möglichkeit in sich, lokale Optima zu überwinden, und so mit einer höheren Wahrscheinlichkeit das globale Optimum zu erreichen. Die Laufzeit ist bei diesem Verfahren sehr stark von dem sogenannten *Cooling Schedule* abhängig. Dieser gibt die Veränderung des Kontrollparameters über die Zeit an.

Bei evolutionären Algorithmen handelt es sich um Optimierungsverfahren, die sich an der biologischen Evolution orientieren. Sie beruhen auf aus der Natur bekannten Mechanismen wie Reproduktion, Mutation, Rekombination und Selektion. Mögliche Lösungen werden als Individuen gesehen. Diese müssen sich zusammen mit anderen in einer bestimmten Umgebung behaupten. Eine Fitnessfunktion gibt dabei an, wie gut sie an die Umgebung angepasst sind. Durch die oben genannten Mechanismen sollen Individuen erzeugt werden, deren Fitnessfunktionen möglichst optimale Werte aufweisen.

Der Hill Climbing Algorithmus ist im Wesentlichen eine lokale Greedy Suche. Gestartet wird mit einer zufälligen Lösung. Von dieser Lösung aus wird mit der Nachbarlösung fortgefahren, welche den höchsten Zielfunktionswert aufweist. Im Gegensatz zum Simulated Annealing wird bei diesem Verfahren nur bergauf gegangen. Außerdem wird nicht zufällig eine ähnliche Lösung ausgewählt, sondern alle ähnlichen Lösungen betrachtet. Werden als ähnliche Lösungen, z.B. Lösungen definiert, die sich in der Position genau eines virtuellen Knotens unterscheiden, so sind dies bereits $O(|VM| * |N|)$. Für jede dieser Lösungen muss der Zielfunktionswert berechnet werden, was das Verfahren sehr aufwändig macht.

Hill-Climbing erfordert einen hohen Rechenaufwand pro Iterationsschritt. Beim Simulated Annealing ist der Rechenaufwand pro Iteration eher gering. Je nach Temperaturfunktion werden allerdings zunächst auch schlechtere Lösungen verfolgt. Bei Evolutionären Algorithmen hängt der Aufwand sehr stark von der Ausprägung der einzelnen Operationen ab.

In dieser Diplomarbeit soll der Simulated Annealing Ansatz weiterverfolgt werden. Zur Optimierung einer Platzierung wird in [GHR] ein sehr ähnlicher Ansatz verfolgt. In diesem werden allerdings nur bergauf gegangen. Laut [GHR], konvergiert der Algorithmus in kurzer Zeit gegen ein Optimum.

4.6.1 Zielfunktion

Das Ziel der dynamischen Neuplatzierung ist die Verkürzung der Experimentlaufzeit. Für unterschiedliche Platzierung kann die erwartete Experimentlaufzeit θ_{real} für ein virtuelles Zeitintervall $\theta_{virtual}$ (siehe 4.4) abgeschätzt werden. Um Doppeldeutigkeiten zu vermeiden, wird im Folgenden θ_{real} zu θ_{comm} umbenannt.

Für die Umsetzung einer Platzierung entstehen Rekonfigurationskosten. Diese können durch das in Abschnitt 4.5 vorgestellte Kostenmodell abgeschätzt werden. Für eine alternative Platzierung ϕ' entstehen damit die in (4.20) vorgestellten Kosten für ein bestimmtes virtuelles Zeitintervall $\theta_{virtual}$.

$$\theta_{real}^{\phi',\theta_{virtual}} = \theta_{comm}^{\phi',\theta_{virtual}} + \theta_{reconfiguration}^{\phi,\phi'} \tag{4.20}$$

Die Kosten einer alternativen Platzierung erhöhen sich also um die Kosten für die Umsetzung der Platzierung.

Trägt man die erwarteten Experimentlaufzeiten θ_{real} der aktuellen Platzierung ϕ und einer möglichen alternativen Platzierung ϕ' über die virtuelle Zeit $\theta_{virtual}$ ab, so ergibt sich beispielsweise das in 4.11 dargestellte Diagramm. Eine alternative Platzierung, die weniger Kommunikationskosten verursacht, lohnt sich erst nach einer bestimmten virtuellen Zeit. Dies ist durch die Rekonfigurationszeit bedingt.

$\theta_{virtual}$ kann als Zeitfenster gesehen werden, das in die Zukunft geblickt werden soll. Für dieses Zeitfenster müssen sich sinnvolle Prognosen zu Lasten und Datenraten bestimmen lassen, da auf diesen Daten das Kommunikationskostenmodell basiert. Dieses, bei der Optimierung betrachtete virtuelle Zeitfenster, soll nachfolgend θ_{window} genannt werden.

Auf Basis der Kosten für die aktuelle und eine alternative Platzierung kann die in (4.21) dargestellte Zielfunktion definiert werden.

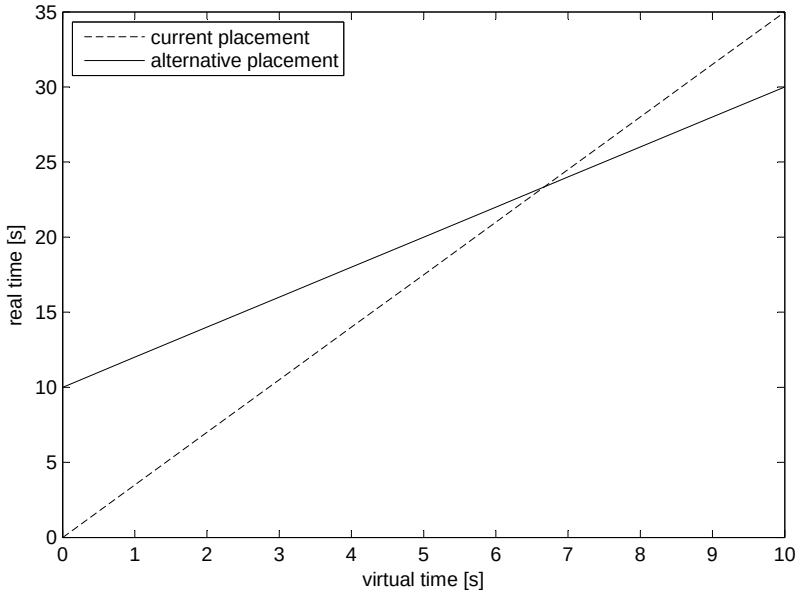


Abbildung 4.11: Erwartete Laufzeiten zweier Platzierungen

$$quality(\phi') = \theta_{comm}^{\phi,\theta_{window}} - \theta_{real}^{\phi',,\theta_{window}} \tag{4.21}$$

Diese stellt die Differenz der erwarteten Experimentlaufzeit der aktuellen Platzierung ϕ und einer alternative Platzierung ϕ' dar. Ist $quality(\phi') > 0$, so ist die Platzierung ϕ' günstiger als die aktuelle Platzierung.

Größe des Vorhersage Zeitfensters

Die Größe des Zeitfensters t_{window} ist entscheidend für die Performance der Neuplatzierung.

Sinnvollerweise sollte es größer sein als die Rekonfigurationsfixkosten($t_{suspend} + t_{resume}$) und kleiner als die Zeit, für die noch sinnvolle Prognosen für Last und Datenraten möglich sind.

4.6.2 Optimierungsalgorithmus

In diesem Abschnitt soll nun näher auf eine mögliche Umsetzung des Simulated Annealing Ansatzes zur Optimierung der Platzierung eingegangen werden.

Ziel der Optimierung ist die Maximierung der Zielfunktion $quality(\phi)$. Diese wurde in 4.6.1 vorgestellt.

Eine Möglichkeit dies mittels eines Simulated Annealing Ansatzes zu tun, ist als Pseudocode in 4.1 dargestellt.

Gestartet wird mit der aktuellen Platzierung. Von dieser ausgehend wird eine ähnliche Platzierungen betrachtet. Diese wird zufällig aus einer Menge von ähnlichen Platzierungen ausgewählt. Die Menge dieser Platzierungen ist durch die Funktion *like* definiert. Diese bildet eine Platzierung ϕ auf eine Untermenge ASS' der möglichen Platzierungen ASS ab. $like : \phi \mapsto ASS'$. Möglichkeiten diese Menge zu definieren, werden in 4.6.3 vorgestellt.

Algorithmus 4.1 Algorithmus zur Optimierung einer Platzierung

```
1:  t ← initialValue
2:  while ¬exitCondition do
3:      nextAssignment ← random(like(currentAssignment))
4:      if quality(nextAssignment) > quality(currentAssignment) then
5:          currentAssignment ← nextAssignment
6:      end if
7:      if e^(−(quality(nextAssignment)−quality(currentAssignment))/T_t) > random() then
8:          currentAssignment ← nextAssignment
9:      end if
10:     t ← t + 1
11: end while
12: return currentAssignment
```

Weist die zufällig ausgewählte Platzierung (*nextAssignment*) einen höheren Zielfunktionswert ($quality(nextAssignment)$) als die aktuelle Platzierung (*currentAssignment*) auf, so wird sie zur aktuellen Platzierung.

Ist dies nicht der Fall, wird sie trotzdem mit einer bestimmten Wahrscheinlichkeit, die durch $e^{-(value-quality(currentAssignment))/T(t)}$ gegeben ist, akzeptiert.

Die Wahrscheinlichkeit, mit der Bergab gegangen wird, hängt von dem Kontrollparameter T_t und der Zielfunktionsnähe beider Platzierungen ab. Hat der Kontrollparameter einen hohen Wert, so ist die Wahrscheinlichkeit hoch, dass eine schlechtere Platzierung verfolgt wird. Unterscheidet sich die Qualität der aktuellen und der möglichen neuen Platzierung stark, so ist die Wahrscheinlichkeit niedrig, dass die neue Platzierung übernommen wird. Mögliche Verläufe des Kontrollparameters werden in Abschnitt 4.6.7 vorgestellt.

Der Algorithmus terminiert wenn eine bestimmte Abbruchbedingung erfüllt ist. Mögliche Abbruchbedingungen werden in 4.6.6 diskutiert.

4.6.3 Ähnliche Platzierungen

Ähnliche Platzierungen sind Platzierungen, die sich in der Position weniger virtueller Knoten unterscheiden. Ist eine Platzierung ϕ gegeben, so kann die Menge der Platzierungen, die sich von ϕ in der Position genau eines virtuellen Knotens unterscheiden, z.B. folgendermaßen definiert werden:

$$like(\phi) = \{\phi' | \exists x \in N(phi(x) \neq phi'(x) \wedge \forall y(x \neq y \Rightarrow phi(x) = phi'(y)))\}$$

Die Anzahl der Unterschiede zweier Platzierung kann als Distanz der Platzierungen gesehen werden. Im obigen Beispiel ist die Distanz 1 für die Platzierung ϕ zu allen Platzierungen der Menge *like*. Lässt man höhere Distanzen zu, so steigt die Anzahl der durch like definierten Menge exponentiell. Es gilt $|like(\phi)| = O\left(\binom{N}{D} * VM^D\right)$. Wobei VM für die Anzahl der virtuellen Maschinen, D für die maximale Distanz und N für die Anzahl der virtuellen Knoten steht.

Um lokale Optima zu überwinden, kann es sinnvoll sein, weniger ähnliche Platzierungen zuzulassen, also die maximale Distanz höher zu wählen. Wählt man sie allerdings zu hoch, so artet der Simulated Annealing Algorithmus zu einer Random Suche aus. In der aktuellen Implementierung wird eine Distanz von 1 benutzt. Andere Distanzen könnten allerdings evaluiert werden.

4.6.4 Berechnung des Zielfunktionswerts

In der Optimierungsphase des Neuplatzierungsalgorithmus werden verschiedene alternative Platzierungen betrachtet. Für jede dieser Platzierungen muss der Wert der zu optimierenden Zielfunktion berechnet werden. Die in 4.6.1 vorgestellte Zielfunktion, basiert auf Kommunikations- und Rekonfigurationskosten. Werden diese für jede betrachte Platzierung neu berechnet, so entstehen hohe Kosten.

Für die Berechnung der Kommunikationskosten wird beispielsweise $O(|links|)$ Zeit benötigt. D.h. die Zeit steigt linear mit der Anzahl von Verbindungen zwischen virtuellen Knoten. Dies ist gerade bei große Szenarien mit vielen Links problematisch. Da dem Optimierungsalgorithmus nur ein bestimmtes Zeitfenster zur Ermittlung einer besseren Platzierung zur Verfügung steht, sollten Kosten für die Berechnung des Zielfunktionswerts möglichst klein sein.

Wirft man einen Blick auf den in 4.6.2 vorgestellten Optimierungsalgorithmus, so stellt man fest, dass sich aufeinander folgende Platzierungen kaum unterscheiden. In jedem Schritt wird eine Platzierung aus der Nachbarschaft der zuletzt betrachteten Platzierung ausgewählt.

Statt einer kompletten Neuberechnung der Kosten ist es daher sinnvoller die Kosten der aktuell betrachteten Platzierung als Inkrement der Kosten der vorher betrachten Platzierung zu sehen. In diesem Fall müssen nur Kostenänderungen berechnet werden. Dieses Vorgehen soll am Beispiel des Kommunikationskostenmodells im Folgenden näher erläutert werden.

Angenommen ϕ_i sei die zuletzt betrachtete Platzierung und ϕ_{i+1} eine neue Platzierung für die Kommunikationskosten berechnet werden sollen, dann lässt sich die Last einer virtuellen Maschine vm unter der neuen Platzierung ϕ_{i+1} wie in Formel (4.22) berechnen:

$$\Lambda_{p,c}^{vm,\phi_{i+1}} = \Lambda_{p,c}^{vm,\phi_i} + \Delta\Lambda_{p,c}^{vm} \tag{4.22}$$

Die Last der virtuellen Maschine vm ergibt sich also zum einen aus der Last, der zuletzt betrachteten Platzierung ϕ_i, die in der virtuellen Maschine entstanden ist und zum anderen aus der Laständerung $\Delta\Lambda_{p,c}^{vm}$. Analog dazu kann auch die Last eines Host-OS $\Lambda_{p,c}^{VM-Host-OS,\phi i+1}$ dargestellt werden.

Im Folgenden soll nun vorgestellt werden wie sich $\Delta\Lambda_{p,c}^{vm}$ berechnen lässt.

M sei eine Menge von virtuellen Knoten $i \in N$, für die sich die Platzierung geändert hat $M = \{j \in N | \phi_{i+1}(j) \neq \phi_i(j)\}$. Zur Vereinfachung nachfolgender Formeln sei außerdem die Gleichung (4.23) gegeben.

$$\kappa_{p,p',c,c'}^{t} = \begin{cases} \kappa_{intra-vm,t} & \text{falls } p = p' \wedge c = c' \\ \kappa_{inter-vm,t} & \text{falls } p = p' \wedge c \neq c' \\ \kappa_{inter-pNode,t} & \text{falls } p \neq p' \wedge c \neq c' \end{cases} \tag{4.23}$$

Dann berechnet sich $\Delta\Lambda_{p,c}^{vm}$ durch die Formel (4.24).

$$\Delta\Lambda_{p,c}^{vm} = \Delta\Lambda_{p,c}^{VM_{tx}} + \Delta\Lambda_{p,c}^{VM_{rx}} + \sum_{\substack{k \in M \\ \phi_{i+1}(k)=(p,c)}} \lambda_k - \sum_{\substack{k \in M \\ \phi_i(k)=(p,c)}} \lambda_k \tag{4.24}$$

Die Änderung der Last einer virtuellen Maschine ist gegeben durch die Änderung von Lasten für Verbindungen zwischen virtuellen Knoten und durch die Änderungen von Lasten, die durch virtuelle Knoten in der VM verursacht werden. Erhält die virtuelle Maschine in der neuen Platzierung ϕ_{i+1} ein neuen virtuellen Knoten k, so wird sie nun mit λ_k belastet. Verliert sie einen virtuellen Knoten k, so sinkt die Belastung um λ_k.

Laständerungen, die sich in einer virtuellen Maschine durch Verbindungsänderungen ergeben, lassen sich beispielsweise durch die Formel (4.25) berechnen. Diese zeigt Laständerungen

in einer VM für eingehende Verbindungen. Laständerungen für ausgehende Verbindungen lassen sich analog dazu bestimmen.

$$\Delta\Lambda_{p,c}^{VM_{rx}} = - \sum_{\substack{j,k \in N \\ j \in M \vee k \in M \\ \phi_i(j)=(p',c') \\ \phi_{i+1}(k)=(p,c)}} \beta_{jk} * \kappa_{p,p',c,c'}^{VM_{rx}} + \sum_{\substack{j,k \in N \\ j \in M \vee k \in M \\ \phi_{i+1}(j)=(p',c') \\ \phi_{i+1}(k)=(p,c)}} \beta_{ik} * \kappa_{p,p',c,c'}^{VM_{rx}} \tag{4.25}$$

Für die Berechnung der Laständerung müssen Verbindungen betrachtet werden, bei denen mindestens einer der beiden Kommunikationspartner migriert wurde. Für jede dieser Verbindungen muss die Last angepasst werden. Hierzu werden Lasten , die unter der alten Platzierung ϕ_i berechnet wurden, subtrahiert und Lasten, die durch Verbindungen in der neuen Platzierung ϕ_{i+1} entstehen, addiert. $\Delta\Lambda_{p,c}^{VM_{tx}}$ ergibt sich analog dazu.

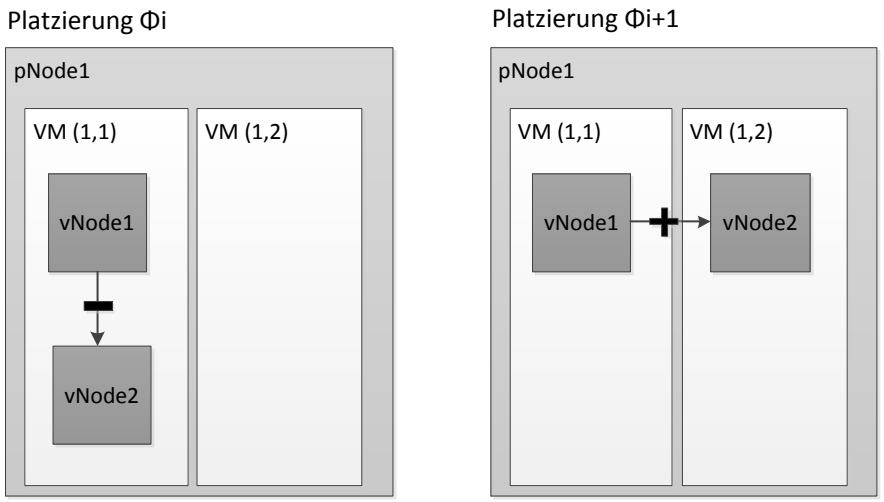


Abbildung 4.12: Beispiel zur Berechnung von Laständerungen

Zur Veranschaulichung der Formel betrachten wir nun das Beispiel in Abbildung 4.12. In diesem sind die Änderungen der Lasten dargestellt, die sich für eine Umplatzierung des Knotens $vNode2$ ergeben. In diesem Beispiel wird aus der $intra - vm$ Verbindung zwischen beiden Knoten, die nur die virtuelle Maschine $VM(1,1)$ belastet, eine $inter - vm$ Verbindungen. Diese verursacht eine Last auf beiden virtuelle Maschinen. Die Laständerung $\Delta\Lambda_{1,1}^{VM_{rx}}$ der virtuellen Maschine $VM(1,1)$ ergibt sich zu $-\beta_{1,2} * \kappa_{intra-vm,VM_{rx}} + \beta_{1,2} * \kappa_{inter-vm,VM_{rx}}$.

