

Verfügbarkeit und Verwaltung virtueller Server mit Xen und Reliable Server Pooling verbessern

Johannes Formann

Inhaltsverzeichnis

I	Einleitung	1
1	Vorwort	1
2	Leitidee	1
II	Techniken	1
3	virtuelle Server	1
3.1	Definition	1
3.2	Aufbau	2
3.2.1	Betriebssystemvirtualisierung	2
3.2.2	Systemvirtualisierung	3
3.3	Existierende Virtualisierungslösungen	5
3.4	Einsatzgebiete für virtuelle Servern	5
3.5	Nachteile von virtuellen Servern	6
4	Xen	6
4.1	Struktur	7
4.2	Optionen	7
4.3	libvirt	8
5	Storage Area Network (SAN)	8
5.1	iSCSI	8
5.2	ATA over Ethernet (AoE)	9
5.3	Network Block Device (NBD)	9
6	Reliable Server Pooling (RSerPool)	9
6.1	Terminologie & Funktionsverteilung	9
6.2	Aufbau	10
6.3	Der Prototyp - rsplib	11
III	Implementierung	11
7	Aufbau des Prototypen	11
7.1	Architektur	11
7.2	Annahmen / Festlegungen	11
7.3	Inhalte der Nachrichten	12
7.4	Lastdefinition	13
8	Implementierung	13

8.1	Pool User / Client	14
8.2	Pool Element	15
8.2.1	Registrierung in sever.cc	15
8.2.2	Diensterbringung in standardservices.cc und standardservices.h	15
IV	Tests	20
9	Versuchsaufbau	20
9.1	verwendete Hardware	20
9.2	genutzte Software	20
9.3	Konfiguration	21
9.3.1	Lastverteilung	21
9.3.2	Failovertests	22
9.4	Messung	22
10	Auswertung der Lasttests	23
10.1	Beobachtungen	23
10.2	Ergebnisse der Messungen	25
10.3	Vergleich der Lastdefinitionen	28
10.4	Bewertung	29
11	Auswertung der Failovertests	29
V	Zusammenfassung	30
12	Zusammenfassung & Ausblick	30
12.1	Zusammenfassung	30
12.2	Offene Themen	31
12.3	Kommentar	32
VI	Anhang	33
A	Läufe zum Testen der Lastverteilung im PC-Pool	33
A.1	Startreihenfolge und Wartezeiten	33
A.2	Weitere Parameter der Läufe	34
A.3	vif-Route	34
A.4	Lastgenerator	35
B	Quelltext	36
B.1	Pakete - XenServpackets.h	36
B.2	Client - XenServ.cc	37
B.3	Server - Pool Element	40

B.3.1	Registrierung in server.cc	40
B.3.2	standardservices.h	44
B.3.3	standardservices.cc	45

Literatur		53
------------------	--	-----------

Teil I

Einleitung

1 Vorwort

In dieser Ausarbeitung wird die Entwicklung des Prototypen zur Verwaltung von virtuellen Servern mittels Reliable Server Pooling beschrieben.

Im folgenden wird zunächst kurz die zugrunde liegende Idee vorgestellt. Im Teil II werden die verwendete Techniken/Software präsentiert, und im III Teil wird die Implementierung erklärt. Im Teil IV werden die durchgeführten Tests vorgestellt und Ausgewertet, um zum Schluss im Teil V ein Resümee gezogen.

Ich bedanke mich für die Unterstützung seitens des Lehrstuhls Technik der Rechner-netze, insbesondere bei Herrn Dreihholz.

2 Leitidee

Die Idee für diese Arbeit ist, zu überprüfen in wie weit das Reliable Server Pooling Framework für die Verwaltung von virtuellen Servern (siehe Kapitel 3) nutzbar ist.

Das Reliable Server Pooling Framework ist ein IETF-Standardisiertes Framework, welches Mechanismen zur Lastverteilung und Failover im Fehlerfall anbietet. Diese Fähigkeiten sollen bei der Verwaltung von virtuellen Servern genutzt werden, und ein einfacheres Management zu bekommen, um eine größere Verfügbarkeit der virtuellen Server.

Teil II

Techniken

3 virtuelle Server

3.1 Definition

Im Rahmen dieser Arbeit ist ein virtueller Server, auch kurz vServer genannt, als eine Instanz einer Betriebssystem- oder Hardwarevirtualisierung¹ definiert.

Ein virtueller Server stellt sich aus Benutzersicht fast² wie ein echtes System dar, läuft aber unter der Kontrolle eines Virtualisierungssystem, welches den Betrieb mehrerer virtueller Server auf einem Server erlaubt.

¹Die Virtualisierung führt eine Abstrahierungsschicht ein, welche es erlaubt die Ressourcen eines Computers aufzuteilen.

²Je nach verwendeter Technik gibt es einige Einschränkungen. Diese Betreffen meist die Hardwareunterstützung oder den Betriebssystemkern.

3.2 Aufbau

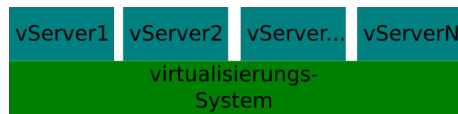


Abbildung 1: Basisstruktur von virtuellen Servern

Der prinzipielle Aufbau eines Virtualisierungssystem, zum Betrieb von virtuellen Servern, ist in Abbildung 1 dargestellt. Auf der Hardware³ läuft das, hier noch nicht näher definierte, Virtualisierungssystem. Das Virtualisierungssystem übernimmt die folgenden Aufgaben:

- Die Ressourcenzuweisung zu den virtuellen Servern. (RAM, CPU-Zeit, Speichermedien, Zugriff auf das Netzwerk & evtl. Peripheriegeräte)
- Die Isolation zwischen den verschiedenen virtuellen Servern. Die virtuellen Server dürfen nicht in der Lage sein Ressourcen zu ändern die einem anderen virtuellen Server exklusiv zugeteilt wurden.
- Die Verwaltung der virtuellen Server. (starten, stoppen u.U. weitere)

Zur Realisierung des Virtualisierungssystems gibt es es verschiedene Ansätze, die wichtigsten werden im folgenden vorgestellt.

3.2.1 Betriebssystemvirtualisierung

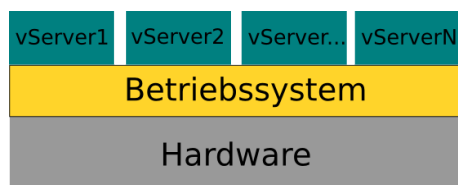


Abbildung 2: Struktur von Betriebssystemvirtualisierung

Bei der Betriebssystemvirtualisierung läuft ein gemeinsam genutzter Betriebssystemkern, welcher die unterschiedlichen virtuellen Server isoliert, so dass jeder virtuelle Server nur die ihm zugeordneten Daten, Programme und Prozesse sehen und nutzen kann. Der schematische Aufbau ist in Abbildung 2 dargestellt.

Die Folgenden Lösungen sind in diesen Ansatz zuzuordnen: OpenVZ/Virtuozzo, CTX, BSD-Jails, Solaris-Zones.

Die Vorteile sind der sehr geringe Overhead und die Möglichkeit Ressourcen dynamisch nach Bedarf zu verteilen (insbesondere RAM/Swap).

³Es gibt auch (nicht X86) Systeme, die ein einfaches Virtualisierungssystem in Hardware unterstützen.

Die Nachteile sind, dass man auf eine bestimmte Betriebssystemversion für alle virtuellen Server auf dem System festgelegt ist, dass häufig die Funktionalität des Kernels für die virtuellen Server eingeschränkt ist⁴ und dass durch den geteilten Kernel die Isolation im Fehlerfalle relativ gering ist.

3.2.2 Systemvirtualisierung

Bei der Systemvirtualisierung wird für jeden virtuellen Server ein eigenes Betriebssystem gestartet, welche voneinander unabhängig sind. Dabei sind meistens verschiedene Betriebssysteme, zumindest verschiedene Betriebssystemversionen, auf einem Virtualisierungssystem möglich.

Man unterscheidet dabei nach dem Funktionsumfang zwischen der vollständigen Systemvirtualisierung, auch Hardwarevirtualisierung genannt, und der teilweisen Systemvirtualisierung, auch Paravirtualisierung genannt.

Nach der Implementierung unterscheidet man zwischen Typ-1 Virtualisierungslösungen, bei denen der Hypervisor⁵ direkt auf der Hardware aufsetzt, und Typ-2 Virtualisierungslösungen, die auf ein laufendes Betriebssystem aufsetzen.