Sind alle Lasten virtueller Maschinen $\Lambda_{p,c}^{vm,\phi_{i+1}}$ und Lasten der Host Betriebsysteme bestimmt, lässt sich die erwartete Experimentlaufzeit bestimmen(siehe 4.4).

Bei dieser Variante wird für die Berechnung der Kommunikationskosten $O(l')$ Zeit benötigt. Wobei l' für die Anzahl der Verbindungen steht, bei denen mindestens einer der beiden Verbindungspartner migriert wurde. Ändert sich die Platzierung nur weniger Knoten von ϕ_i zu ϕ_{i+1}, so kann mit dieser Variante sehr viel Rechenzeit eingespart werden, da dann gilt $l' << l$.

4.6.5 Verkleinerung des Suchraums

Im Abschnitt 4.6.3 wurde die Menge ähnlicher Lösungen definiert. In jeder Iteration des Algorithmus wird daraus zufällig ein Element ausgewählt. Fällt die Wahl auf ein Element, dass die Zielfunktion nicht weiter minimiert, wird dieses mit einer bestimmten Wahrscheinlichkeit weiter berücksichtigt. Um eine schnellere Konvergenz des Optimierungsalgorithmus zu erreichen ist es sinnvoll Lösungen , die mit hoher Wahrscheinlichkeit nicht zu einer optimalen Lösung führen, heraus zu filtern.

Wie in Kapitel 4.4 erläutert, sind Kommunikationskosten zwischen Knoten stark abhängig von der Art der Verbindung zwischen ihnen. Wird ein Knoten zu einer virtuellen Maschine migriert, auf der sich kein Knoten befindet, mit dem er eine Verbindung eingeht, so ist die Wahrscheinlichkeit sehr hoch, dass sich die Kommunikationskosten erhöhen. In diesem Fall kann der Knoten nur $inter - vm$ und $inter - pnode$ Verbindungen zu anderen Knoten eingehen. Diese verursachen höhere Kosten als $intra - vm$ Verbindungen. Ähnliche Platzierungen, die durch die Migration genau der eben genannten Knoten entstehen, sollten herausgefiltert werden.

Neben den sich durch die Migration ändernden Verbindungen können auch Merkmale wie Anzahl der Prozesse und genutzten Arbeitsspeicher eines Knotens betrachtet werden. Nutzt ein Knoten viel Speicher, erzeugt aber nur eine geringe Kommunikationslast, so ist die Wahrscheinlichkeit gering, dass die Migration eines solchen Knotens die Platzierung verbessert. Es entstehen in diesem Fall hohe Migrationskosten, denen eher kleine Kommunikationskosteneinsparung gegenüber stehen.

4.6.6 Abbruchbedingung

In diesem Abschnitt sollen Abbruchbedingungen für den Simulated Annealing Algorithmus diskutiert werden.

Mögliche Abbruchbedingungen sind:

- Abbruch bei Konvergenz
- Abbruch nach einer bestimmten Anzahl von Iterationen
- Abbruch beim Überschreiten eines Zeitlimits

Eine häufig eingesetzte Möglichkeit einen Optimierungsalgorithmus zu beenden ist die Konvergenz. Ist von der aktuellen Lösung keine bessere Lösung mehr erreichbar, so terminiert der Algorithmus. Je nach Komplexität des Problems und der gewählten Startlösung kann die Laufzeit des Algorithmus dabei allerdings sehr unterschiedlich sein. Ist der Algorithmus zeitkritisch, sollte daher entweder nach einer festen Anzahl von Iterationen oder nach einem bestimmten Zeitlimit abgebrochen werden.

Die in 4.6.1 vorgestellte Zielfunktion beruht auf Prognosen zu Datenraten von Verbindungen und Auslastungen von Knoten. Diese sind nur für eine bestimmte Zeit gültig. Daher muss der Optimierungsalgorithmus nach einer gewissen Zeit abgebrochen werden. Zusätzlich kann der Algorithmus noch auf Konvergenz geprüft werden. Konvergiert dieser vor Ablauf des Zeitlimits, kann er beendet werden.

4.6.7 Cooling Schedule

Maßgeblich für die Performance eines Simulated Annealing Algorithmus ist der Cooling Schedule. Er wird spezifiziert durch:

- Den initialen Wert des Kontrollparameters (Temperatur)
- Änderung des Kontrollparameters über die Anzahl der Iterationen (Zeit)
- Den finalen Wert des Kontrollparameters (nicht unbedingt nötig)

In der Literatur wird zwischen dynamischen und statischen Cooling Schedules unterschieden.

Bei der statischen Variante werden Werte des Kontrollparameters vor der Ausführung des Algorithmus festgelegt. Eine Änderung der Werte ist zur Laufzeit nicht möglich.

Bei der dynamischen Variante hingegen werden Kontrollparameterwerte zur Laufzeit adaptiv angepasst. Zur Anpassung werden meist statistische Werte wie Mittelwerte und Standardabweichungen bisher errechneter Zielfunktionswerte verwendet. In der Literatur lassen sich viele unterschiedliche dynamische Cooling Schedules finden z.B. die von Huang [RSV91] und Lam [LJM88].

Ein dynamischer Schedule besitzt bessere Anpassungsmöglichkeiten an das Problem, ist allerdings auch mit zusätzlichen Kosten verbunden. Da dem Algorithmus zur Optimierung der Platzierung nur ein begrenztes Zeitfenster zur Verfügung steht und der Nutzen zusätzlicher Kosten nur schwer abgeschätzt werden kann, soll hier ein statischer Cooling Schedule verwendet werden.

Ein oft in der Praxis eingesetzter statischer Cooling Schedule ist der geometrische Schedule. Dieser wird z.B. im Simulated Annealing Algorithmus von Kirkpatrick [Kir84] genutzt.

Geometrischer Cooling Schedule

Der Simulated Annealing Algorithmus [Kir84] von Kirkpatrick gilt als die „Urversion“ der Simulated Annealing Algorithmen. Er basiert auf dem Metropolis [MRR$^+$53] Algorithmus.

Der Metropolis Algorithmus wurde ursprünglich zur Ermittlung von Eigenschaften wie Volumen und Druck von Substanzen bei einer bestimmten Temperatur eingesetzt. Im Metropolis Algorithmus werden dabei Substanzen durch eine Menge von Molekülen, die miteinander interagieren, modelliert.

Zur Ermittlung der Eigenschaften wird der Gleichgewichtszustand dieses Systems ermittelt. Im Metropolis Algorithmus werden energetische Zustände unterschiedlicher Molekül-Konfigurationen betrachtet. Die einzelnen Konfigurationen unterscheiden sich z.B. in der Lage einzelner Moleküle.

Kirckpatrick nutzte das Prinzip des Metropolis Algorithmus zur Lösung kombinatorische Optimierungsprobleme; Statt Molekülkonfigurationen werden Lösungen betrachtet; statt der Energie einer Konfiguration, der Wert einer Kosten-/Zielfunktion und statt der Temperatur wird ein Kontrollparameter T genutzt.

Genau wie im Metropolis Algorithmus werden Konfigurationen(Lösungen) bei einer bestimmten Temperatur (Wert des Kontrollparameters) betrachtet. Im Gegensatz zum Metropolis Algorithmus wird die Temperatur beim Simulated Annealing jedoch mit der Zeit verringert und damit ein Abkühlungsprozess simuliert.

Zuerst wird das zu optimierende System bei einer hohen Temperatur „geschmolzen“. Danach folgen Schritte, in denen das System stufenweise und kontrolliert abgekühlt wird.

Jede Temperatur sollte solange gehalten werden bis sich ein Gleichgewichtszustand einstellt.

Ein typischer Annealing Schedule nach Kirckpatrick ist in 4.13 dargestellt. Definiert werden müssen

- initiale Temperatur
- Änderung der Temperatur
- Zeit(Iterationen) konstanter Temperatur

initiale Temperatur Werte von T_0 sind extrem von der Skalierung der Zielfunktion abhängig. Nach Kirckpatrick sollte die initiale Temperatur so gewählt werden, dass Lösungen, die den Zielfunktionswert erniedrigen, mit einer Wahrscheinlichkeit von $p = 0.8$ akzeptiert werden. Um ein initialen Wert auszuwählen, der dieses Kriterium erfüllt, kann man z.B. einen Probelauf starten, in dem negative Änderungen des Zielfunktionswertes $diff^-$ aufgezeichnet werden. Mittels dieser lässt sich dann durch Formel (4.26) der Wert von T_0 bestimmen. Zur

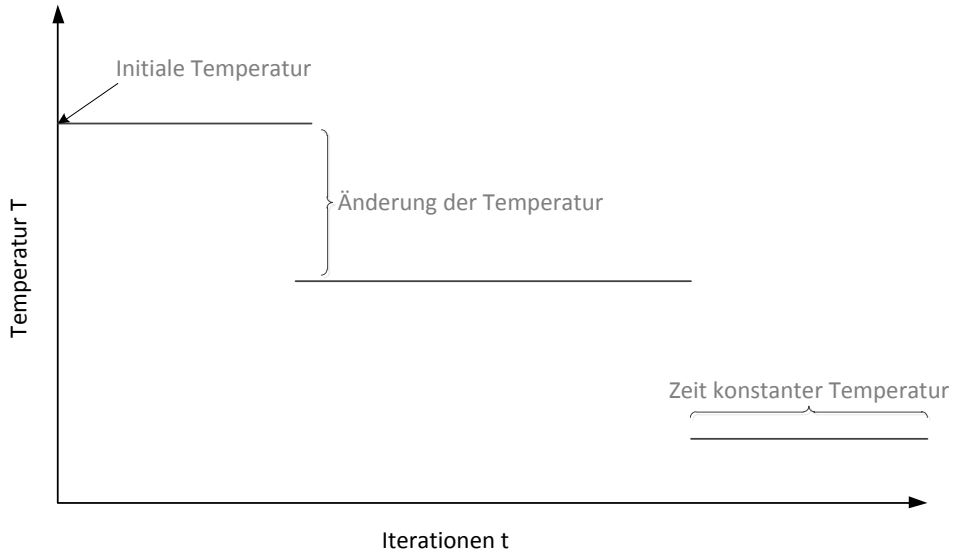


Abbildung 4.13: Typischer Cooling Schedule

Erinnerung: Ein Funktionswert, der den Wert der Optimierungsfunktion verringert, wird mit einer Wahrscheinlichkeit von $p = e^{\frac{\Delta f}{T}}$ akzeptiert.

$$T_0 = \frac{\overline{diff^-}}{ln(p = 0.8)} \tag{4.26}$$

Alternativ kann die Standardabweichung der Variation der Zielfunktion verwendet werden. Also $T_0 = \sigma_0$. Nach Hall [Whi84] ist dieses Vorgehen sehr effektiv.

Beides erfordert allerdings einen hohen Aufwand zur Bestimmung der initialen Temperatur. Daher soll hier eine einfachere, aber häufig in der Praxis verwendete Methode benutzt werden. Dabei wird die maximale Differenz der Zielfunktion abgeschätzt und T_0 dann auf diesen Wert gesetzt. Also $T_0 = \max(\Delta f)$

Wird die in Kapitel 4.6.1 vorgestellte Zielfunktion benutzt, kann die maximale Differenz, z.B. durch das betrachtete virtuelle Zeitfenster t_{window} abgeschätzt werden. Dies soll im folgenden Absatz kurz motiviert werden.

Ein virtueller Knoten benötigt mit hoher Wahrscheinlichkeit nicht mehr Ressourcen als ein Prozessor zur Verfügung stellen kann: also $usedCycles_{vNode} \leq offeredCycles_{cpu}$. Beim Übergang zu einer neuen Platzierung wird die Position weniger Knoten verändert(in der Regel nur die eines Knotens). Im schlimmsten Fall erhöht sich die Last in der Ziel-VM bei einer Migration eines Knotens um $offeredCycles_{cpu}$. Damit steigt der TDF maximal um 1

(z.b. wenn Ziel-VM höchst ausgelastete VM ist). Dadurch erhöht sich die erwartete Laufzeit maximal um t_{window}.

Hierbei handelt es sich natürlich nur um eine sehr grobe Abschätzung; in der Regel wird die maximale Differenz des Funktionswertes wesentlich kleiner sein. Die Evaluation des Optimierungsalgorithmus zeigt aber, dass sich auch mit dieser groben Abschätzung gute Ergebnisse erzielen lassen.

Zeit konstanter Temperatur Die Zeit bzw. die Zahl der Iterationen für die eine bestimmte Temperatur gehalten werden muss, hängt von der Größe des Problems ab. Meist wird die Anzahl der von einem Zustand aus erreichbaren Nachbarzustände als Richtwert herangezogen.

Um eine gewisse Veränderung zu erreichen, wird die Temperatur für eine bestimmte Anzahl von akzeptierten Veränderungen t_{accept} gehalten. Diese kann sich z.B. an der Zahl der Nachbarzustände orientieren. Gegen Ende der Optimierung werden in der Regel allerdings kaum noch Änderungen akzeptiert. Daher bedarf es häufig vieler Versuche um t_{accept} zu erreichen. Deswegen ist es sinnvoll eine Obergrenze für die mögliche Versuche einzuführen t_{max}. Typischerweise wird $t_{max} = 1.66 * t_{accept}$ gewählt.

Wie in 4.6.3 vorgestellt, hat eine Lösung bei unserem Optimierungsproblem $O(|N| * |VM|)$ ähnliche „Nachbar"-Lösungen (wenn nur die Position eines Knotens verändert wird). t_{accept} könnte daher z.B. wie folgt gewählt werden: $t_{accept} = |N| * |VM|$.

Funktion zur Absenkung der Temperatur Eine häufig eingesetzte, einfache Funktion zur Erniedrigung der Temperatur ist die (4.27) dargestellte exponentiale Funktion.

(4.27) $T_{k+1} = \alpha * T_k$

Dabei wird α aus dem Intervall $]0, 1[$ gewählt. Kirckpatrick schlägt ein günstigen Wert von $\alpha = 0.95$ vor.

4.7 Lastvorhersage

Mit Hilfe des Kommunikationskostenmodell kann für ein virtuelles Zeit $t_{virtual}$ die erwartete Experimentlaufzeit t_{real} berechnet werden. In der Optimierung ist dabei besonders das auf die aktuelle virtuelle Zeit folgende Zeitintervall interessant. Dies wurde in 4.6.1 als t_{window} bezeichnet. Um die erwartete Experimentlaufzeit für dieses Intervall möglichst

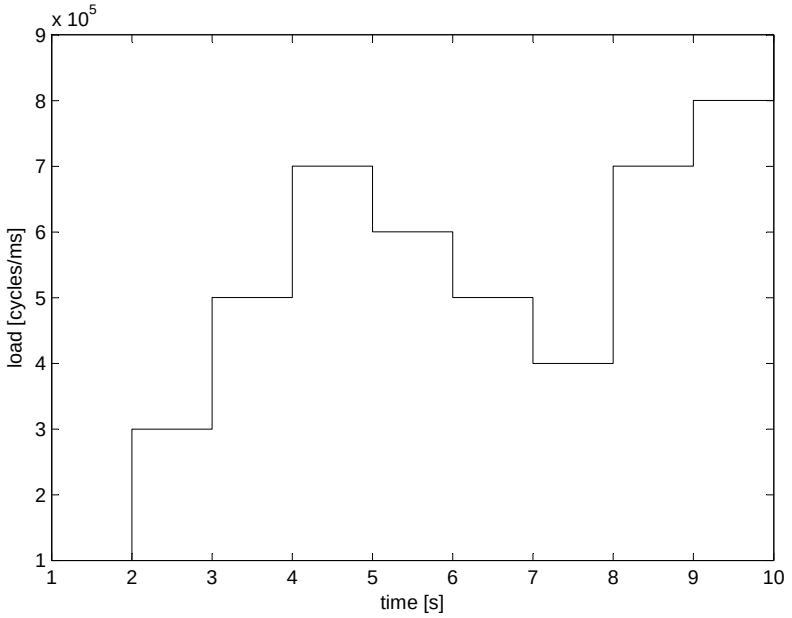


Abbildung 4.14: Historie der Last eines virtuellen Knoten

gut abschätzen zu können werden gute Prognosen der Last und Datenraten von Knoten benötigt.

Mithilfe von Monitoren können vergangene mittlere Lasten und Datenraten aufgezeichnet werden. Ein Beispiel für eine Historie von Lastwerten zeigt Abbildung 4.14. In dieser Grafik wurden mittlere Lasten in einem konstant breiten virtuellen Zeitintervallen von 1 Sekunde aufgezeichnet.

Auf Basis einer Historie müssen Lasten und Datenraten für das, bei der Optimierung betrachtete zukünftige virtuelle Zeitintervall t_{window}, abgeschätzt werden.

Dazu können z.B. Verfahren zur „one step load prediction“ eingesetzt werden. Für eine zeitlich aufeinander folgende Reihe von Messwerten $\{m1, m2, ..., m_n\}$, auch Zeitreihe genannt, lässt sich mit diesen der zu erwartetende, nächste Messwert m_{n+1} ermitteln.

In der Literatur werden zur Voraussage von m_{n+1} unterschiedliche Ansätze verfolgt. Ein grober Überblick wird in dem nachfolgenden Paragraph gegeben.

letzter Wert Bei diesem Ansatz wird $m_{n+1} = m_n$ gesetzt.

Tendenz basierte Vorhersage In [YFS03] wird eine Ansatz vorgestellt, der Tendenz basierte Vorhersage genannt wird. Auf Basis der letzten beiden Werte einer Zeitreihe, also m_n

und m_{n-1}, wird eine Tendenz der Messreihe abgeleitet $m_n - m_{n-1}$. Steigt der Wert der Messreihe $m_n - m_{n-1} > 0$, so wird $m_{n+1} = m_n + incValue$ gesetzt, andernfalls gilt $m_{n+1} = m_n - decValue$. Werte für $incValue$ und $decValue$ orientieren sich an Vorhersagefehlern und der Größe des Messwerte.

Untersucht wurden in [YFS03] Messreihen, bei denen Messpunkte mit einer Frequenz von 0.1 HZ, 0.05 HZ und 0.025 HZ aufgezeichnet wurden. Bei der niedrigsten Frequenz von 0.1 Hz lag die Abweichung der Voraussage vom tatsächlichen Wert bei unter 17 Prozent. Für ein Intervall von 10 Sekunden lässt sich also der nächste Messwert mit diesem Verfahren relativ gut abschätzen. Auffällig war, dass mit dem einfacheren „letzter Wert“ Verfahren in [YFS03] ähnliche Ergebnisse erzielt werden konnten.