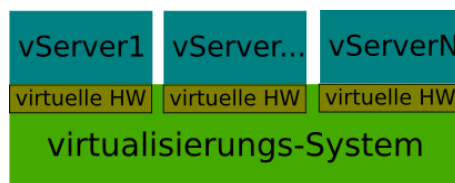


Abbildung 3: Schema für vollständige Virtualisierung

vollständige Systemvirtualisierung / Hardwarevirtualisierung Bei der vollständigen Virtualisierung emuliert das Virtualisierungssystem für die virtuellen Server das Verhalten von real existierender Hardware. Emuliert werden dabei nur die Ein- und Ausgabegeräte, wie auch in Abbildung 3 skizziert, nicht die CPU, sonst handelt es sich um Systememulationen. Daher laufen die Gastsysteme ohne weitere Anpassungen, vorausgesetzt es gibt Treiber für die bereitgestellte „Hardware“⁶.

Der Vorteil an diesen Vorgehen ist, dass Systeme ohne Anpassungen installiert oder von bestehenden Servern kopiert werden können, und auch Betriebssysteme laufen, die nicht auf ein Virtualisierungssystem angepasst werden können⁷.

Der Nachteil an diesen Vorgehen ist, dass durch die Emulierung der Hardware relativ viele Ressourcen⁸ benötigt werden.

⁴z.B. Module laden, logische Geräte einrichten, Imagedateien mounten, Netzwerkfilter einrichten

⁵Der wichtigste Teil einer Virtualisierungslösung, regelt den Zugriff auf die Ressourcen

⁶Es wird meistens sehr verbreitete Hardware emuliert, so dass in der Regel Treiber vorhanden sind.

⁷z.B. weil der Quelltext nicht verfügbar ist (Windows) oder sich der Aufwand nicht lohnen würde (Linux 2.2 Kernel)

⁸vor allen CPU-Leistung, aber auch RAM.

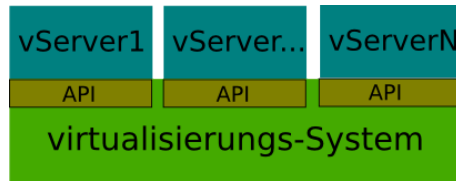


Abbildung 4: Schema für teilweise Virtualisierung

teilweise Systemvirtualisierung / Paravirtualisierung Bei der teilweisen Systemvirtualisierung stellt das Virtualisierungssystem, wie in Abbildung 4 zu sehen, den virtuellen Servern ein API zur Kommunikation bereit. Dieses API ermöglicht einen effizienten Datenaustausch zwischen den virtuellen Servern und dem Virtualisierungssystem, muss aber von den jeweiligen Betriebssystemen unterstützt werden.

Der Vorteil ist der im Vergleich zur vollständigen Virtualisierung deutlich reduzierte Overhead.

Der Nachteil ist, dass das Betriebssystem, welches in den virtuellen Server laufen soll, zur Unterstützung der API modifiziert werden muss, was nicht immer möglich ist⁹.

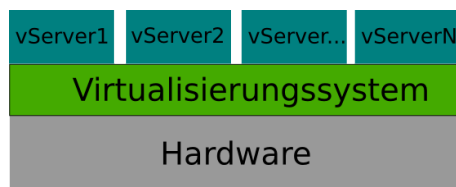


Abbildung 5: Schema für Typ-1 Virtualisierungslösung

Typ-1 Virtualisierung / Typ-1 Hypervisor Bei den Typ-1 Virtualisierungslösungen, teilweise auch „Bare Metal Hypervisor“ genannt, läuft der Hypervisor¹⁰ direkt auf der Hardware. Das Schema dazu ist in Abbildung 5 dargestellt.

Dieser Ansatz hat als Vorteil, dass eine sehr gute Isolation der virtuellen Server möglich ist, und die Ressourcenzuweisung sehr gut kontrolliert werden kann. Durch das Laufen direkt auf der Hardware ist der Overhead verhältnismäßig gering.

Als Nachteil ist zu sehen, dass der Hypervisor etwas komplexer wird, da die Hardware vom Hypervisor verwaltet werden muss.

Typ-2 Virtualisierung Bei den Typ-2 Virtualisierungslösungen, teilweise auch „Hosted Hypervisor“ genannt, läuft der Hypervisor als Programm in einem Betriebssystem. Das Schema dazu ist in Abbildung 6 dargestellt.

Dieser Ansatz hat als Vorteile, dass die Installation meistens einfach ist, und die Virtualisierungslösung keine Hardwaretreiber benötigt.