Polynomial Fitting In [ZSI06] wird ein Ansatz für die „one step ahead prediction“vorgestellt, der auf Polinomial Fitting basiert. Beim Polinomial Fitting wird davon ausgegangen, dass Datenpaare (x_i, y_i)(z.B Zeitpunkt einer Messung und zugehöriger Messwert) korreliert sind.

Für die Korrelation wird eine Polynomfunktion $f(x) = y$ mit $f(x) = \sum_{i=0}^{N} A_i * x^i$ benutzt, wobei N den Grad der Funktion bezeichnet. Durch ein kleinstes Qudrate Fitting werden für eine Menge von Datenpaaren (x_i, y_i) die Konstanten A_i der Polynomfunktion bestimmt. Auf Basis dieser Funktion wird dann der nächste Wert der Messreihe abgeschätzt.

In [ZSI06] wurden Polynomfunktion von Grad 2 und 3 betrachtet. Laut den Messergebnissen von Thang sind mittlere Vorhersagefehler mit dem Polynomial Fitting Ansatz 38 bis 86 Prozent kleiner als bei dem Tendenz basierten Verfahren von Yang [YFS03].

Lineare Modelle Bei diesem Ansatz werden Zeitreihen als Realisierung stochastischer Prozesse angesehen, die einen linearen Filter steuern, der als Input weißes Rauschen erhält. Der lineare Filter hat dabei die Form $m'_i = \sum_{j=1}^{c} \omega_j * a_{i-j} + a_i$, wobei ω_j Parameter des Filters sind, a_j ein Wert der weißen Rausch Sequenz ist und c den Grad des Filters darstellt.

Die Parameter des Filters ω_j müssen so gewählt werden, dass die mittlere quadratische Abweichung der Modellwerte m'_i von den tatsächlichen Messwerten m_i minimal ist.

In [DO00] werden verschiedene lineare Modelle vorgestellt und bewertet: unter anderem Auto Regressive (AR) und Main Avarage (MA) Modelle, sowie Mischformen beider (ARMA). Einfache Auto Regressive Modelle mit einem Grad von 16 oder höher stellten sich als ausreichend für eine Vorhersage bei 1 HZ Daten(Messwerte in Abstand von einer Sekunde) bis zu 30 Sekunden in die Zukunft heraus.

Um eine möglichst gute Voraussage zu erreichen, sollten die vorgestellten Ansätze in realen Testszenarien auf ihre Tauglichtkeit geprüft werden. Dies war allerdings im Rahmen der

Diplomarbeit nicht möglich. Aufgrund des geringen Implementierungsaufwands wurde sich daher zunächst für den einfachsten „letzter Wert“ Ansatz entschieden.

4.8 Lage

In verteilten Systemen stellt sich immer die Frage nach dem Ort der Ausführung eines Algorithmus. Im Wesentlichen lassen sich 2 Ansätze unterscheiden: der verteilte und zentralen Ansatz.

Beim zentralen Ansatz wird der Algorithmus auf einem Rechner, meist Koordinator genannt, ausgeführt. Dieser Koordinator besitzt eine globale Sicht auf das System, was ihm ein hohes Optimierungspotential ermöglicht. Gleichzeitig ist für die Erstellung dieser globalen Sicht allerdings ein gewisser Aufwand nötig. Da ein Rechner nur über begrenzte Ressourcen verfügt, kann der Koordinator leicht zum Flaschenhals werden. Meist skalieren zentrale Ansätze deshalb nicht so gut wie verteilte.

Beim verteilten Ansatz wird der Algorithmus auf mehreren Rechner ausgeführt. Jeder Rechner hat dabei in der Regel nur eine beschränkte lokale Sicht auf das System. Das Optimierungspotential ist deshalb geringer als im zentralen Ansatz. Im Gegenzug entstehen aber auch keine Kosten für die Erstellung einer globalen Sicht, was diesen Ansatz skalierbarer macht.

4.8.1 Optimierung der Platzierung

Zentraler Ansatz

Beim zentralen Ansatz erfolgt die Optimierung der Platzierung zentral auf einem Rechner. Dabei kann z.B. für die Optimierung der in 4.6.2 vorgestellte Simulated Annealing Algorithmus verwendet werden. 4.2 zeigt in Pseudocode eine Skizze des zentralen Algorithmus.

Basis der Optimierung bildet die in 4.6.1 vorgestellte Zielfunktion. Diese fußt auf zwei Modellen: einem Kommunikationskostenmodell und einem Rekonfigurationskostenmodell. Diese müssen vor einer Optimierung aktualisiert werden. Dazu müssen dem Koordinator, z.B. Prognosen zu Lasten virtueller Knoten, Datenraten für Verbindungen und genutztem Speicher eines virtuellen Knotens zur Verfügung gestellt werden. Dies ist mit gewissen Kosten verbunden, die sehr stark von der Größe des jeweiligen Testszenarios abhängen.

Um diese Kosten möglichst niedrig zu halten, wird deshalb vorab der aktuelle Zustand des Systems auf seine Optimierbarkeit geprüft ($(isImprovablePlacement(currentPlacement))$). Dazu kann z.B. die Auslastung physikalischer Rechner des Testsystems herangezogen werden. Weichen Lasten der Rechner sehr stark voneinander ab, ist dies ein Indiz für eine ungünstige Platzierung. Da die Informationen, die für die Beurteilung der aktuellen Situation

herangezogen werden beschränkt sind, sollte in bestimmten Abständen eine Optimierung der Platzierung erzwungen werden.

Der Optimierungsalgorithmus liefert eine alternative Platzierung. Diese wird, bevor sie umgesetzt wird, geprüft. Da die Rekonfiguration der TVEE mit Kosten verbunden ist, muss abgewogen werden, inwieweit sich diese Investition lohnt. Ist nur mit geringen Laufzeiteinsparungen zu rechnen, so sollte die Platzierung nicht umgesetzt werden. Das Risiko wäre in diesem Fall zu hoch, da die für die Zukunft getroffenen Prognosen nicht eintreten müssen. Wurde sich für die Platzierung entschieden,, so wird die TVEE rekonfiguriert ($reconfigurateTVEE(alternativePlacement)$).

Algorithmus 4.2 Zentraler Neuplatzierungsalgorithmus

```
1:  currenPlacement ← initialPlacement
2:  loop
3:      wait(TDFscaledIntervall)
4:      if isImprovablePlacement(currentPlacement) then
5:          updateModel()
6:          alternativePlacement ← optimizePlacement(currentPlacement)
7:          if isBetterPlacement(alternativePlacement) then
8:              reconfigurateTVEE(alternativePlacement)
9:          end if
10:     end if
11: end loop
```

Der zentrale Algorithmus wird periodisch ausgeführt ($wait(TDFscaledIntervall)$), wobei die Dauer der Wartezeit mit dem aktuellen TDF skaliert wird. Dies ist nötig, da der Algorithmus auf einem Rechner aufgeführt werden soll, der in Echtzeit arbeitet.

Nimmt der TDF gerade einen hohen Wert an, so ist die Frequenz, mit der der Neuplatzierungsalgoritmus angestoßen wird, klein. Dies ermöglicht ihm mehr Zeit in die Optimierung der Platzierung zu investieren . Für große Szenarien, die viele Ressourcen benötigen, bedeutet dies, dass sich zwar die Komplexität des Optimierungsproblems erhöht, aber auch gleichzeitig mehr Zeit für die Suche einer optimalen Lösung zur Verfügung steht.

Verteilte Ansätze

In diesem Abschnitt sollen zwei verteilte Ansätze vorgestellt werden. Beim ersten steht jedem Rechner wie im zentralen Ansatz das Wissen über den globalen Zustand zur Verfügung. Bei zweiten werden Optimierungen auf Basis von lokalem Wissen durchgeführt.

globales Wissen Im zentralen Ansatz wird der Optimierungsalgorithmus nur auf einem Rechner ausgeführt. Unabhängig von der Größe des Testszenarios stehen in diesem Fall immer die gleichen Ressourcen zur Verfügung. Mit zunehmender Komplexität des Szenarios steigt allerdings die Größe des Suchraums für die Optimierung. Die Wahrscheinlichkeit, eine optimale Lösung zu finden, sinkt dadurch.

Dies motiviert den Ansatz, den Optimierungsalgorithmus parallel auf mehreren Knoten des Testsystems auszuführen. Durch mehr Rechenleistung ist es vielleicht möglich, eine bessere Lösung zu finden. Anstatt also den Optimerungsalgorithmus nur auf dem Koordinator auszuführen, wird er in allen oder einigen virtuellen Maschinen des Testsystems ausgeführt.

Dabei ist es möglich, unterschiedliche Optimierungsalgorithmen zu verwenden sowie Parameter der Algorithmen zu variieren. Wird für die Optimierung der Simulated Annealing Algorithmus verwendet, so kann z.B. der Temperaturverlauf variiert werden. Werden Nachbarzustände ähnlich wie beim Simulated Annealing ausgewählt, so ist die Wahrscheinlichkeit hoch, dass unterschiedliche Instanzen des Algorithmus auch unterschiedliche Wege im Suchraum einschlagen. Dadurch lässt sich der Suchraum besser abdecken und die Wahrscheinlichkeit die optimale Lösung zu finden steigt.

Allen virtuellen Maschinen wird bei diesem Ansatz ein bestimmtes Zeitfenster für die Berechnung einer günstigeren Platzierung zur Verfügung gestellt. Ist die verfügbare Zeit abgelaufen, so werden alle Lösungen verglichen und die beste Platzierung kann in einem nächsten Schritt umgesetzt werden.

Ein Nachteil dieses Ansatzes sind hohe Optimierungskosten. Damit jede virtuelle Maschine eine globale Optimierung durchführen kann, muss jeder virtuellen Maschine globales Wissen zur Verfügung gestellt werden. Anstatt Prognosen über Last und Datenraten nur an den Koordinator zu übertragen, müssen sie im globalen verteilten Ansatz an alle virtuellen Maschinen verteilt werden. Dies kann z.B. per Multicast erfolgen. Jede Maschine wird dabei zusätzlich durch die zu empfangenden Daten belastet.

Da die virtuellen Maschinen für die Ausführung des Experiments benutzt werden, erhöht sich die Laufzeit des Experiments um die Laufzeitkosten der Optimierung.

Dieser Ansatz eignet sich also nur für große Szenarien mit einem hohen Optimierungspotential. Die Frage ist allerdings, ob sich im Mittel in großen Szenarien nicht sowieso ein Lastgleichgewicht einstellt.

lokales Wissen Der im Folgenden vorgestellte verteilte Ansatz orientiert sich an einem Algorithmus namens Sender Initiated Diffusion. Dieser wird im Bereich der Taskmigration eingesetzt und wurde in Abschnitt 3.1.1 vorgestellt.

Ziel des Algorithmus ist es, durch ein verteiltes lokales Loadbalancing ein globales Lastgleichgewicht zu erreichen. Dafür wird das System in sich überlappende Domänen unterteilt: siehe Abbildung 4.15.

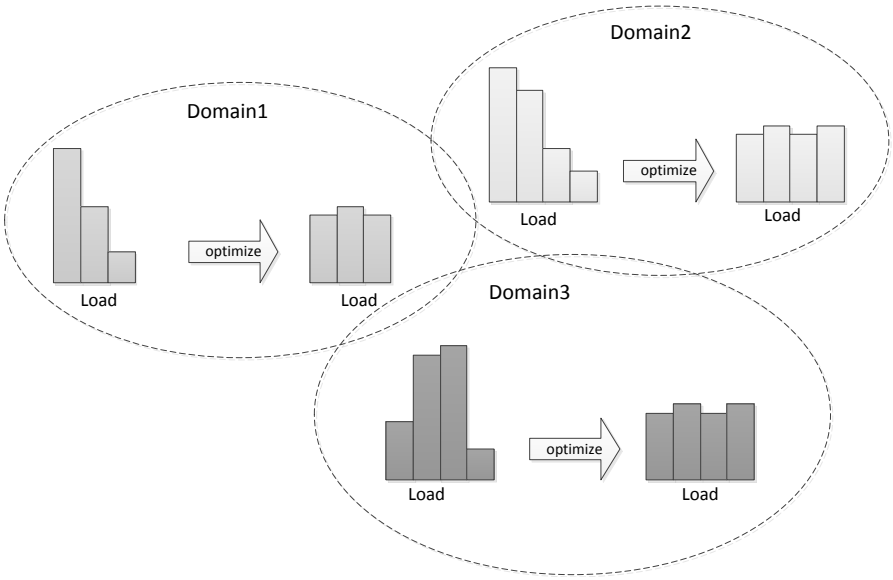


Abbildung 4.15: Verteiltes Loadbalancing

Dieser Ansatz soll hier verwendet werden, um die Experimentlaufzeit eines Experiments zu minimieren. Dabei wird angenommen, dass bei einer guten Platzierung, die Last der einzelnen virtuellen Maschinen ausgeglichen ist. Da unterschiedliche Platzierungen aber zu ungleich hohen Gesamtlasten führen können, ist dies nicht immer zutreffend. Eine Reduktion der Last des höchst ausgelasteten Knoten wird sich allerdings mit hoher Wahrscheinlichkeit durch dieses Verfahren erreichen lassen.

Im folgenden soll nun vorgestellt werden, wie sich der Sender Initiated Diffusion Algorithmus auf die Optimierung einer Platzierung anpassen lässt. Dazu muss zunächst das System, das aus einer Menge von virtuellen Maschinen besteht, in sich überlappende Bereiche aufgeteilt werden.

Dies kann z.B. auf Basis von Nachbarschaftbeziehungen erfolgen. Eine Domäne umfasst in diesem Fall eine virtuelle Maschine und deren direkte Nachbarn. Ein Beispiel für die Bildung einer Domäne, mithilfe von Nachbarschaftsbeziehungen, ist in Abbildung 4.16 dargestellt. Es zeigt eine Routerkette, deren Knoten auf verschiedene Vms verteilt wurden und die Domäne, in der sich die virtuelle Maschine $VM1$ befindet.

In 4.16 wurde die Nachbarschaft von VMs über Verbindungen virtueller Knoten definiert. Dabei ist eine VM benachbart zu einer anderen VM, wenn es mindestens einen Knoten in der VM gibt, der eine Verbindung zu einem Knoten der anderen virtuellen Maschine aufweist.

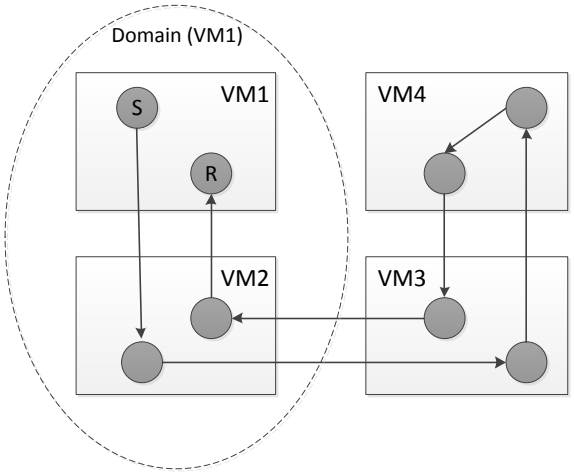


Abbildung 4.16: Bildung einer Domäne

In 4.16 hat der Sender- und Empfängerknoten der Routerkette auf der virtuellen Maschine $vm1$ eine Verbindung zu jeweils einem Knoten der virtuellen Maschine $vm2$. Damit sind diese benachbart und gehören in eine Domäne.

Wird der Neuplatzierungsalgorithmus aufgerufen, so wird in jeder der Domänen ein Load Balancing durchgeführt. Dazu holt jede virtuelle Maschine zunächst die aktuellen Prozessorlasten ihrer Nachbarn ein. Aus diesen Daten wird die mittlere Auslastung der Domäne, wie in (4.28) gezeigt, ermittelt.

$$\lambda_{avg}^{vm} = \frac{1}{|N_{vm}|+1} * \left(\lambda_{vm} + \sum_{i \in N_{vm}} \lambda_i \right) \tag{4.28}$$

Dabei steht N_{vm} für die Menge der Nachbarn der virtuellen Maschine vm und λ_x für die Last der Maschinen.

Liegt die Last der virtuelle Maschine vm über der mittleren Last der Domäne, so wird ein Optimierungsalgorithmus angestoßen. Dieser minimiert die mittlere quadratische Abweichung von der Durchschnittslast durch das Verteilen von Knoten der virtuellen Maschine vm auf ihre Nachbarn (siehe Abbildung 4.17). Die Zielfunktion der Optimierung ist in (4.29) dargestellt.

$$\delta^2 = \frac{1}{|N_{vm}|+1} * \sum_{i \in N_{vm} \cup \{vm\}} \left(\lambda_i - \lambda_{avg}^{vm} \right)^2 \tag{4.29}$$

Dabei steht N_{vm} für die Menge der benachbarten VMs und λ_x für die Last einer virtuellen Maschine x.

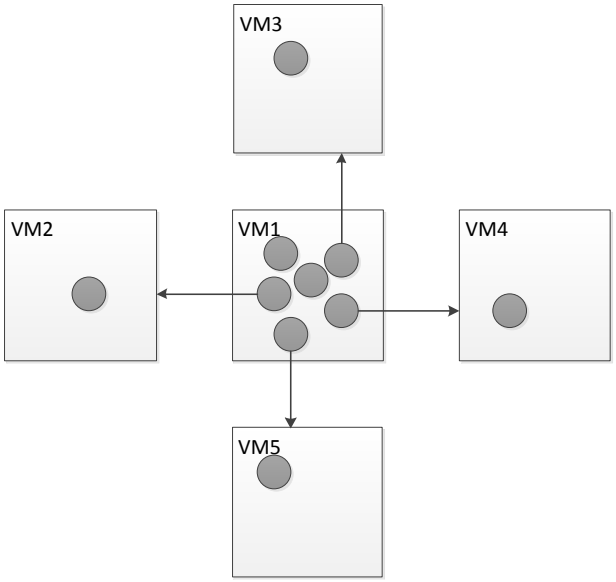


Abbildung 4.17: Verteilte Optimierung der Platzierung

Liegt die virtuelle Maschinen vm unter der mittleren Auslastung, so überspringt sie die Optimierungsphase. Sie wird von anderen virtuellen Maschinen Knoten erhalten.

Für die Berechnung erwarteter Lasten kann das in 4.4 vorgestellte Kommunikationskostenmodell werden. Dafür muss jede virtuelle Maschine einen Ausschnitt der Testszenario Topologie kennen. Prognosen zur Auslastung virtueller Knoten und Datenraten von Verbindungen können in einem Modell wie in Abbildung 4.18 dargestellt, gespeichert werden. Dieses Modell enthält neben den Knoten, die sich auf der eigenen virtuellen Maschine befinden, auch Knoten aus dem Randbereich zu anderen VMs. Bei diesen ist allerdings nur die Information wichtig, auf welcher virtuellen VM sie sich befinden, da diese zur Bildung der Nachbarschaftsbeziehungen benötigt wird.

Um zusätzlich die Gesamtlast zu minimieren, kann der Wert der angestrebten mittleren Last niedriger gewählt werden.

Hat eine virtuelle Maschine wenig Last und eine große Zahl an hoch ausgelastete Nachbarn, so wird sie viele Knoten erhalten. Damit es nicht zu einer Überlastung solcher Maschinen kommt, sollte die Last, die eine virtuelle Maschine an ihre Nachbarn abgeben darf, beschränkt werden.

Nachdem jede virtuelle Maschine Knoten ausgewählt hat, die zu ihren Nachbarn transferiert werden sollen, kann die TVEE rekonfiguriert werden. Dies kann, wie in Abschnitt 4.20

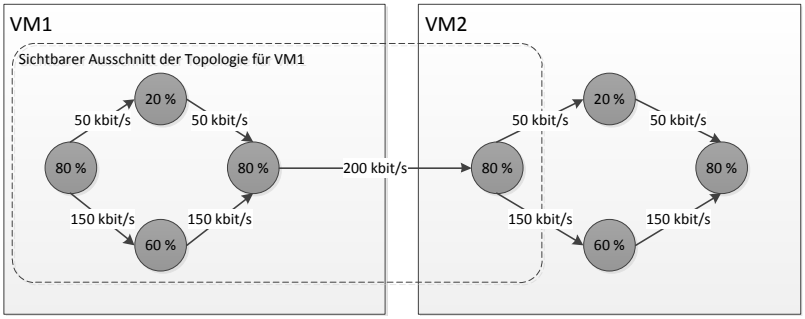


Abbildung 4.18: Lokales Topologiemodell

geschildert, verteilt erfolgen. Soll vor einer Umsetzung zunächst der erwartete Nutzen der Platzierungsänderung bestimmt werden, so müssen lokale Platzierungsänderungen kommuniziert werden. Diese können z.B. an einen Koordinator oder an alle geschickt werden. Aus den lokalen Änderungen lässt sich die neue Gesamtplatzierung rekonstruieren. Für diese können dann Rekonfigurations- und Kommunikationskosten bestimmt werden. Ein Großteil der Berechnungen kann dabei schon lokal auf den jeweiligen virtuellen Maschinen erfolgen. Jede virtuelle Maschine kann z.B. Kosten für das Anhalten der Prozessauführung virtueller Knoten bestimmen. Lohnt sich die Umsetzung der Platzierung, so kann daraufhin die Rekonfiguration angestoßen werden.

Durch die Rekonfiguration der TVEE können sich Nachbarschaftsbeziehungen zwischen virtuellen Maschinen ändern. Erhält eine virtuelle Maschine einen Knoten, so kann sie z.B. neue Nachbarn dazu gewinnen. Damit eine VM die Nachbarschaftsbeziehungen nach einer Rekonfiguration aktualisieren kann, muss die Quell VM eines Knotens Informationen über Verbindungen eines zu migrierenden Knontens mit anderen Knoten an die Ziel VM weitergeben. Mit diesen Informationen kann das lokale Topologiemodell dann angepasst werden.

Vorteil des lokalen verteilten Ansatzes besteht in dem geringen nötigen Austausch von Informationen zwischen virtuellen Maschinen. Im Gegensatz zum zentralen Ansatz müssen nur wenige Lastdaten ausgetauscht werden. Statt der Datenraten von Verbindungen zwischen Knoten und deren Auslastungen, müssen nur aktuelle Lasten von virtuellen Maschinen kommuniziert werden. Dadurch können Kosten gespart werden.

Dem gegenüber entstehen allerdings Kosten für die Ausführung des Optimierungsalgorithmus. Während beim zentralen Ansatz die Optimierung auf einem Rechner ausgeführt werden kann, auf dem keine Knoten des Experiments laufen, ist dies beim verteilten Ansatz nicht möglich. Eine Ausführung des Optimierungsalgorithmus kostet also Ressourcen, die für die das Experiment genutzt werden könnten.

Zudem wird für die lokale Optimierung ein Modell der Topologie zur Berechnung der Kommunikations- und Rekonfigurationskosten benötigt. Dies kostet Speicher in den VMs, der für zusätzliche virtuelle Knoten verwandt werden könnte.

Inwieweit sich durch diesen Ansatz eine Platzierung verbessern lässt, kann ohne Tests schwer abgeschätzt werden.

Diskussion der Ansätze

In den letzten Abschnitten wurden mögliche Ansätze für die Optimierung einer Platzierung im Detail vorgestellt. Dieses Kapitel soll nun einen Überblick über alle geben und diese miteinander vergleichen. Beim Vergleich sollen Kosten für die Optimierung und erwarteter Nutzen gegeneinander abgewogen werden. Kosten sind:

- Zusätzliche Rechenzeit bzw. Last in den VMs
- Benötigter Speicher in den VMs

Der erwartete Nutzen entspricht dem erwarteten Optimierungspotential des Ansatzes.

Tabelle 4.1 zeigt die vorgestellten Ansätze sowie Kosten und Nutzen. Dabei steht $-$ für geringe Kosten und $---$ für hohe Kosten, $+$ bezeichnet einen geringen erwarteten Nutzen und $+++$ einen hohen erwarteten Nutzen.

Ansatz	Last	Speicher	Optimerungspotential
Zentral	$-$	$-$	$++$
Verteilt(global)	$---$	$---$	$+++$
Verteilt(lokal)	$--$	$--$	$+(+)$

Tabelle 4.1: Vergleich verschiedener Optimierungsansätze

Speicher In allen Ansätzen werden in den virtuellen Maschinen Lastmonitore ausgeführt. Diese nehmen zu zwei Zeitpunkten Messwerte wie die genutzte Anzahl von Zyklen eines virtuellen Knotens oder die Anzahl der gesendeten Bytes an eine bestimmte Zieladresse auf. Aus diesen Messwerten wird mit $l = \frac{measuredValue(t_2) - measuredValue(t_1)}{t_2 - t_1}$ die Last eines Knotens oder Datenraten für Verbindungen zwischen virtuellen Knoten ermitteln. Auf Basis dieser Daten werden Prognosen über zukünftige Mittelwerte erstellt. Für die Speicherung der Messwerte wird in allen Ansätzen Arbeitsspeicher benötigt.

Befinden sich auf einer virtuellen Maschine beispielsweise 5000 Knoten und hat jeder dieser Knoten im Durschchnitt 5 Verbindungen zu anderen Knoten, so wird in der aktuellen Implementierung ungefähr 2 mb Arbeitsspeicher in einer virtuellen Maschine, für die Speicherung der letzten beiden Messwerte benötigt.

Zusätzlich zu Messwerten wird in den verteilten Ansätzen ein Topologiemodell des Testszenarios in jeder VM erstellt. In diesem Modell werden Prognosen zu mittleren Datenraten von Verbindungen und Lasten einzelner Knoten hinterlegt.

In der aktuellen Implementierung wird die Topologie des Testszenarios auf einen Graph abgebildet. Dabei werden virtuelle Knoten auf Knoten des Graphen und Verbindungen zwischen virtuellen Knoten auf Kanten abgebildet. An jeder Kante des Graphen kann die mittlere Datenrate und an jedem Knoten die mittlere Last gespeichert werden. Für dieses Modell wird zusätzlicher Speicher benötigt.

Befinden sich wie im Beispiel oben 5000 Knoten auf einer virtuellen Maschine und hat jeder dieser Knoten durchschnittlich 5 Verbindungen, so wird beim lokalen verteilten Ansatz etwa 8 mb für das Topologiemodell benötigt. Im lokalen Ansatz muss jede TVEE nur den für ihn sichtbaren Teil der Topologie speichern. Dieser umfasst Randknoten und Knoten, die sich auf ihr befinden.

Im globalen verteilten Ansatz hingegen benötigt jede VM die Sicht auf die komplette Topologie. Nimmt man die gleiche Anzahl von Knoten pro VM an wie im vorigen Beispiel und eine Gesamtzahl von 64 VMs, so ist der Speicherbedarf pro VM z.B. 512 mb - also deutlich höher.

Im zentralen Ansatz wird hingegen das komplette Topologiemodell auf dem Koordinator gespeichert. In den einzelnen VMs wird dafür also kein zusätzlicher Speicher benötigt.

Zusätzliche Rechenzeit, bzw. Last Beim zentralen Ansatz wird der Optimierungsalgorithmus auf dem Koordinator ausgeführt. Dieser ist nicht Teil des Experiments, verursacht also keine Kosten in einer virtuellen Maschine. Pro virtueller Maschine entstehen im zentralen Ansatz allerdings Kosten für das Übermitteln der Lasten und Datenraten an den Koordinator.

Für 5000 Knoten pro VM und 5 Verbindungen pro Knoten muss jede VM beispielsweise 1 mb an Nutzdaten übertragen. Bei einer 1 Gbit Verbindung wird für den Transfer der Daten ungefähr 10 Millisekunden Rechenzeit benötigt.

Im verteilten globalen Ansatz müssen Lastdaten an alle VMs verteilt werden. Da jede VM gleichzeitig Sender und Empfänger ist, wird für den Transfer der Daten ungefähr doppelt soviel Zeit benötigt. Zudem kommt noch die Rechenzeit, die für die Ausführung des Optimierungsalgorithmus benötigt wird. Diese wird in höherer Größenordnung als der Lastdatentransferzeit liegen. Wird zu wenig Zeit in die Optimierung investiert, ist mit schlechten Resultaten zu rechnen.

Im verteilten lokalen Ansatz wird die benötigte Rechenzeit im Wesentlichen von dem Optimierungsalgorithmus bestimmt. Da jede VM nur einen Ausschnitt der Topologie kennt, ist das Optimierungsproblem weniger komplex als im globalen Fall. Aus diesem Grund muss für die Optimierung weniger Zeit eingeplant werden. Die benötigte Rechenzeit wird

voraussichtlich trotzdem um einiges höher sein als die im zentralen Fall für die Kommunikation von Lasten benötigte Zeit. Diese betrug pro VM in einem großen Szenario ungefähr 10 Millisekunden.

Optimierungspotential Beim zentralen Ansatz wird der Optimierungsalgorithmus auf einem Rechner, genannt Koordinator, ausgeführt. Dieser optimiert die in 4.6.1 vorgestellte Zielfunktion. Da der Koordinator über eine globale Sicht verfügt, ist das Optimierungspotential hoch.

Im zentralen Ansatz sind allerdings die Ressourcen beschränkt. Ist man an einer optimalen Lösung interessiert, bietet sich daher der verteilt globale Ansatz an.

Das Optimierungspotential des verteilt lokalen Ansatzes kann nur schwer abgeschätzt werden, zumal in diesem Ansatz nicht direkt die Experimentlaufzeit minimiert werden kann.

4.8.2 Koordination der Rekonfiguration der TVEE

In Kapitel 4.3 wurden Operationen vorgestellt die zur Rekonfiguration der TVEE im Zuge einer neuen Platzierung nötig sind. Außerdem wurde in Abschnitt 4.3.3 eine sinnvolle Reihenfolge der Operationen motiviert. Wie diese Reihenfolge auch bei einer verteilten Ausführung von Operationen eingehalten werden kann, wurde in Abschnitt 4.3.4 vorgestellt.

In diesem Abschnitt soll nun auf die Koordination der Rekonfiguration eingegangen werden. Also z.B. auf die Frage, wer die zur Konfiguration der TVEE nötigen Aktionen einer Operation , wie z.B. Erzeugung einer Software Brücke, Suspend eines Knotens, generiert.

Hier sollen zwei Ansätze vorgestellt werden: Ein verteilter und ein zentraler Ansatz.

Zentraler Ansatz

Bei der zentralen Rekonfiguration, dargestellt in 4.19, wird die Anpassung der TVEE zentral von einem Koordinator gesteuert. Dieser bestimmt auf Basis gewünschter Platzierungsänderungen Aktionen, die im Zuge der Rekonfiguration auszuführen sind.

Um diese Aktionen bestimmen zu können, benötigt der Koordinator Informationen über die aktuelle Konfiguration der Emulationsumgebung. Wichtige Informationen sind z.B. die aktuelle Position virtueller Knoten sowie die Konfiguration von Softwarebrücken und Vlans. Aus diesen Informationen kann er dann nötige Aktionen, wie z.B. das Erzeugen einer Software-Brücke und das Anbinden eines Netzwerkgerätes, an eine Brücke ableiten.

Um diese Informationen nicht vor jeder Rekonfiguration ermitteln zu müssen, sollte zu Beginn ein Abbild der Emulationsumgebung erstellt werden. Dies kann im Folgenden bei

jeder Rekonfiguration aktualisiert werden. Ein Beispiel, wie auf Basis dieses Abbildes die Softwarebrücken-Konfiguration angepasst werden kann, wurde in 4.3.2 vorgestellt.

Wurden nötige Aktionen bestimmt, so müssen diese in einem nächsten Schritt in den virtuellen Maschinen ausgeführt werden. Dazu wird in jeder VM ein Daemon gestartet, der auf auszuführende Aktionen lauscht. Eine Verbindung zwischen dem Koordinator und einer VM kann über TCP realisiert werden.

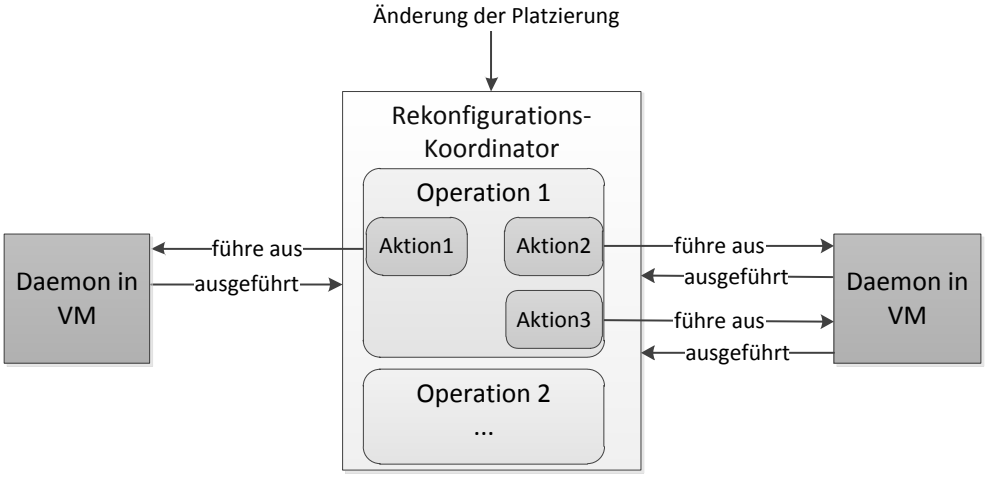


Abbildung 4.19: Zentrale Rekonfiguration

Erhält eine VM eine Aktion, so führt sie diese aus und bestätigt die Ausführung mit einer Nachricht („ausgeführt“). Besteht eine Operation, wie in 4.19 zu sehen, aus mehreren Aktion, so kann durch diese Nachricht sichergestellt werden, dass alle Aktionen einer Operation ausgeführt sind, bevor die nächste Operation gestartet wird. Wie in 4.3.3 vorgestellt ist eine bestimmte Reihenfolge von Operationen einzuhalten.

Für jede auszuführende Aktion wird eine Nachricht an die VM geschickt, in der sie auszuführen ist. Um möglichst wenig Nachrichten verschicken zu müssen, können Nachrichten für ähnliche Aktionen zusammengefasst werden. Muss z.B. der Zustand mehrerer Knoten in einer VM gesichert werden, so muss nicht für jeden Knoten eine einzelne Nachricht geschickt werden. Stattdessen kann eine Nachricht genutzt werden, die alle Knoten-IDs zu sichernder Knoten enthält. Dadurch wird das zu kommunizierende Datenvolumen gesenkt.

Verteilter Ansatz

Ein möglicher verteilter Ansatz ist in Abbildung 4.20 dargestellt.

Im Gegensatz zum zentralen Ansatz werden auszuführende Aktionen nicht von einem Koordinator, sondern von Daemons in den VMs bestimmt. Auf Basis der Platzierungsänderung und einem lokalen Abbild der Emulationsumgebung ermittelt jeder Daemon Aktionen, die in seiner VM auszuführen sind.

Wird ein zentraler Ansatz zur Optimierung der Platzierung gewählt, so können die als Input benötigten Platzierungsänderungen jeder VM, z.B. über ein Broadcast, zur Verfügung gestellt werden. Wird ein verteilter Ansatz gewählt, so liegen Informationen über Platzierungsänderungen bereits vor; sie müssen also nicht kommuniziert werden.

Im Gegensatz zum zentralen Ansatz der Rekonfiguration wird im verteilten Ansatz in jeder VM ein Abbild der Emulationsumgebung benötigt. Dieses muss virtuelle Knoten, die sich in ihr befinden sowie aktuelle Brücken und Vlan Konfigurationen umfassen. Hauptsächlich wird das Abbild für die Layer 2 Topologie Adaption benötigt.

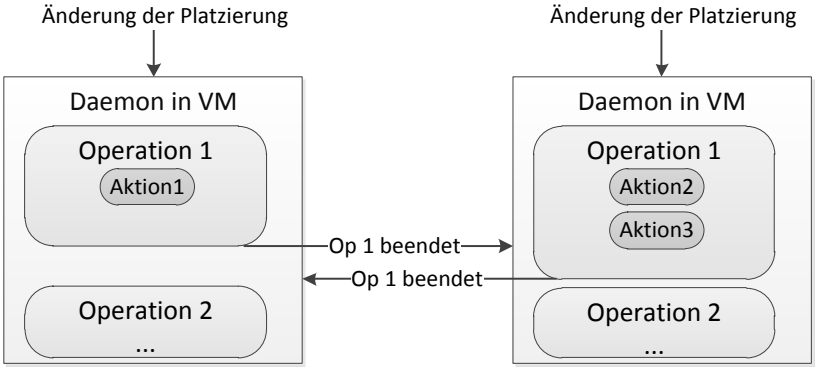


Abbildung 4.20: Zentrale Rekonfiguration

Um sicherzugehen, dass Operationen in der richtigen Reihenfolge ausgeführt werden, wird der in 4.3.4 vorgestellte verteilte Ansatz benutzt. Nach der erfolgreichen Ausführung einer Operation schickt jede VM den anderen VMs eine Nachricht(z.B. „Op 1 beendet“). Im Gegensatz zum zentralen Ansatz müssen allerdings für die Aktionen, aus denen sich eine Operation zusammensetzt, keine Nachrichten verschickt werden.

Für die Anpassung der Layer 2 Topologie müssen allerdings zusätzliche Informationen in die Sicherungsdatei eines Knotens aufgenommen werden. Da die Ziel-VM nur eine lokale Sicht auf das Abbild der Emulationsumgebung hat, besitzt sie keine Informationen darüber, an welche Brücken ein Knoten, den sie erhält, vorher angebunden war. Diese Informationen werden allerdings zur erfolgreichen Anpassung der Layer 2 Topologie benötigt.