⁹zu hoher Aufwand, oder keinen Zugriff auf den Quellcode

¹⁰der Kern der Virtualisierungslösung, der die Ressourcenzuteilung & Kontrolle übernimmt

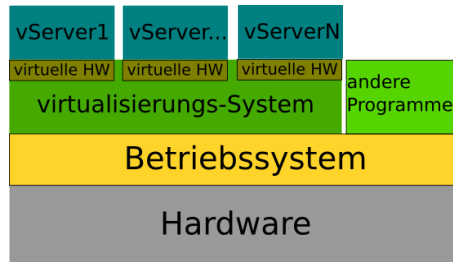


Abbildung 6: Schema für Typ-2 Virtualisierungslösung

Als Nachteil ist zu sehen, dass durch das Betriebssystem und durch das Laufen in ohne Betriebssystemprivilegien¹¹ ein deutlich höherer Overhead vorhanden ist.

3.3 Existierende Virtualisierungslösungen

Die Tabelle 1 ordnet verschiedene auf dem Markt für X86-Systeme befindliche Virtualisierungslösungen den zuvor beschriebenen Eigenschaften zu.

	Hardwarevirtualisierung	Paravirtualisierung
Typ-1	VMWare ESX Server Xen mit Systemen die Intel VT oder AMD-V Technik beherrschen	Xen
Typ-2	VMware Workstation/Server/Player Virtual PC von Microsoft Parallels Workstation QEMU mit Modul qemu (sonst Emulator)	User-Mode-Linux

Tabelle 1: Zurodnung existierender Virtualisierungstechniken

3.4 Einsatzgebiete für virtuelle Servern

Der Einsatz von virtuellen Server ermöglicht eine Konsolidierung von Servern und eine flexiblere Verwaltung der Ressourcen.

Serverkonsolidierung Das Ziel bei der Serverkonsolidierung ist es, viele wenig ausgelastete oder alte Server auf wenige leistungsfähige Server zusammenzufassen, und dadurch Betriebskosten¹² zu sparen.

Virtuelle Server werden aus verschiedenen Gründen gebraucht:

¹¹bei X86 Hardware die Ring-0 Rechte

¹²z.B. Strom, Klimatisierung, Wartung

- Verschiedene Dienste sollen auf unterschiedlichen Systemen laufen, um die Gefahr von Fehlerausbreitungen¹³ zu reduzieren.
- Es handelt sich um Produktions-, Test- und Backupsystem.
- Verschiedene Dienste setzen inkompatible Betriebssysteme voraus, oder sind anderweitig inkompatibel¹⁴.

Dabei muss man im Einzelfall schauen, welche Ansprüche an die Lösung gestellt werden. Falls zum Beispiel eine sehr hohe Isolation der unterschiedlichen virtuellen Server gefordert wird, fallen die Betriebssystemvirtualisierungslösungen häufig aus.

Flexible Verwaltung der Ressourcen Durch das Abstrahieren von der Hardware können virtuelle Server viel leichter dem Bedarf angepasst werden, sofern entsprechende Ressourcen vorhanden sind. Beispielsweise kann einem virtuellen Server ein höherer Anteil an CPU & RAM zugewiesen werden, falls dieser hoch belastet ist, oder auch relativ einfach auf einen anderen, leistungsfähigeren, Server umgezogen werden.

Auch können virtuelle Server oft sehr schnell bei Bedarf installiert werden, da das Virtualisierungssystem verschiedene Hilfen dafür anbietet¹⁵.

3.5 Nachteile von virtuellen Servern

Virtuelle Server bringen aber auch einige Nachteile mit sich:

- Die Virtualisierungslösung erzeugt zusätzlichen Overhead
- Bei einem Hardwaredefekt sind, aufgrund der Serverkonsolidierung, in der Regel mehr Systeme betroffen.
- Es gibt zusätzlichen Administrations & Verwaltungsaufwand
 - Installation und Wartung des Virtualisierungssystems
 - Planen der Ressourcenzuteilung, berücksichtigen welche virtuellen Server zusammen auf einen Server passen (von den Ressourcenbedarf), und evtl. auch welche nicht zusammen auf einen Server dürfen (z.B. Produktivsystem und Backupsystem)

4 Xen

Xen[1, 2] ist ursprünglich ein Typ-1 System zur teilweisen Systemvirtualisierung, mit aktuellen Prozessoren¹⁶ ist auch vollständige Systemvirtualisierung möglich. Xen wurde an der Universität von Cambridge ursprünglich für das XenoServers-Projekt entwickelt[5].

¹³d.h. wenn Dienst A ein Problem bekommt, und dabei sogar evtl. das Betriebssystem beeinträchtigt, soll Dienst B weiter funktionieren

¹⁴z.B. wollen gleiche Netzwerkports öffnen

¹⁵z.B. die Möglichkeit „Vorlagen“ zu erstellen, bei denen dann nur noch die jeweils Individuellen Parameter gesetzt werden müssen, wie der Name des Systems, oder die IP-Nummer

¹⁶die Intels Vanderpooloder AMDs Pacifica Technik unterstützen

Da Xen als Virtualisierungslösung für dieses Projekt gewählt wurde, wird es im folgenden vorgestellt.