Diskussion der Ansätze

In den letzten beiden Abschnitten wurden zwei Ansätze zur Koordination der Rekonfiguration vorgestellt: ein verteilter und ein zentraler Ansatz. In diesem Teil der Arbeit sollen nun beiden Ansätze verglichen werden. Dabei soll besonders auf den Koordinations-Overhead beider eingegangen werden. Betrachtet werden sollen:

- Zusätzliche Last in VM
- Benötigter Speicher in VM

Last Die Last in VMs, die durch die Koordination der Rekonfiguration entsteht, wird hauptsächlich durch die nötige Kommunikation bestimmt. Daher sollen hier die Anzahl der benötigten Nachrichten pro Rekonfiguraiton in beiden Ansätzen verglichen werden.

Die Rekonfiguration lässt sich wie in 4.3 vorgestellt, in folgende Operationen unterteilen:

- Setzen eines hohen TDF Wertes
- Setzen virtueller Knoten in einen Haltezustand(Suspend)
- Migration von virtuellen Knoten und Netshaper Instanzen. Zerfällt in Teiloperationen Sichern(Dump), Transfer und Wiederherstellen (Undump)
- Anpassung Layer 2 Topologie (Layer 2 Adaption)
- Wiederaufnahme Prozessausführung in virtuellen Knoten(Resume virtueller Knoten)
- Wiederaufnahme Paketzustellung in Netshaper Instanzen(Resume Netshaper)
- Rücksetzen der TDF Änderung

Im verteilten Ansatz werden Nachrichten nach Ausführung aller Aktionen einer Operation ausgetauscht. Jede virtuelle Maschine verschickt dabei eine Nachricht (kann über Mulicast oder Broadcast an alle weitergeleitet werden) und empfängt $|VM| - 1$ Nachrichten nach jeder Operation. Dies bedeutet, dass für die gesamte Rekonfiguration für eine konstante Zahl an Operationen jede VM $O((|VM|))$ Nachrichten kommunizieren muss. Jede Nachricht kann dabei sehr kurz sein, da sie nur als Bestätigung für eine ausgeführte Operation fungiert.

Im zentralen Ansatz empfängt und sendet jeder Daemon in einer VM eine Nachricht für jede Aktion bzw. Gruppe von Aktionen, die er auszuführen hat.

Tabelle 4.2 zeigt wieviele Nachrichten eine VM pro Operation empfängt:

Für die Operationen Start und Stopp der globalen virtuellen Zeit muss nicht mit den VMs kommuniziert werden (sondern mit dem Koordinator der Zeitvirtualisierung).

Operation	Anzahl Nachrichten
Stopp globaler Zeit	0
Suspend	1
Dump	n_{dump}^{vm}
Transfer	1
Undump	1
Layer 2 Adaption	$\|createBridge_{vm}\| + \|destroyBridge_{vm}\| + \|attachVNic_{vm}\|$
Resume virtueller Knoten	1
Resume Netshaper	1
Start globaler Zeit	0

Tabelle 4.2: Übersicht Nachrichten zentrale Koordination

Ermittelt der Daemon vor Ausführung der Operationen Resume und Suspend die virtuellen Knoten, die sich derzeit in seiner VM befinden (z.B. über *vzlist*), so müssen diese Operationen nur vom Koordinator angestoßen werden. Es wird daher nur eine Nachricht benötigt, die sehr klein sein kann.

Die Nachrichten, die für die Aktionen der Operation Dump eines virtuellen Knotens und dessen Netshaper Instanzen benötigt werden, richten sich nach der Anzahl der zu sichernden virtuellen Knoten in der VM (n_{dump}^{vm}). Jede Nachricht muss die Id des zu sichernden Knoten enthalten. Alternativ kann auch eine Nachricht mit allen Knoten IDs verschickt werden.

Die Anzahl der Nachricht, die für die Layer 2 Adaption benötigt werden, wird durch die Zahl der zu erstellenden Softwarebrücken($|createBridge_{vm}|$), der zu zerstörenden Brücken ($|destroyBridge_{vm}|$) und der Zahl der wieder anzubindenden virtuellen Netzwerkgeräte $|attachVNic_{vm}|$ an eine Brücke bestimmt. Jede Nachricht enthält dabei mindestens die Id einer Brücke.

Wird die Rekonfiguration zentral koordiniert, werden $0(n_{dump}^{vm} + |createBridge_{vm}| + |destroyBridge_{vm} + |attachVNic_{vm}|)$ Nachrichten benötigt. Jede Nachricht muss dabei meist allerdings nur wenige Bytes an Nutzdaten enthalten. Zudem können die meisten Nachrichten zu größeren Nachrichten zusammengefasst werden. Selbst wenn eine virtuelle Maschine 5000 Knoten verliert und andere 5000 Knoten erhält und dabei 5000 Brücken erzeugt und gelöscht werden, beläuft sich die Gesamtgröße der Nachrichten auf unter 200 kbyte.

In beiden Ansätzen ist der Overhead für die Kommunikation im Vergleich zu denen durch Operationen entstehenden Kosten gering. Zwar ist der Kommunikationsaufwand für eine VM im zentralen Ansatz höher, dafür entstehen allerdings auch keine Kosten für die Ermittlung nötiger Aktionen in den VMs.

Speicher Im zentralen Ansatz wird kein zusätzlicher Speicher in den VMs benötigt. Das komplette Abbild der Emulationsumgebung befindet sich bei der zentralen Koordination auf dem Koordinator. Auf diesem befinden sich in der Regel keine virtuellen Knoten des Experiments.

Im verteilten Ansatz speichert jede VM den für sie sichtbaren Teil des Abbildes. Dies umfasst z.B. Knoten und Brücken, die sich in ihr befinden. Zwar wird dafür zusätzlicher Speicher benötigt, dieser liegt allerdings höchstens im 1 mb Bereich(5000 Knoten in VM), ist also eher zu vernachlässigen.

Zusammenfassend kann man sagen, dass für die Koordination der Rekonfiguration beide Ansätze gleich gut geeignet sind. Ob die Koordination verteilt oder zentral ausgeführt wird, sollte sich daher nach der Wahl des Optimierungsalgorithmus richten. Wird eine verteilter Optimierungsansatz verfolgt, so sollte die Rekonfiguration auch verteilt sein. Dadurch wird eine nötige Kommunikation von Platzierungsänderungen vermieden.

5 Implementierung

In diesem Kapitel wird nun näher auf den implementierten Prototypen eingegangen. Es wurde sich für die Implementierung des zentralen Neuplatierungsansatzes entschieden, da sich dieser als viel versprechend herausstellte.

Sowohl die Optimierung der Platzierung als auch die Rekonfiguration der TVEE werden zentral koordiniert. Für die Implementierung des Koordinators wurde Java verwendet. Daemons, die in den einzelnen virtuellen Maschinen ausgeführt werden, um z.B. Rekonfigurationsaktionen entgegen zu nehmen, wurden ebenfalls in Java realisiert.

Zur Umsetzung einzelner Aktionen der Rekonfiguration, wie z.B. das Setzen eines virtuellen Knotens in den Haltezustand, wurden Bash Skripte verwendet.

Die Grundideen des zentralen Ansatzes wurden schon im Abschnitt 4.8 erläutert. Eine Erläuterung der Rekonfiguration erfolgte in Abschnitt 4.3. Deshalb soll hier nur kurz auf Implementierungsdetails eingegangen werden.

5.1 Rekonfiguration

In dem Prototyp wird die Rekonfiguration zentral von einem Koordinator gesteuert. Dieser veranlasst den Aufruf von Tools und Skripten in den VMs, die für die Durchführung einzelner Operationen wie z.B. dem Suspend von Knoten nötig sind. Genutzte Tools und durchzuführende Erweiterungen bestehender Tools werden in diesem Abschnitt erläutert.

5.1.1 Suspend/Resume virtueller Knoten

Für Suspend und Resume von virtuellen Knoten wird das OpenVZ Tool *vzctl* eingesetzt. Über *vzctl chkpnt 1 –suspend* lässt sich beispielsweise der Container mit der Id 1 in den Haltezustand bringen.

Für jeden Knoten auf einer virtuellen Maschine muss das Tool einzeln aufgerufen werden. Dabei terminiert ein Aufruf des Tools erst wenn alle Prozesse innerhalb eines Containers gestoppt wurden. Der aktuelle Zustand eines Prozesses wird dabei periodisch überprüft. Um die damit verbundenen Wartezyklen nicht zu verschwenden werden alle nötigen Aufrufe des Tools parallel ausgeführt (*vzctl...&*). Um sicherzustellen, dass am Ende alle Container einer

VM suspended/resumed sind, wird auf die Terminierung aller Tool Instanzen gewartet ((*„wait“*)).

5.1.2 Migration virtueller Knoten

Einen wesentlichen Teil der Migration bilden das Sichern (*dump*) und Wiederherstellen (*undump*) eines virtueller Knoten. Dafür wird das OpenVZ Tool *vzctl* genutzt. Mittels des Befehls *vzctl chkpnt 1 –dump –dumpfile /vz/dump/1.dump* lässt sich z.B. der Zustand des Containers mit der Id 1 in das Verzeichnis */vz/dump/1.dump* sichern.

Der Transfer der Sicherung zum Zielrechner geschieht über das Tool *netcat*. *netcat* überträgt Daten von der Standardeingabe über TCP zu einem entfernten Rechner. Dazu muss auf dem Zielrechner das Tool im Listen Modus gestartet werden. Damit gleichzeitig unterschiedliche VMs Daten übertragen können, werden im Hintergrund so viele netcat Instanzen im Listen Modus gestartet wie virtuelle Maschinen an einem Experiment teilnehmen. Jeder VM steht damit eine eigene TCP Verbindung zur Übertragung von Daten an eine bestimmte Ziel VM zur Verfügung.

5.1.3 Migration von Netshaper Instanzen

Wie bereits in Abschnitt 4.3 erwähnt wird für die Sicherung der Netshaper Instanz das Proc Dateisystem benutzt. Über diese Schnittstelle kann eine Netshaper Instanz konfiguriert werden; über *echo 100 > /proc/vz/simple_ns/ve1_etho_rcv/bandwidth* kann z.B. die Bandbreite des empfangenden Netzwerkgeräts mit der Id 0 des Containers 1 gesetzt werden. Kopiert man den kompletten Proc Ordner einer Netshaper Instanz beispielsweise */proc/vz/simple_ns/ve1_etho_rcv* mit allen darin enthaltenen Dateien, so lassen sich aktuelle Einstellungen der Netshaper Instanz einfach festhalten.

Zur Sicherung von gepufferten Frames und der Parameterliste(wird in MANet Szenarien benötigt) musste die bisherige Proc Schnittstelle des Netshaper Tools noch erweitert werden. Über die Dateien *macdump* und *framedump* lassen sich nun Frames und Parameter auslesen.

Der Transfer der gesicherten Daten geschieht, wie bei der Migration der virtuellen Knoten, über das netcat Tool. Wiederhergestellt wird eine NetshaperInstanz durch das Zurückkopieren des gesicherten Ordners auf dem Zielrechner.

5.1.4 Anpassung der Layer 2 Topologie

Zur Anpassung der Layer 2 Topologie müssen Softwarebrücken erstellt und gelöscht werden. Außerdem sind Vlans Einzurichten und mit Softwarbrücken zu verbinden.

Für die Konfiguration Softwarebrücken wird das Tool *brctl* genutzt. Für das Einrichten von Vlans wird *vconfig* verwendet.

5.2 Optimierung der Platzierung

Die Optimierung der Platzierung geschieht zentral auf dem Koordinator mittels des in 4.6.2 vorgestellten Optimierungsalgorithmus. Die Zielfunktion des Optimierungsalgorithmus baut dabei auf einem Kommunikationskostenmodell auf. Zur Berechnung der Kosten in diesem Modell werden Informationen über die Topologie des Testszenarios und die aktuelle Position virtueller Knoten benötigt. Zu Beginn des Experiments müssen diese dem Neuplatzierungsalgorithmus zur Verfügung gestellt werden.

Um möglichst unabhängig von anderen Tools zu sein, die in der TVEE eingesetzt werden, ermittelt der Neuplatzierungsalgorithmus zu Beginn die relevanten Informationen selber. Damit dies funktioniert darf das Tool erst gestartet werden, wenn die TVEE bereits für ein Experiment konfiguriert ist. Die nötigen Informationen werden über zwei Tools ausgelesen, die in den virtuellen Maschinen des Testsystems auszuführen sind.

vzlist zeigt alle Container an die in einer virtuellen Maschine ausgeführt werden. Über dieses Tool lässt sich also die Position jedes Knotens bestimmen.

brctl show liefert eine Übersicht über alle Softwarebrücken in einer VM. Zusätzlich zu den Brücken werden alle Netzwerkgeräte angezeigt, die mit diesen verbunden sind. Mittels dieser Daten lässt sich eine Kommunikatiosmatrix von Knoten bestimmen. Daraus lässt sich dann das Topologiemodell ableiten.

5.3 Monitore

Dieser Abschnitt beschäftigt sich mit Implementierungsdetails der Monitore. Diese zeichnen Daten wie mittlere Datenrate von Verbindungen zwischen Knoten sowie mittlere Lasten von virtuellen Knoten auf.

5.3.1 Mittlere Datenraten

Zur Bestimmung der mittleren Datenrate einer Verbindung zwischen zwei Knoten muss zu zwei Zeitpunkten die Datenmenge , die von einem zum anderen Knoten übertragene wurde, erfasst werden. Über $d = \frac{tx_{bytes}(t_2) - tx_{bytes}(t_1)}{t_2 - t_1}$ lässt sich dann die mittlere Datenrate bestimmen.

Für die Ermittlung der mittleren Datenraten einer Verbindung musste die Statistik des Netshapertools erweitert werden. Bisher wurde nur die Gesamtdatenmenge,die über die Netshaper Instanz gesendet wurde, erfasst. (Z.B. $tx_{bytes} = 50000$). Für die Bestimmung mittlerer Datenraten von Verbindungen muss die Statistik allerdings nach Ziel der Pakete aufgeschlüsselt werden. Für jede mögliche Ziel Adresse (Mac Adresse) müssen die Anzahl der gesendeten Bytes festgehalten werden.

Dies führte zu einer erweiterte Statistik, die über die Proc Datei *statistic* in dem proc Ordner einer Netshaper Instanz ausgelesen werden kann. Der Inhalt der Datei hat dabei das Format *(macAdresse txBytes)**.

5.3.2 Mittlere Auslastung

Zur Bestimmung der mittleren Auslastung virtueller Knoten muss die Anzahl genutzter Zyklen zu zwei Zeitpunkten erfasst werden. Diese Information wird der Datei „/proc/vz/-vestat“ entnommen.

5.4 Probleme Rekonfiguration

Während des Tests der Rekonfiguration traten verschieden Probleme auf. Die, die bisher noch nicht gelöst werden konnten, sollen hier kurz vorgestellt werden.

5.4.1 Routing Tabelle

Bei einem Routerketten Testszenario mit 8 Knoten, die auf zwei virtuelle Maschinen verteilt wurden, konnten Routertabelleneinträge einzelner Knoten nach der Migration nicht korrekt wiederhergestellt werden. Teilweise fehlten Einträge. Für jeden fehlenden Eintrag wird eine Kernelmeldung ausgegeben. Z.B. „CPT ERR: e78f8800,5 :NLMERR: -101“. Bisher konnte die Ursache dafür noch nicht ermittelt werden. Benutzt wurde der Kernel ovzkernel-2.6.18-194.3.1.el5.028stab069.6 und die *vzctl* Tool Version 3.0.24.2.

5.4.2 Probleme im Zusammhang mit netperf und iperf

Netperf und *iperf* sollten als Traffic Generatoren für den Test der Rekonfiguration verwendet werden. Allerdings traten Probleme im Zusammenhang mit der Rekonfiguration mit diesen beiden Tools auf.

Wird ein Container, in dem sich eine *netserver* Instanz befindet in den Haltezustand versetzt und danach weiter ausgeführt, so wird eine fin Nachricht an die mit ihm kommunizierende

netperf Instanz geschickt. Die TCP Verbindung, die zwischen beiden besteht, wird dadurch vorzeitig geschlossen.

Ein weiteres Problem mit *netperf* und *iperf* besteht im Zusammenhang mit der Zeitvirtualisierung. Von Zeit zu Zeit springt der TDF kurz nach der Rekonfiguration in sehr kurzer Zeit auf den Maximalwert. Betrachtet man die Last einzelner migrierter virtueller Knoten, so ist diese sehr viel höher als normal. Lässt man einen massiven Anstieg des TDF durch die Wahl eine sehr kleinen Maximalwerts nicht zu, so stellt sich nach kurzer Zeit wieder ein normales Verhalten ein. Grund für das seltsame Verhalten könnte z.B. ein *busy waiting* sein.

Aufgrund dieser Probleme wurde für die Generierung von Traffic auf netcat gewechselt. Zum Zeitpunkt der Abgabe der Diplomarbeit war allerdings noch nicht klar, ob sich dadurch oben genannte Probleme vermeiden lassen.

Für eine transparente Rekonfiguration ist es wichtig die globale virtuelle Zeit nur sehr langsam weiterlaufen zu lassen. Dazu wird zu Beginn der Rekonfiguration der TDF auf den Maximalwert gesetzt: aktuell 1000. Eine Millisekunde vergeht damit in einer Sekunde. Dies ist jedoch kritisch für die Performance der Rekonfiguration. In der OpenVZ Implementierung der Operationen *Kill*, *Suspend* und *Resume* von Containern werden Timer benutzt. Meist liegen Zeiten der Timer im Millisekunden Bereich. Durch die Zeitskalierung benötigen Operationen allerdings teilweise mehrere Sekunden. Für eine effiziente Rekonfiguration müsste also der entsprechende OpenVZ Code angepasst werden. Dies wurde für die Operation *Suspend* auch schon bereits durchgeführt.

6 Evaluation

6.1 Konstanten Rekonfigurationskostenmodell

In diesem Abschnitt sollen die Konstanten des in 4.5 vorgestellten Rekonfigurationsmodells für die aktuelle Hard- und Sofware Konfiguration der Testumgebung bestimmt werden.

In der aktuellen Hardware Konfiguration besitzt jeder physikalische Rechner der Testumgebung eine Intel Xeon CPU mit 8 Kernen und 24 Gigabyte Arbeitsspeicher. Jeder Kern wird mit 2,4 Ghz getaktet.

Als Betriebssystem kommt ein modifiziertes Red Hat 4.1.2-48 zum Einsatz.

Abbildung 6.1 zeigt die Kosten für *suspend* und *resume* in Abhängigkeit von der Anzahl der Knoten. In jedem Knoten wird ein Prozess ausgeführt. Dieser steht stellvertretend für die Software Under Test. Jeder Prozess belastet das System nur leicht.