4.1 Struktur

Der Xen-Hypervisor läuft als Typ-1 Hypervisor direkt auf der Hardware. Um den Xen-Hypervisor klein zu halten, ist in diesen nur die Speicherverwaltung, das Prozessmanagement, Basis-Treiber die die BIOS-Funktionen nutzen und die virtuellen Kommunikationskanäle implementiert.

Die Hardware wird dabei von der privilegierten¹⁷ Domain-0, das ist der erste virtuelle Server der automatisch gestartet wird, verwaltet¹⁸. Die Domain-0 kommuniziert dabei über spezielle Kanäle mit dem Hypervisor, wie die anderen virtuellen Server auch.

Damit die verschiedenen virtuellen Server nur auf den zugewiesenen RAM zugreifen können, werden die Betriebssystemkernel bei der Paravirtualisierung in dem Ring-1¹⁹ betrieben. Bei der vollständigen Systemvirtualisierung werden die Speicherbereiche für die virtuellen Server in den „Non-Root-Mode“^[16] versetzt, und damit die Rechte ausreichen beschränkt, dass für den virtuellen Server alle 4 CPU-Ringe zur Verfügung stehen.

4.2 Optionen

Xen bietet neben einem recht geringen Overhead (siehe [7, 14]) eine Reihe von Features, die die Verwaltung von virtuellen Servern vereinfachen:

- virtuelle Server können ohne Downtime²⁰ auf andere Server migriert werden, sofern einige Bedingungen erfüllt sind
 - der Speicher auf dem das Image²¹ des virtuellen Servers liegt muss auf beiden beteiligten Servern zugreifbar sein
 - der Zielservers muss die gleichen CPU-Befehle oder eine Obermenge davon unterstützen
 - die Netzwerkverbindung muss „umgeleitet“ werden können. (Entweder gleiches L2-Netzwerk oder die Möglichkeit passende Routen zu setzen)
 - dem virtuellen Server darf keine Hardware zugewiesen sein.
- virtuelle Server können zu Wartungs-²² oder Debugzwecke angehalten werden. Es ist möglich dabei den Speicherinhalt in eine Datei zu schreiben.

¹⁷die anderen virtuellen Server haben normalerweise keinerlei direkte Zugriffsmöglichkeit auf die angeschlossene Hardware

¹⁸Einzelne Geräte können auch anderen virtuellen Servern zugewiesen werden, wenn eine Reihe von Bedingungen erfüllt ist.

¹⁹Die X86-Architektur bietet 4 verschiedene Sicherheitsringe mit abnehmenden Privilegien an. Dabei hat Ring-0 hat sämtliche Rechte, auch für sog. sensitive Befehle, und die Ringe 1-3 hierarchisch abfallende Rechte. D.h. ein Prozess in Ring-1 kann auch auf Speicherbereiche von Ring-3 Zugreifen, aber nicht umgekehrt.

²⁰für einen kurzen Zeitraum, normalerweise deutlich kürzer als 1 Sekunde, wird der virtuelle Server angehalten um die letzten unsauberen Speicherseiten zu kopieren

²¹der persistente Speicher für den virtuellen Server welcher u.A. das Betriebssystem enthält

²²z.B. Erstellung eines Backups/Snapshots

- Es gibt einen sehr flexiblen CPU-Schedulers ist möglich sowohl Gewichtungen bei der CPU-Nutzung anzugeben, als auch feste Verteilungen vorzugeben. Dies kann auch gemischt werden.

4.3 libvirt

Libvirt[6] ist eine Bibliothek, welche für verschiedene Virtualisierungslösungen²³ eine (möglichst) einheitliche und stabile API bietet, und mit C,C++ und Python direkt genutzt werden kann.

Das API bietet Funktionen die für die Verwaltung von virtuellen Servern benötigt werden. Dazu gehört unter Anderen das Anlegen der Konfiguration, starten & stoppen von virtuellen Servern, Informationen über den Zustand des Hostsystems und der virtuellen Server zu liefern.

Die libvirt nutzt bei Xen direkte Schnittstellen zu den jeweiligen Daemon, so dass die Performance im Vergleich zu den Kommandozeilenwerkzeugen deutlich besser ist. Da die Daten in nativen Datentypen geliefert werden, ist keine Konvertierungen notwendig, was die Programmierung erleichtert.