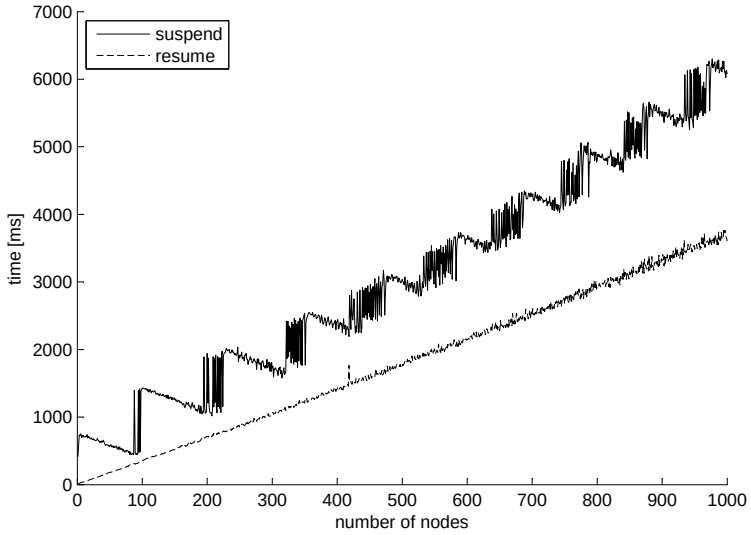


Abbildung 6.1: Kosten für Suspend und Resume Operation in Abhängigkeit von der Anzahl der Knoten

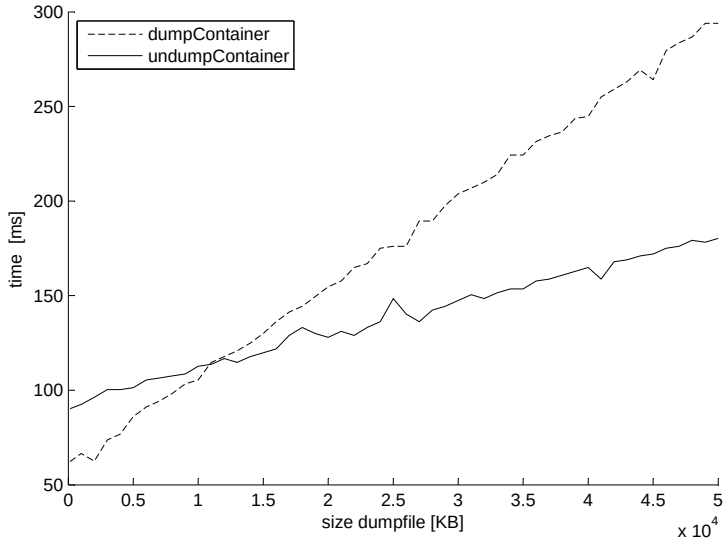


Abbildung 6.2: Kosten für Dump und Undump eines Knotens in Abhängigkeit von erwarteter Größe des Dumpfiles

Man erkennt, dass die Zeit, die für die *suspend* Operation benötigt wird schwankt, aber im Groben linear steigt. Warum die Zeit mit höherer Knotenanzahl sogar teilweise rückläufig ist, kann ich mir nicht erklären. Antwortet ein Prozess nicht direkt auf das Stopp Signal, so wird eine bestimmte Zeit gewartet. Aktuell bei einem TDF von 1000 1 Sekunde. Wie in 6.1 zu sehen kann bei geringem Unterschied der Knotenanzahl die benötigte Zeit um 1 Sekunde schwanken. Daher lassen sich Kosten für die *suspend* Operation nicht besonders präzise voraussagen.

6.2 zeigt die Kosten für das Sichern und Wiederherstellen des Zustands eines Knotens in Abhängigkeit von der Größe der erzeugten Sicherungsdatei. Die Größe der erzeugten Datei kann gut durch den genutzten Arbeitsspeicher des Knotens abgeschätzt werden.

'"Gedumpt" wurde in diesem Test in den Arbeitsspeicher der virtuellen Maschine (/dev/shm/). Man erkennt, dass die Kosten linear mit der Größe des Sicherungsdatei steigen.

6.3 zeigt die Kosten für das Sicher, und Wiederherstellen einer Netshaper Instanz in Abhängigkeit von der Größe der sich derzeit im Puffer befindlichen Frames.

Man erkennt, dass das Sichern von Frames teurer ist als das Wiederherstellen. In der aktuellen Implementierung werden Frames über die Proc Schnittstelle gesichert. Dabei wird eine proc-read Funktion separat für jeden zu sichernden Frame aufgerufen. Beim Wiederherstellen, (dies geschieht über eine proc-write Funktion) werden mehrere Frames gleichzeitig wiederhergestellt. Für das Sichern der Frames ist also der Overhead für Funktionsaufrufe höher.

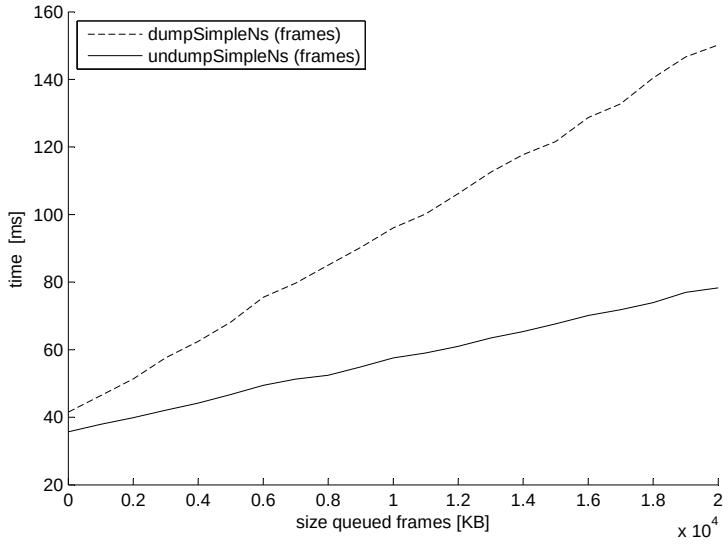


Abbildung 6.3: Kosten für Dump und Undump einer Netshaper Instanz in Abhängigkeit von der Größe der gepufferten Frames

6.4 zeigt die Kosten für den Dump und Undump einer Netshaper Instanz in Abhängigkeit von der Größe der Parameterliste. Auch hier lässt sich wieder beobachten, dass das Sichern der Parameterliste teurer ist als das Wiederherstellen. Dies ist aus den gleichen Gründen wie oben der Fall.

6.5 zeigt die Kosten für das Beenden von Knoten. Diese steigen mit der Anzahl der Knoten schwanken allerdings sehr stark. Ähnlich wie bei der *suspend* Operation werden vermutlich auch hier Timer benutzt, die unter bestimmten Umständen getriggert werden.

Wie lange das Beenden von einer bestimmten Anzahl von Knoten dauert lässt sich nur sehr schwer abschätzen. Für eine grobe Abschätzung wird hier eine Ausgleichsgerade verwendet.

6.6 zeigt als letztes die Dauer für den Transfer von Daten in Abhängigkeit von dem zu übertragenden Datenvolumen und der Art der Verbindung zwischen den VMs. Liegt die Ziel VM auf einem anderen physikalischen Knoten als die Quell VM so entstehen leicht höhere Kosten.

Aus den vorgestellten Graphen lassen sich die für das Modell benötigten Konstanten bestimmen. Diese sind in den Tabellen 6.1 und 6.2 dargestellt.

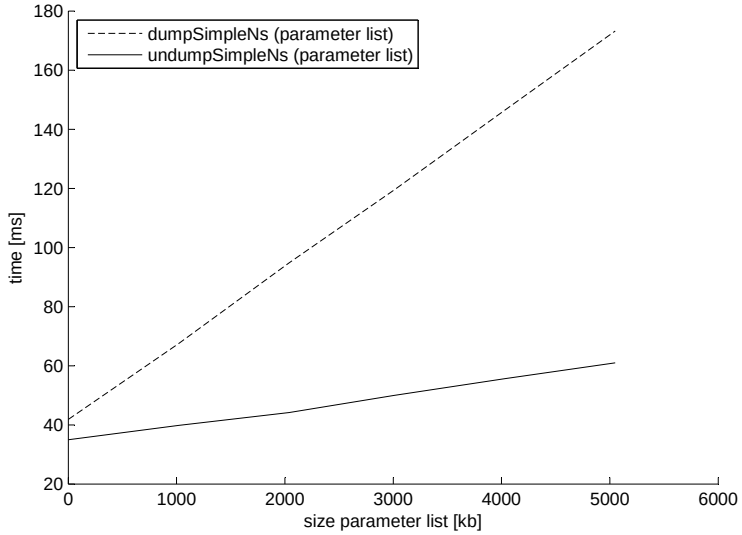


Abbildung 6.4: Kosten für Dump und Undump einer Netshaper Instanz in Abhängigkeit von der Größe der Parameterliste

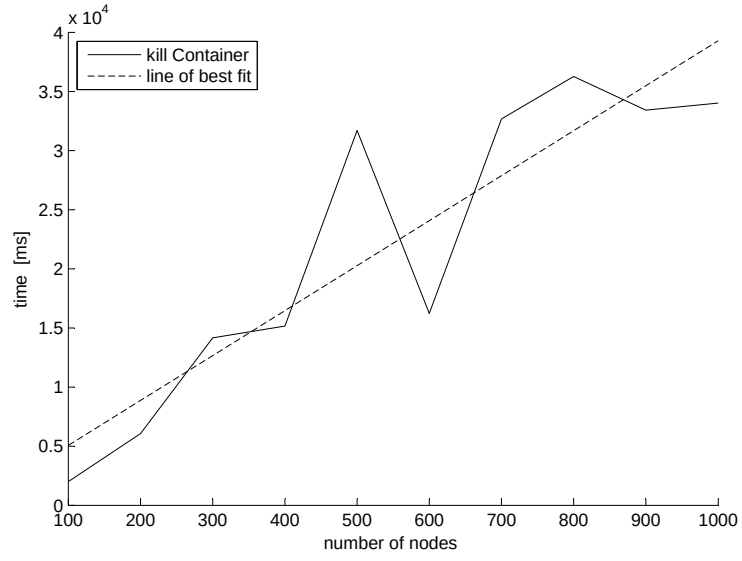


Abbildung 6.5: Kosten für das Beenden eines Knotens

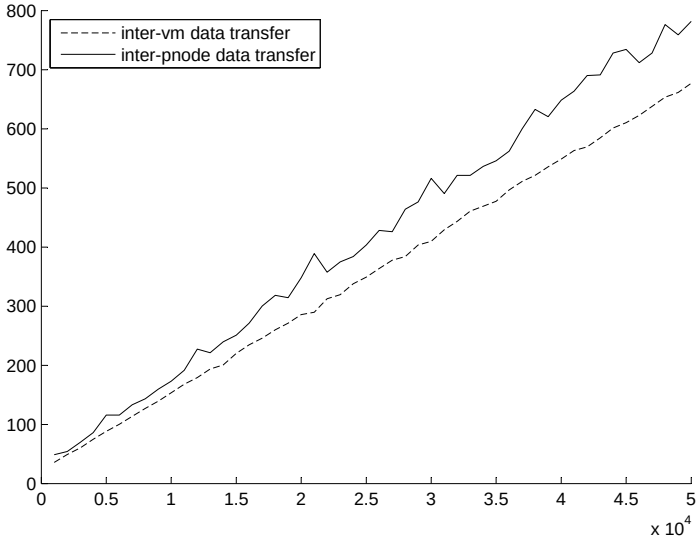


Abbildung 6.6: Kosten Transfer von Daten in Abhängigkeit vom zu übertragenden Datenvolumen

$suspend$	$6,078464[ms]$
$resume$	$3,602432[ms]$
$dumpMem$	$4,632[ms/mb]$
$dumpFrame$	$5,461[ms/mb]$
$dumpMac$	$26,238[ms/mb]$
$transfer_{intervm}$	$13,084[ms/mb]$
$transfer_{interpnode}$	$14,964[ms/mb]$
$undumpMem$	$1,780[ms/mb]$
$undumpFrame$	$2,124[ms/mb]$
$undumpMac$	$5,753[ms/mb]$
$killVNode$	$38,02[ms]$
$createBridge$	$200,0[ms]$
$destroyBridge$	$210,0[ms]$

Tabelle 6.1: κ Konstanten

dumpVnode	$60,324[ms]$
dumpNs	$40,991[ms]$
transferData	$40,452[ms]$
undumpVnode	$90,754[ms]$
undumpNs	$35,791[ms]$
kill	$1254,321[ms]$

Tabelle 6.2: c Konstanten

6.2 Performance des Optimierungsalgorithmus

In diesem Kapitel soll die Performance des in 4.6.2 vorgestellten Optimierungsalgorithmus evaluiert werden. Dazu wird die in 4.6.1 erläuterte Zielfunktion verwendet.

Zur Evaluierung werden 3 unterschiedliche Testszenarien betrachtet:

- ein Grid Szenario
- ein Waxman Graph Szenario
- ein Routerketten Szenario

Für alle 3 Szenarien wird das Konvergenzverhalten des Simulated Annealing Algorithmus untersucht. Dabei wird die Geschwindigkeit mit der der Wert des Kontrollparameters T sinkt variiert. Dies geschieht durch unterschiedliche Wahl der Konstante α (siehe 4.6.7).

Für jedes Testszenario steht dem Algorithmus ein Zeitfenster von 30 Sekunden für die Optimierung einer Platzierung zur Verfügung. Die Anzahl der Knoten variiert mit den Testszenarien zwischen 6400 und 50000 Knoten. Jeder Knoten nutzt zwischen 200 kbyte und 10 mbyte Arbeitsspeicher. Der Puffer der Netshaper Instanzen haben eine Größe zwischen 0 kbyte und 200 kbyte. Eine Paramterliste, in der zusätzlich Einstellungen für bestimmte Verbindungen zwischen Knoten abgelegt werden können, wird nicht verwendet. Ausgeführt wird die Optimierung auf einem Rechner mit 2 Kernen, die mit 2,4 Ghz getaktet sind.

Für die Berechnung erwarteter Experimentlaufzeiten wird ein virtuelles Zeitfenster von $t_{window} = 60s$ verwendet.

Jedes Testszenario wird mit jeweils 2 unterschiedlichen Testbed Konfigurationen evaluiert. In der einen Konfiguration besteht das Testbed aus 8 physikalischen Rechnern mit jeweils 8 virtuellen Maschinen ($8 * 8 = 64VMs$) und in der anderen Konfiguration aus 16 physikalischen Rechnern mit 8 virtuellen Maschinen pro physikalischem Knoten ($16 * 8 = 128VMs$).

Für jedes Testszenario wird der Optimierungsalgorithmus 5 mal für alle möglichen Konfigurationen ausgeführt. Die unten aufgeführten Graphen zeigen jeweils Mittelwerte.

6.2.1 Grid Szenario

Das erste betrachtete Testszenario ist ein Grid Szenario. In diesem werden 6400 Knoten in einem regulären quadratischen Gitter angeordnet. Jeder Knoten besitzt Verbindungen zu seinen direkten Nachbarn. Ein Knoten kann maximal 4 Verbindungen eingehen. Ein Beispiel für eine reguläres quadratisches Grid zeigt Abbildung 6.7.

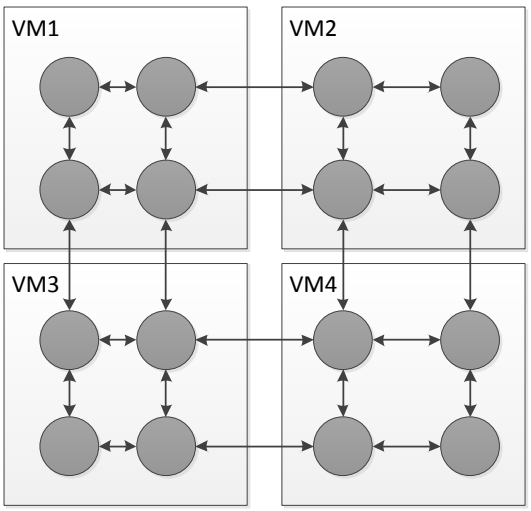


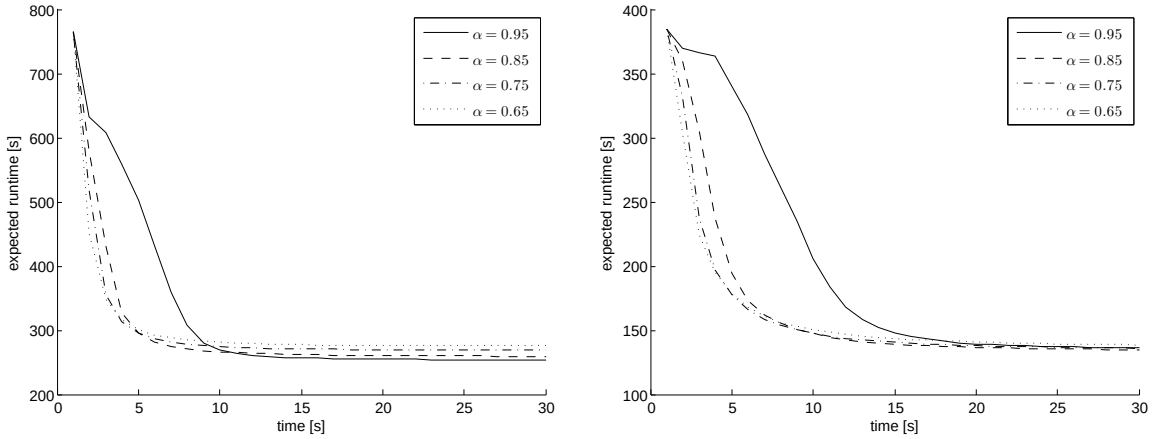
Abbildung 6.7: Grid Testszenario

Datenraten der Verbindungen zwischen virtuellen Knoten wurden zufällig zwischen 1 und 100 mbit gewählt. Die Last, die ein virtueller Knoten verursacht orientiert sich an der Nutzung der Verbindungen zu seinen Nachbarknoten.

Die 6400 Knoten wurde zufällig auf alle virtuellen Maschinen der Testumgebung verteilt:in der ersten Konfiguration der Testumgebung auf 64 und in der Zweiten auf 128 virtuelle Maschinen.

In den beiden Abbildungen weiter unten ist die Optimierung der Randomverteilung dargestellt. Links für die erste und rechts für die zweite Testumgebungskonfiguration. In Abständen von einer Sekunde wurde die unter der aktuellen Platzierung erwartete Experimentlaufzeit festgehalten.

Man erkennt, dass in der Testbed Konfiguration mit 64 VMs die Optimierung bereits nach 10 Sekunden konvergiert. Die Geschwindigkeit mit der die Optimierung konvergiert ist dabei maßgeblich von der Wahl des Faktors α abhängig. Für den höchsten Faktor $\alpha = 0.95$ vergeht z.B. wesentlich mehr Zeit bis sich Funktionswerte nur noch minimal ändern.



(a) Testbed mit 64 VMs

(b) Testbed mit 128 VMs

Abbildung 6.8: Performance des Optimierungsalgorithmus - Grid Szenario

Man erkennt aber auch, dass sich mit einem langsameren Abkühlungsprozess, also mit einer höheren Wahl von α, bessere Ergebnisse einstellen. In der Testbedkonfiguration mit 64 Knoten ergibt sich z.B. ein Unterschied von 14.5 Prozent in der erwarteten Experimentlaufzeit. In der anderen Konfiguration scheint die Optimierung nach 30 Sekunden noch nicht zu konvergieren.

Im der Testbedkonfiguation mit 64 Knoten sinkt der erwartete TDF von 12.69 auf 2.37. Für die Umsetzung der Platzierung, die die Kosten auf $\frac{1}{6}$ reduzieren, werden 511.79 Sekunden Rekonfigurationszeit benötigt.

6.2.2 Waxman Graph Szenario

Bei dem nächsten Testszenario handelt es sich um einen Random Graph. Dieser wurde mit Brite [MLMB01] erzeugt. Er besitzt 20000 Knoten, die zufällig nach einer Waxman Verteilung verbunden sind. Genau wie im Grid Testszenario liegen Datenraten für Links zwischen Knoten zwischen 1 und 100 mbit.

Die 20000 Knoten wurden gleich verteilt auf die zur Verfügung stehenden VMs verteilt. Ein Beispiel für einen Random Graph ist in Abbildung 6.9 gegeben.

In den beiden Abbildungen weiter unten ist wieder die erwartete Experimentlaufzeit über die Dauer der Optimierung abgetragen (für beide Testumgebungskonfigurationen). Man erkennt, dass sich für $\alpha = 0.95$ die Funktionwerte etwa 3 Sekunden zunächst wieder erhöhen. Für den

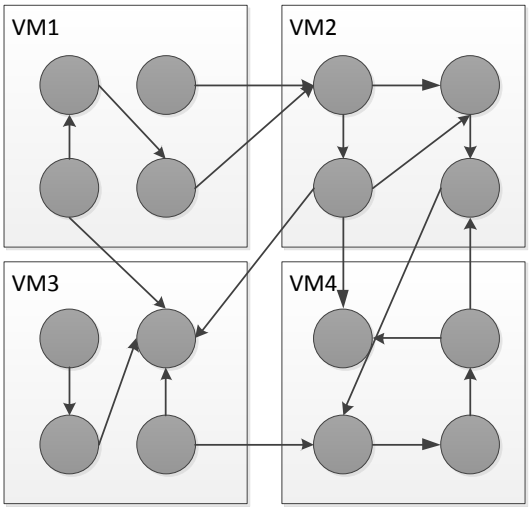
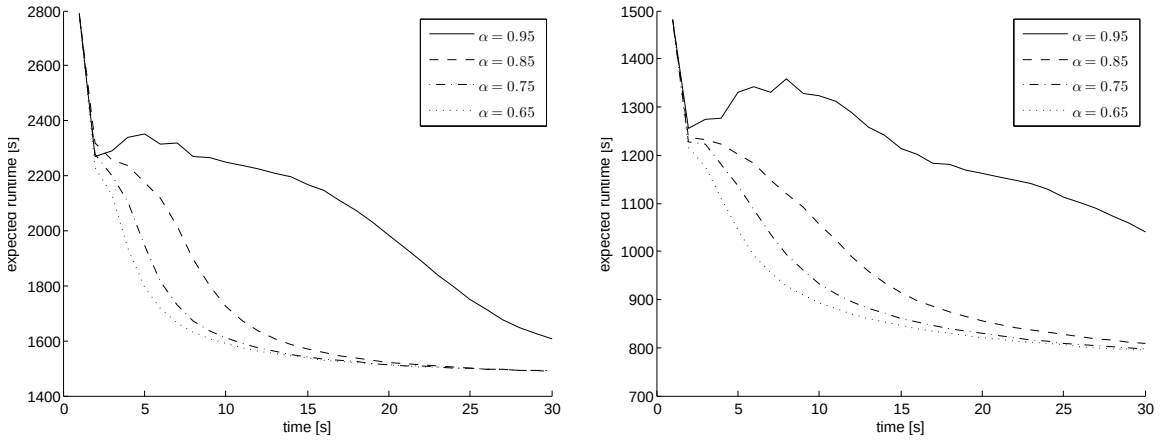


Abbildung 6.9: Waxman Testszenario

höchsten α Wert ist außerdem die erwartete Experimentlaufzeit am Ende der Optimierung sehr viel höher als bei den Anderen.



(a) Testbed mit 64 VMs

(b) Testbed mit 128 VMs

Abbildung 6.10: Performance des Optimierungsalgorithmus - Waxman Graph Szenario

Für eine Konvergenz reicht die Zeit von 30 Sekunden nicht aus. Durch die höhere Knotenanzahl (Faktor 3 zu Grid Szenario) und einer größeren Anzahl von Links ist der Suchraum sehr viel größer als im Grid Szenario. Trotzdem ist auch bei einer hohen Wahl von α eine deutliche

Verbesserung der Platzierung innerhalb von 30 Sekunden Optimierungszeit möglich. So sinkt die erwartete Experimentlaufzeit in der Testbedkonfiguration mit 128 Knoten beispielsweise um etwa 33%.

6.2.3 Routerketten Szenario

Im letzten Szenario soll eine Routerkette betrachtet werden. Ein Beispiel für eine Kette mit 16 Knoten ist in Abbildung 6.11 dargestellt.

In diesem Test besteht die Kette aus 50000 Knoten. Wobei sich an einen Ende der Kette ein Sender und am anderen Ende ein Empfänger befindet. Der Sender schickt Daten mit einer Datenrate von 100 mbit über die Routerknoten an den Empfänger. Jeder Link hat eine mittlere Datenrate von 100mbit.

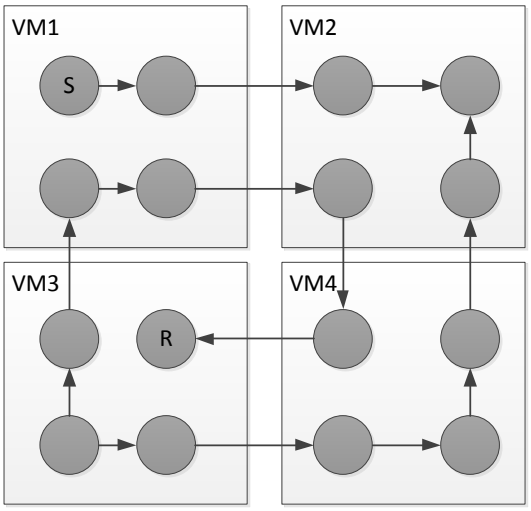
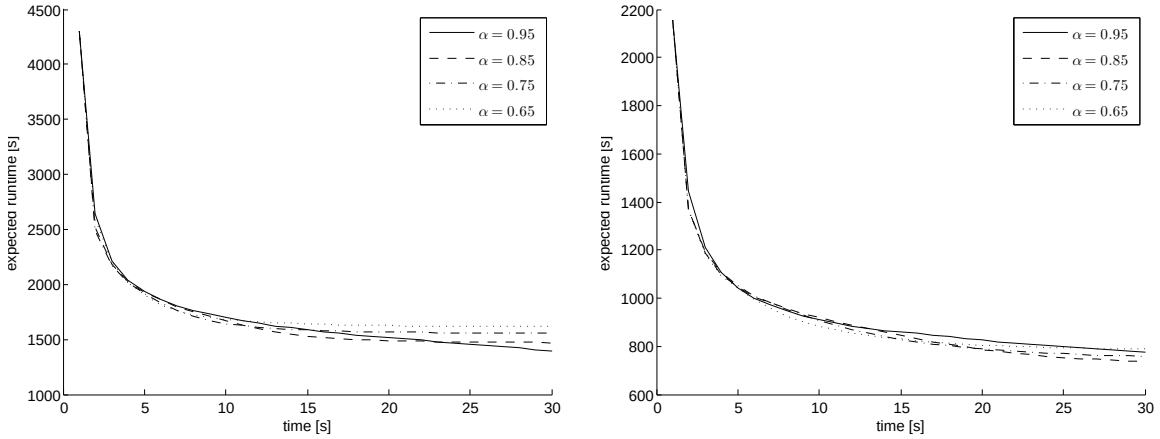


Abbildung 6.11: Routerketten Testszenario

Es wird davon ausgegangen, dass das Weiterleiten von Paketen die Router nur sehr wenig belastet. Jeder Router wird daher mit einer Last von 12000 cycles/s belegt. Nur beim Sender und Empfänger entstehen höhere Lasten.

Die Knoten werden wie in 6.11 gezeigt, gleichmäßig auf die virtuellen Maschinen derart verteilt, dass Knoten nur $inter - vm$ oder $inter - pc$ Verbindungen zu anderen Knoten eingehen können. Zwar sind Lasten der VMs durch diese Verteilung ungefähr gleich hoch. Die Gesamtlast bietet allerdings ein hohes Optimierungspotential.

Betrachtet man die in den beiden Abbildungen weiter unten dargestellte Performance des Algorithmus in diesem Testszenario so stellt man fest, dass sich die Graphen für unterschiedliche α Werte nur wenig unterscheiden. Im Gegensatz zum Waxman Szenario konvergiert der Testlauf mit dem höchsten α Wert nicht viel langsamer als die Anderen. Der Grund dafür liegt darin, dass zu Anfang fast alle Nachbarzustände mit hoher Wahrscheinlichkeit besser sind als der Aktuelle. Knoten werden nur zu virtuellen Maschinen migriert auf denen sich Knoten befinden, die mit ihnen verbunden sind. In diesem Szenario wird dadurch bei der Migration eines beliebigen Knotens zu Anfang aus einer inter-vm bzw. inter-pc Verbindung eine intra-vm Verbindung, was die Kosten erheblich senkt.



(a) Testbed mit 64 VMs

(b) Testbed mit 128 VMs

Abbildung 6.12: Performance des Optimierungsalgorithmus - Router Chain Szenario

In der Testbedkonfiguration mit 64 virtuellen Maschinen ließ sich der TDF von 71.61 auf 17.2 senken. Dafür sind 528 Sekunden Rekonfigurationszeit nötig.

6.2.4 Fazit

Schon nach geringer Zeit von 30 Sekunden sind auch in großen Szenarien von bis zu 50000 Knoten deutliche Verbesserungen der Platzierung möglich.

Steht einem auch bei komplexen Szenarien nur ein kleines Zeitfenster zur Optimierung zur Verfügung sollte der Abkühlungszeit möglichst kurz sein. Zwar wird dadurch die gefundene Lösung mit hoher Wahrscheinlichkeit nicht optimal sein, das Verfahren konvergiert allerdings in diesem Fall schneller. Wie im Waxman Szenario gezeigt, ließ sich mit niedrigerem α Wert ein besseres Ergebnis innerhalb des Zeitfensters von 30 Sekunden erzielen als mit hohem α Wert.

Im zentralen Ansatz der Optimierung wird in bestimmten virtuellen Zeitintervallen der aktuelle Zustand des Systems überprüft. Je nach Auslastung des System ist dabei das reale Zeitintervall länger oder kürzer.

Ist die aktuelle Platzierung verbesserungswürdig, so erfolgt eine Optimierung innerhalb dieses realen Zeitintervalls. Dem Optimierungsalgorithmus steht also je nach Höhe der Kosten, die das Testszenario aktuell verursacht, unterschiedlich viel Optimierungszeit zur Verfügung.

In der Regel steigen die Kosten mit der Größe des Testszenarios. Im Routerketten Szenario mit 50000 Knoten war z.B. der TDF sehr viel höher als im Grid Szenario mit 6400 Knoten. Dem Algorithmus steht also für große Szenarien mehr Zeit zur Verfügung. Daher ist im zentralen Ansatz eher noch mit besseren Resultaten zu rechnen als oben gezeigt. Dort stand dem Algorithmus unabhängig von der Größe des Szenarios immer nur 30 Sekunden zur Optimierung zur Verfügung.

6.3 Performance Neuplatzierung

In diesem Kapitel wird die Performance des Neuplatzierungsalgorithmus untersucht. Dazu werden 2 Testszenarien betrachtet. Für beide Szenarien wird ein Experiment durchgeführt, dass eine Laufzeit von 1200 Sekunden virtuelle Zeit hat.

In den zwei Testszenarien ändern sich alle 120 Sekunden Datenraten von bestimmten Verbindungen zwischen Knoten. Auf diese Veränderung reagiert der Neuplatzierungsalgorithmus z.B. mit der Umsetzung einer neuen Platzierung.

6.3.1 Sensor Szenario

Als erstes soll ein Sensorszenario betrachtet werden. Dieses besteht aus 400 Sensorknoten, die in einem regulären quadratischen Gitter angeordnet sind. Jeder Sensorknoten wird auf ein virtuellen Knoten abgebildet. Dieser wird zufällig auf eine von 32 virtuellen Maschinen verteilt(4 physikalische Rechner, 8 virtuelle Maschinen).

Jeder Sensor nimmt in bestimmten Zeitabständen Messdaten auf. Die aufgezeichneten Daten werden, z.B. zur Auswertung, an eine Senke geschickt. Eine Senke ist dabei ein Knoten des Sensornetzwerks, der alle Daten anderer Knoten sammelt.

Für die Übertragung der Messwerte wird ein Spannbaum erzeugt, dessen Wurzel die Senke ist. Entlang der Kanten des Spannbaums werden die Daten der Sensorknoten verschickt. Jeder Knoten leitet dazu seine eigenen und Daten seiner Kinder an den Vaterknoten weiter. Alle Knoten produzieren eine gewisse Datenmenge pro Zeit. In diesem Experiment beträgt die Datenrate pro Knoten 10 mbit.

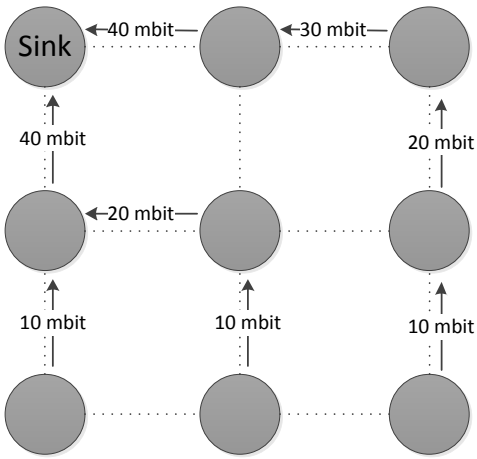


Abbildung 6.13: Beispiel Sensortestszenario

Ein Beispiel für einen Spannbaum und Datenraten an Kanten des Spannbaums ist in Abbildung 6.13 dargestellt.

Alle 120 Sekunden ändert sich in diesem Szenario die Senke. Die neue Senke wird zufällig aus allen Knoten ausgewählt. Nach der Änderung wird der Spannbaum neu bestimmt. Datenraten von Links zwischen Knoten ändern sich daraufhin.

In diesem Szenario bestimmt der Neuplatzierungsalgorihtmus in Intervallen von 12 Sekunden virtueller Zeit mittlere Lasten und Datenraten der Knoten. Auf Basis dieser Daten erstellt er mit dem „letzte Wert“ Ansatz (siehe Abschnitt 4.7) eine Prognose für die nächsten 12 Sekunden.

Diese Prognose dient als Input für die Optimierung der aktuellen Platzierung, die maximal 2 Sekunden virtueller Zeit dauern darf. Dabei wird bei der Optimierung das bis zur nächsten Aktualisierung übrig bleibende Zeitfenster nach der Optimierung betrachtet. Also $t_{window} = 10s$

In 6.14 ist der TDF über die virtuelle Zeit für die Ausführung des Experiments mit und ohne Migration dargestellt. Bleibt die Platzierung während der Ausführung des Experiments konstant verändert sich der TDF alle 120 Sekunden fast beliebig. Mal passt die initiale Platzierung besser mal schlechter.

Wird das Experiment mit Migration ausgeführt. Springt der TDF alle 120 Sekunden auf einen hohen Wert, bleibt dort für 14 Sekunden und fällt dann wieder auf einen niedrigen Wert ab.

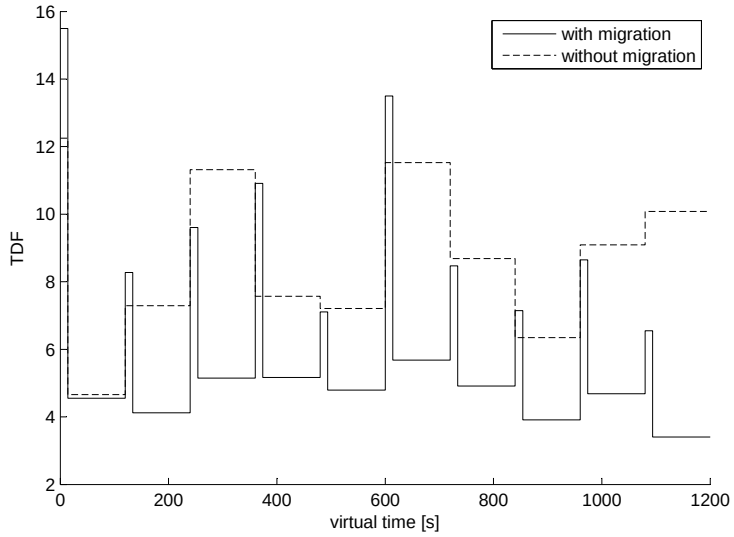


Abbildung 6.14: Sensor Szenario - TDF Verlauf

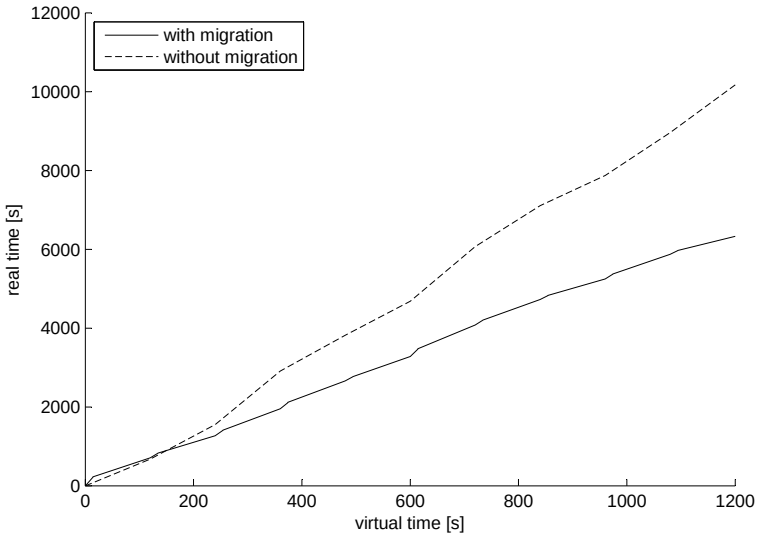


Abbildung 6.15: Sensor Szenario - Experimentlaufzeit

Dadurch dass Prognosen nur alle 12 Sekunden gemacht werden erhält der Neuplatzierungsalgorithmus erst 12 Sekunden nach der Änderung passende Werte. Auf deren Basis optimiert er 2 Sekunden und setzt dann die neue bessere Platzierung um. Dadurch sinkt der TDF wieder.