5 Storage Area Network (SAN)

An das, in diesen Projekt als hochverfügbar angesehene, Storage Area Network wurden die folgenden Anforderungen gestellt:

- Es sollte auf alles Server gleich benannte Gerätedateien erzeugen können, damit es für die Konfiguration des virtuellen Servers egal ist, auf welchen er gestartet wird.
- Es sollte als Backend Dateien nutzen können. (Aufgrund der Möglichkeiten im Testlabor)
- Es muss mit einem Xen-Kernel kompatibel sein.
- Der Aufwand für die Einrichtung sollte sich im Rahmen halten.

Diese Anforderungen sorgten dafür, dass verschiedene Techniken getestet wurden.

5.1 iSCSI

iSCSI wird von den verschiedenen SAN-Protokollen in der Dokumentation (z.B. [4, 8]) und den Maillinglisten am häufigsten erwähnt. Aus dem Grund wurde es auch zunächst als Protokoll ausgewählt.

iSCSI basiert auf TCP und überträgt SCSI-Daten und stellt damit virtuelle Punkt zu Punkt Verbindungen her. iSCSI ist im RFC 3720 standardisiert worden.

Bei den Tests zeigte sich aber, dass die Einrichtung sich deutlich komplizierter erwies als geplant, was zum einen daran lag, dass auf den zur Verfügung stehenden Systemen

²³aktuell: Xen, QEmu und (Linux) KVM

das Userland und der Kernel dafür selbst kompiliert werden mussten, und auch die Dokumentation teilweise nicht besonders ausführlich war. Deshalb wurde von dem geplanten Einsatz abgesehen.

5.2 ATA over Ethernet (AoE)

ATA over Ethernet setzt im Gegensatz zu iSCSI direkt auf Ethernet auf, und ist somit nur im lokalen Netz nutzbar. ATA over Ethernet wurde von der Firma „The Brantley Coile Company“ entwickelt, und nutzt statt SCSI-Befehle ATA-Befehle.

Da im Testlabor das Userland und die Kernelmodule vorhanden/über die Paketverwaltung installierbar waren, wurde das als Alternative ausgewählt.

Die Einrichtung war sehr einfach, aber es traten beim Einsatz als Xen-Backend reproduzierbare Fehler auf, die den ganzen Server zum Stillstand brachten.

Da sich das Problem leider nicht ohne weiteres²⁴ beheben lies, musste auch von dieser Lösung Abstand genommen werden.

5.3 Network Block Device (NBD)

Das Network Block Device ist eine linuxspezifische Möglichkeit ein Blockdevice über TCP zu ex- und importieren.

Als Default werden nur 16 Geräte unterstützt, kann aber im Quelltext auf bis zu 128 Geräte erhöht werden. Die einzelnen Kommunikationsbeziehungen müssen explizit angelegt werden, bei einem Verbindungsabbruch muss die Verbindung manuell neu angelegt werden.

Es stellt also eine funktionierende Lösung da, ist aber im Vergleich zu AoE unkomfortable. Diese Lösung wurde im folgenden genutzt.

6 Reliable Server Pooling (RSerPool)

Mit Reliable Server Pooling wird vom IETF[3] eine Architektur spezifiziert, welche logische Sitzungen eines Clients mit einem Pool von Servern verwaltet. Dabei stehen Mechanismen zur Lastverteilung auf die im Pool befindlichen Server bereit, als auch zur Fehlererkennung und für die Unterstützung eines Failovers auf ein Funktionierendes Pool Element.

Reliable Server Pooling nutzt SCTP als Transportprotokoll, um auf Netzwerkebene eine hohe Sicherheit & Verfügbarkeit erzielen zu können.

6.1 Terminologie & Funktionsverteilung

(Server) Pool: Eine Gruppe von Servern die den gleichen Dienst anbieten.

Pool Handle (PH): Das eindeutige Identifikationsmerkmal eines Server Pools, dient zur Unterscheidung von Server Pools.

²⁴verschiedene Kernelversionen und Variationen zwischen Kernelmodul oder fest einkompiliert wurden getestet

Pool Element (PE): Ein einzelner Server in einem Pool. Besitzt eine im Pool eindeutige ID, die Pool Element ID.

Pool User (PU): Ein Client der den Dienst nutzt.

Pool Policy: Regel die vorgibt, nach welchen Kriterien ein neuer Nutzer auf die Pool Elements verteilt werden soll. Mögliche Pool Polycys sind u.A. Round Robin (abwechseln), Random (zufällige Verteilung), Least Used (der die geringste Last hat).