Während ohne Migration TDF Werte sehr stark schwanken bewegen sich Werte des TDF mit Migration um den Wert 5.

In 6.15 ist für beide Fälle also mit und ohne Migration die reale Experimentlaufzeit über die virtuelle Zeit abgetragen. Durch Migration lässt sich in diesem Beispiel die Experimentlaufzeit von etwa 10000 Sekunden auf 6000 Sekunden senken, was eine Zeitersparnis von 40 Prozent bedeutet.

6.3.2 Waxman Szenario

Für den zweiten Test soll das Waxman Szenario verwendet werden, das bereits schon in Abschnitt 6.2.2 vorgestellt wurde. Es umfasst 20000 Knoten, die zu Beginn des Experiments auf 64 virtueller Maschinen gleichmäßig verteilt werden (8 physikalische Rechner mit je 8 virtuellen Maschinen).

Alle 120 Sekunden werden in diesem Szenario zufällig Datenraten von Verbindungen zwischen Knoten geändert. Und zwar werden zufällig 80000 Links ausgewählt und deren Datenrate neu gesetzt. Nach der Veränderung liegen Datenraten immer noch zwischen 1 und 10 mbit.

Jede Sekunde wird in diesem Szenario eine Prognose für die nächsten 120 Sekunden erstellt. Der Neuplatzierungsalgorithmus geht davon aus das die gemessenen mittleren Lasten der letzten Sekunde für 120 Sekunden gleich bleiben (in einem realen Szenario ist natürlich von dieser Annahme abzuraten).

Optimiert werden soll in diesem Beispiel maximal 12 Sekunden virtuelle Zeit. Für die Optimierung wird ein Zeitfenster t_{window} von 108 betrachtet.

Abbildung 6.16 zeigt wieder den Verlauf des TDF mit und ohne Migration. Im Gegensatz zum Sensorszenario ist hier der TDF im Fall ohne Migration bis auf das erste Zeitintervall immer höher. Er nähert sich außerdem einem bestimmten Wert an. Mit zunehmender Anzahl von Veränderungen wird die Platzierung immer schlechter.

Mit Migration zeichnet sich ein ähnliches Bild wie im Sensorszenario ab. Allerdings sinkt der TDF durch die Optimierung relativ gesehen nicht so stark wie im Sensorszenario. Während sich im Sensorszenario die Werte des TDF halbierten wird hier nur eine Erniedrigung von etwa 13 % erreicht.

Dies macht sich auch in der Laufzeitersparnis bemerkbar. Diese ist für das Waxman Szenario wie man in Abbildung 6.17 sehen kann deutlich niedriger.

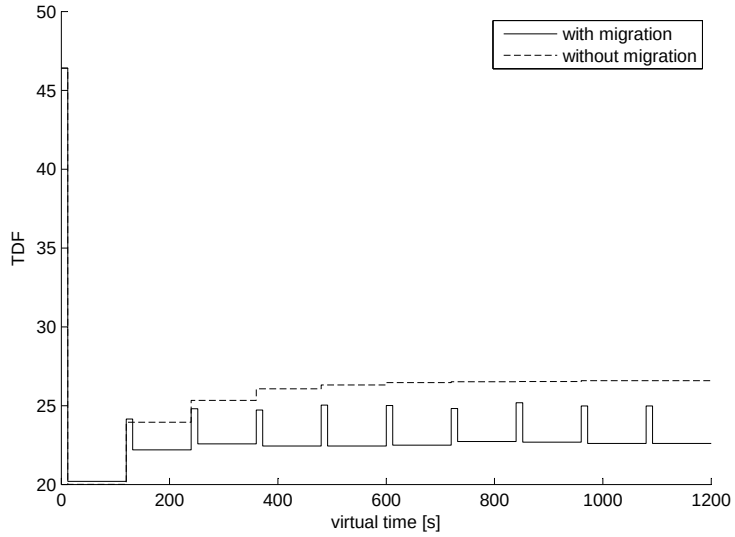


Abbildung 6.16: Waxman Szenario - TDF Verlauf

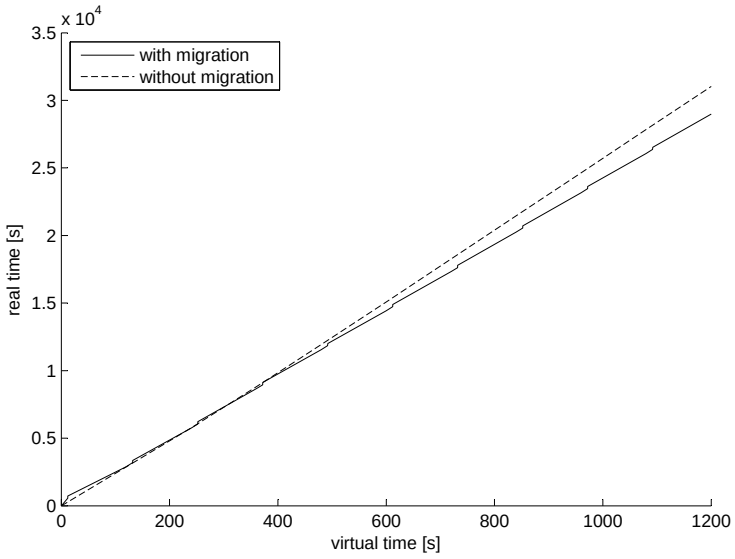


Abbildung 6.17: Waxman Szenario - Experimentlaufzeit

Werden in großen Szenarien zufällig Datenraten verändert, so scheinen sich diese Änderungen im Mittel auszugleichen. Dadurch ändert sich die Lastsituation und damit indirekt der TDF kaum. Eine Neuplatzierung von Knoten spart daher kaum Experimentlaufzeit. Selbst bei einem Zeitfenster von 120 Sekunden, welches realistisch gesehen viel zu hoch gewählt ist, kann kaum Laufzeit eingespart werden.

7 Zusammenfassung und Ausblick

Dieses Kapitel fasst die Diplomarbeit zusammen und stellt wesentliche Resultate der Arbeit vor. Des Weiteren wird kurz auf offene Probleme eingegangen.

7.1 Zusammenfassung

Das Ziel der Diplomarbeit war die Erweiterung einer zeitvirtualisierten Emulationsumgebung, namens TVEE, um eine Möglichkeit zur dynamischen Neuplatzierung virtueller Knoten.

Kapitel 1 enthielt eine Motivation und Beschreibung des Ziels der Diplomarbeit.

In Kapitel 2 wurde ein genauerer Blick auf die zeitvirtualisierte Emulationsumgebung geworfen. Neben der Architektur und Konfiguration der Emulationsumgebung, wurde vor allem auf bestehende Techniken zur Laufzeitminimierung eines Experiments eingegangen.

In Kapitel 3 stand die Taskmigration in verteilten und parallelen Systemen, ein einfacheres, aber artverwandtes Problem, im Zentrum der Betrachtung.

Kapitel 4.2 widmete sich dann dem eigentlichen Problem: der dynamische Neuplatzierung von virtuellen Knoten. Zunächst wurden wesentliche Bestandteile der Neuplatzierung herausgearbeitet. Hierzu zählen sowohl die Optimierung einer aktuellen Platzierung von virtuellen Knoten also auch die Umsetzung einer neuen Platzierung (Rekonfiguration der TVEE).

Im Folgenden wurden Mechanismen zur transparenten Rekonfiguraiton vorgestellt. Diese umfassten z.B. die Migration von virtuellen Knoten und die Anpassung der Schicht 2 Architektur.

Zur Optimierung einer Platzierung mussten zukünftige Kosten alternativer Platzierung abgeschätzt werden. Dazu wurden zwei Kostenmodelle definiert: das Kommunikationskostenmodell und das Rekonfigurationskostenmodell. Auf Basis dieser beiden Modelle wurde eine Zielfunktion für die Optimierung einer Platzierung entwickelt.

Für die Optimierung kamen unterschiedliche Algorithmen in Frage. Unter anderem Hill Climbing, evolutionäre Algorithmen, und Simulated Annealing. Aufgrund der hohen Flexibilität wurde sich für einen Simulated Annealing Ansatz entschieden.

Das oben erwähnte Kommunikationskostenmodell basierte auf Prognosen zu mittleren Datenraten von Verbindungen zwischen virtuellen Knoten und deren Lasten. Für die Voraussage zukünftiger Werte wurden verschiedene Ansätze zur *One Step Ahead Prediction* vorgestellt.

An Ende des Kapitels wurde die mögliche Lage des Optimierungs- und des Rekonfigurationsalgorithmus diskutiert. Für die Optimierung wurden mehrere verteilte und ein zentraler Ansatz vorgestellt und gegeneinander abgewogen. Der zentrale Ansatz stellte sich als am erfolgversprechendsten heraus.

In Kapitel 5 wurde auf die Implementierung der zentralen Neuplatzierung eingegangen. Dabei standen Details im Vordergrund, die in vorigen Abschnitten noch nicht ausgeführt wurden. So wurde z.B. näher auf die für die Rekonfiguration der TVEE benutzten Tools eingegangen. Außerdem wurde auf aktuelle Probleme bei der Rekonfiguration aufmerksam gemacht.

In Kapitel 6 wurden zunächst Konstanten des Rekonfigurationskostenmodells bestimmt. Anschließend wurde die Performance des Optimierungsalgorithmus in 3 unterschiedlichen Szenarien untersucht. Es stellte sich heraus, dass bereits nach geringer Optimierungszeit von wenigen Sekunden auch für große Szenarien von 20000 Knoten sich erwartete Experimentlaufzeiten stark senken lassen.

Abschließend wurden in 2 unterschiedlichen Szenarien die Auswirkungen der dynamischen Neuplatzierung auf die Experimentlaufzeit untersucht. In beiden Szenarien ließ sich die Experimentlaufzeiten senken; im Sensorszenario um fast 40 Prozent.

Zum Schluss sollen noch offene Probleme diskutiert werden.

7.2 Offene Probleme und Ausblick

Zur Zeit entstehen bei der Rekonfiguration der TVEE hohe Kosten für eigentlich günstige Operationen (wie z.B. für das Beenden eines virtuellen Knotens und das Setzen eines virtuellen Knotens in den Haltezustand). Durch die starke Verlangsamung der Zeit während der Rekonfiguration verzögern sich Timer Events. Für eine effiziente Rekonfiguration sollten sich daher Mechanismen überlegt werden, auf welche Weise diese unnötigen, durch Timer entstehenden Kosten gesenkt werden können.

Die Rekonfiguration sollte mit Tools wie Netperf und Iperf als SuTs getestet werden. Eine transparente Rekonfiguration mit diesen beiden Tools war allerdings nicht möglich (siehe Abschnitt 5.4.2). Aus diesem Grund wäre es sinnvoll zu evaluieren, ob die aufgetretenen Probleme rein Tool abhängig sind.

Des Weiteren sollten die vorgestellten Verfahren zur *one step ahead prediction* in unterschiedlichen Szenarien zu evaluiert werden. Eine gute Prognose zukünftiger Lasten und Datenraten

ist von entscheidender Bedeutung für die Qualität ermittelter neuer Platzierungen. In diesem Zusammenhang wäre es sinnvoll Auswirkungen schlechter Prognosen auf die Experimentlaufzeit zu untersuchen. Bei schlechten Prognosen könnte sich im schlimmsten Fall die Experimentlaufzeit sogar erhöhen.

Literaturverzeichnis

[AH06] G. Apostolopoulos, C. Hasapis. A Cluster of Virtual Machines for Robust, Detailed, and High-Performance Network Emulation. *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, 14:11–14, September 2006. (Zitiert auf Seite 10)

[BTA+] L. Bajaj, M. Takai, R. Ahuja, K. Tang, R. Bagrodia, M. Gerla. GloMoSim: A Scalable Network Simulation Environment. (Zitiert auf Seite 9)

[CCR+03] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003. doi:http://doi.acm.org/10.1145/956993.956995. (Zitiert auf Seite 9)

[CPJL98] F. C., M. P., D. J.-L. Data-parallel load balancing strategies. *Parallel Computing*, 24:1665–1684(20), October 1998. doi:doi:10.1016/S0167-8191(98)00049-0. URL http://www.ingentaconnect.com/content/els/01678191/1998/00000024/00000011/art00049. (Zitiert auf Seite 25)

[CS03] M. Carson, D. Santay. NIST Net: a Linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, 2003. doi:http://doi.acm.org/10.1145/956993.957007. (Zitiert auf Seite 9)

[DO00] P. A. Dinda, D. R. O'Hallaron. Host load prediction using linear models. *Cluster Computing*, 3:265–280, 2000. URL http://dx.doi.org/10.1023/A:1019048724544. 10.1023/A:1019048724544. (Zitiert auf Seite 66)

[ELZ86] D. L. Eager, E. D. Lazowska, J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.*, 12:662–675, 1986. URL http://portal.acm.org/citation.cfm?id=5527.5535. (Zitiert auf Seite 26)

[FTO] J. Flynn, H. Tewari, D. OMahony. JEmu: A Real Time Emulation System for Mobile Ad Hoc Networks. (Zitiert auf Seite 10)

[GHR] A. Grau, K. Herrmann, K. Rothermel. NETplace: Efficient Runtime Minimization of Network Emulation Experiments. (Zitiert auf den Seiten 10, 16, 44 und 54)

[GHR09] A. Grau, K. Herrmann, K. Rothermel. Efficient and Scalable Network Emulation using Adaptive Virtual Time. *Proceedings of 18th Internatonal Conference on Computer Communications and Networks*, 18:1–6, Aug 2009. (Zitiert auf Seite 10)

[GMHR08] A. Grau, S. Maier, K. Herrmann, K. Rothermel. Time Jails: A Hybrid Approach to Scalable Network Emulation. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, PADS '08, pp. 7–14. IEEE Computer Society, Washington, DC, USA, 2008. doi:http://dx.doi.org/10.1109/PADS.2008.19. URL http://dx.doi.org/10.1109/PADS.2008.19. (Zitiert auf den Seiten 13 und 15)

[Gnu] Gnutella. http://rfc-gnutella.sourceforge.net/. (Zitiert auf Seite 9)

[GRL05] S. Guruprasad, R. Ricci, J. Lepreau. Integrated Network Experimentation using Simulation and Emulation. *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, 2005. (Zitiert auf Seite 9)

[GYM+06] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, G. M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. *3rd Symposium on Networked Systems Design & Implementation*, pp. 87–100, 2006. (Zitiert auf den Seiten 10 und 15)

[Hem05] S. Hemminger. Network Emulation with NetEm, 2005. (Zitiert auf Seite 9)

[Kes88] S. Keshav. REAL : A Network Simulator, 1988. (Zitiert auf Seite 9)

[Kir84] S. Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34:975–986, 1984. URL http://dx.doi.org/10.1007/BF01009452. 10.1007/BF01009452. (Zitiert auf den Seiten 23, 61 und 62)

[LJM88] J. Lam, D. Jean-Marc. Performance of a new annealing schedule. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, DAC '88, pp. 306–311. IEEE Computer Society Press, Los Alamitos, CA, USA, 1988. URL http://portal.acm.org/citation.cfm?id=285730.285780. (Zitiert auf Seite 61)

[LK87] F. Lin, R. Keller. The Gradient Model Load Balancing Method. *IEEE Transactions on Software Engineering*, 13:32–38, 1987. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.1987.232563. (Zitiert auf Seite 25)

[LRCM95] E. Luque, A. Ripoll, A. Cortes, T. Margalef. A distributed diffusion method for dynamic load balancing on parallel computers. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:43, 1995. doi:http://doi.ieeecomputersociety.org/10.1109/EMPDP.1995.389156. (Zitiert auf Seite 26)

[MHR] S. Maier, D. Herrscher, K. Rothermel. On Node Virtualization for Scalable Network Emulation. (Zitiert auf Seite 10)

[MI] D. Mahrenholz, S. Ivanov. Real-Time Network Emulation with ns-2. (Zitiert auf Seite 10)

[MLMB01] A. Medina, A. Lakhina, I. Matta, J. Byers. BRITE: An Approach to Universal Topology Generation. In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS01),*. 2001. (Zitiert auf Seite 94)

[MRR+53] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, E. Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953. doi:10.1063/1.1699114. URL http://dx.doi.org/10.1063/1.1699114. (Zitiert auf Seite 62)

[NP] NET-Project. http://net.informatik.uni-stuttgart.de/. (Zitiert auf Seite 13)

[NSNK97] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, R. H. Katz. Trace-based mobile network emulation. *SIGCOMM Comput. Commun. Rev.*, 27(4):51–61, 1997. doi:http://doi.acm.org/10.1145/263109.263140. (Zitiert auf Seite 9)

[Ril03] G. F. Riley. The Georgia Tech Network Simulator. In *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pp. 5–12. ACM, New York, NY, USA, 2003. doi:http://doi.acm.org/10.1145/944773.944775. (Zitiert auf Seite 9)

[RSV91] F. Romeo, A. Sangiovanni-Vincentelli. A theoretical framework for simulated annealing. *Algorithmica*, 6:302–345, 1991. URL http://dx.doi.org/10.1007/BF01759049. 10.1007/BF01759049. (Zitiert auf Seite 61)

[SKS92] N. G. Shivaratri, P. Krueger, M. Singhal. Load Distributing for Locally Distributed Systems. *Computer*, 25:33–44, 1992. doi:http://doi.ieeecomputersociety.org/10.1109/2.179115. (Zitiert auf den Seiten 23 und 24)

[Whi84] S. R. White. Concepts of scale in simulated annealing. In *American Institute of Physics Conference Series*, volume 122 of *American Institute of Physics Conference Series*, pp. 261–270. 1984. doi:10.1063/1.34823. (Zitiert auf Seite 63)

[WLR89] M. Willebeek-LeMair, A. P. Reeves. A general dynamic load balancing model for parallel computers. In *Tech. Rep. EE-CEG-89- 1, Cornell School of Electrical Engineering*. 1989. (Zitiert auf Seite 24)

[xen] *Xen User's Manual*. (Zitiert auf Seite 15)

[YFS03] L. Yang, I. Foster, J. M. Schopf. Homeostatic and Tendency-Based CPU Load Predictions. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pp. 42.2–. IEEE Computer Society, Washington, DC, USA, 2003. URL http://portal.acm.org/citation.cfm?id=838237.838601. (Zitiert auf den Seiten 65 und 66)

[ZSIo6] Y. Zhang, W. Sun, Y. Inoguchi. CPU Load Predictions on the Computational Grid *. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, pp. 321–326. IEEE Computer Society, Washington, DC, USA, 2006. doi:http://dx.doi.org/10.1109/CCGRID.2006.27. URL http://dx.doi.org/10.1109/CCGRID.2006.27. (Zitiert auf Seite 66)

Alle URLs wurden zuletzt am 21.01.2011 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Sebastian Bartmann)