Pool Registrar (PR): Verwaltet den Pool (Welche Pool Elemets sind verfügbar, u.U. die dort herrschende Last) und löst für anfragenden Pool-Usern das Pool Handle in Adressen der Pool Elements auf, unter Beachtung der Pool Policy.

Endpoint Handlespace Redundancy Protocol (ENRP): Protokoll das zwischen Pool Registrars genutzt wird, um auf allen PR eine konsistente Sicht auf den Pool zu haben.[13]

Aggregate Server Access Protocol (ASAP): Das Protokoll welches zwischen den PU und den PR und zwischen den PE und PR genutzt wird.[9] Es regelt die Registrierung von PE, und die Auflösung von PH zu Serveradressen.

6.2 Aufbau

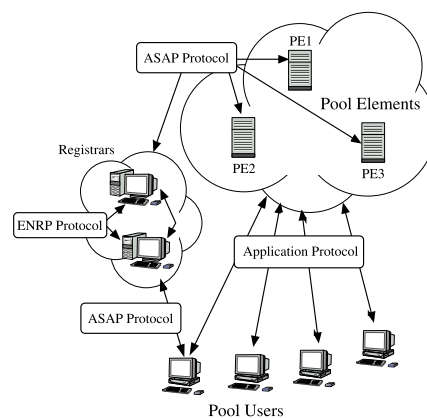


Abbildung 7: Architektur von Reliable Server Pooling nach[13]

Der Aufbau der Reliable Server Pooling Architektur ist in Abbildung 7 abgebildet. Man kann dabei gut die drei logischen Elemente (Pool User, Registrars und Pool Elements) erkennen und deren Kommunikationsbeziehungen.

Da die Reliable Server Pooling Architektur sehr hohe Verfügbarkeiten ermöglichen soll, können, wie in der Abbildung 7 auch zu erkennen, die Serverdienste auf mehrere Geräte verteilt werden. Die Pool Registrars gleichen sich dabei über das ENRP-Protokoll ab, und können gleichzeitig mehrere Pools verwalten. Die Pool Elements melden sich mit dem

ASAP-Protokoll bei einem Registrar an (den so genannten Home-Registrar), der danach periodisch die Erreichbarkeit des Pool Elements überprüft.

Ein Pool User fragt über das ASAP-Protokoll einen beliebigen PR, der dann einen (oder auch mehrere) nach der Pool Policy ausgewählte PE zurück liefert. Zu dem ersten Eintrag wird dann versucht eine Verbindung aufzubauen. Zwischen den PU und dem PE wird dann das Anwendungsspezifische Protokoll gesprochen. Je nach Anwendung muss das Protokoll ergänzt werden, um den in der Reliable Server Pooling Architektur vorgesehene Failover bei einem Ausfall eines PE zu ermöglichen.

6.3 Der Prototyp - rsplib

Der Prototyp rsplib[11] wurde maßgeblich von Thomas Dreibholz entwickelt. Als Lizenz wurde die GPL gewählt.

Die rsplib implementiert ein Framework auf Basis der RSerPool-Architektur, welches dem Programmierer die RSerPool spezifischen Arbeiten weitgehend abnimmt²⁵, so dass die Entwicklung von Applikationen die RSerPool nutzen stark vereinfacht wird.

Teil III

Implementierung

7 Aufbau des Prototypen

Im folgenden wird die Struktur des realisierten Prototypenvorgestellt und einige Annahmen und Designentscheidungen die die Implementierung beeinflusst haben, erläutert.

7.1 Architektur

Es handelt sich um eine Client-Server Architektur welche das rsplib-Framework[11] nutzt.

Die Clients haben nur die Funktion die Anforderung, bestehend aus den Parametern zum Starten des virtuellen Servers, an den Pool zu übermitteln, und später periodisch Nachrichten zur Kontrolle des Zustandes zu versenden.

Auf der Serverseite werden die Nachrichten bearbeitet, die virtuellen Server verwaltet, das Failover realisiert und die Auslastung²⁶ für die Lastverteilung von RSerPool ermittelt.

7.2 Annahmen / Festlegungen

Für dieses Projekt wurden u.A. folgende Festlegungen getroffen:

²⁵die benötigten Parameter & Eventhandler muss der Anwendungsentwickler definieren

²⁶die Ermittlung der Auslastung wird später detaillierter besprochen, da dies relativ komplex ist.

- Der persistente Speicher²⁷ der virtuellen Server wird in diesen Projekt nicht verwaltet. Dieser ist auf allen im Pool befindlichen Servern gleich nutzbar, und ist per Definition verfügbar.
- Die Clients starten nicht mehrere virtuelle Server mit dem gleichen persistenten Speicher, wenn das Dateisystem nicht dafür ausgelegt ist.
- Es wird nur beim Start eines virtuellen Servers geschaut welcher Server zu dem Zeitpunkt die geringste Last hat. Spätere Änderungen der Lastverteilung werden ignoriert.

7.3 Inhalte der Nachrichten

In der Datei XenServpackets.h (siehe Anhang B.1) sind die verwendeten Pakete definiert, der Inhalt wird im folgenden erläutert.

ServerID Ein bis zu 32 Zeichen langer im Pool eindeutiger²⁸ Bezeichner für den virtuellen Server.

blockdevice Ein maximal 256 Zeichen langer String, der den Pfad zu dem persistenten Speicher des virtuellen Servers auf den Pool Elements definiert. Dies ist möglich, da in diesen Projekt der gleiche Zugriff bei allen Pool Elements verlangt wurde.

IP_List Ein maximal 256 Zeichen langer String, der die für den virtuellen Server zu nutzenden IP-Adressen angibt. Mehrere IP-Adressen werden jeweils durch ein Leerzeichen getrennt.

requested_ram Ein Integer Wert, der die gewünschte RAM-Zuweisung für den virtuellen Server in Megabyte enthält.

ErrorCode & responseCode Bieten die Möglichkeit den Client codiert zusätzliche Informationen zu der angeforderten Aktion zurückzuliefern. Diese Information ist nicht für die Funktion notwendig, aber unter Umständen für den Anwender, der den Pool User nutzt, hilfreich.

available_ram Ein Integer Wert, der den freien RAM des Pool Elements in Kilobyte enthält. Wird in der Implementierung nicht genutzt, bietet dem Pool User theoretisch die Möglichkeit den Bedarf den Möglichkeiten im Fehlerfalle anzupassen.

intervall Ein Integer Wert für den Zeitraum zwischen den periodischen Anfragen des Clients in Sekunden. In der aktuellen Implementierung ungenutzt.

running Ein Boolean Wert der angibt, ob der virtuelle Server laufen kann. (False bis vor dem Startversuch, oder nach dem herunterfahren)

²⁷das Festplattenimage

²⁸Die Eindeutigkeit wird durch die Implementierung nicht sichergestellt, aber es wird sich darauf verlassen, dass sich dieser vom Nutzer definierte String so verhält.

Stonith Ein maximal 128 Zeichen langer String, welcher eine eindeutige Bezeichnung des Pool Elements erlauben soll.²⁹ Dient dazu das Projekt um eine Möglichkeit zu erweitern im Failoverfall den Server auf dem der virtuelle Server zuvor gelaufen hat über externe Mechanismen sicher abzuschalten. Damit kann ein Betrieb auf zwei Server gleichzeitig vermieden werden. Ohne diesen Mechanismus könnte dies sonst bei Netzwerkstörungen passieren, und damit unter Umständen die Datenintegrität auf dem persistenten Speicher gefährden.

7.4 Lastdefinition

Eines der größten Probleme bei der Verwaltung von virtuellen Servern mit Reliable Server Pooling ist eine geeignete Lastdefinition.

Das Reliable Server Pooling Framework nutzt zur Lastverteilung bei den dynamischen Pool Policies³⁰ nur eine eindimensionale abstrakte Lastdefinition.

Um virtuelle Server zu charakterisieren verwendet man eher mehrdimensionale Lastdefinitionen. Mögliche Parameter sind: CPU-Nutzung, benötigte RAM-Menge, benötigter Speicherplatz, Netzwerkbandbreite, Disk-IO-Durchsatz.

Da die Parameter weitgehend unabhängig voneinander sind³¹, ist es schwer diese so auf eine eindimensionale Last abzubilden, dass eine möglichst optimale Verteilung erzielt wird.

Bei diesen Projekt wurden die beiden Parameter CPU-Nutzung und RAM-Bedarf als relevante Parameter definiert, und schon mit den zwei Parametern war es, wie im Teil IV beschrieben, schwer eine gute Verteilung zu erzielen.

Die folgenden drei verschiedenen Lastdefinitionen wurden getestet (und implementiert):

- Das Maximum aus CPU-Auslastung und belegten RAM (beides in Prozent).
- Nur die CPU-Auslastung.
- Ein gewichtetes Mittel aus CPU-Auslastung und RAM-Auslastung.

Die Pool Policy soll die virtuellen Server möglichst gut verteilen. „Gut“ heißt in diesen Kontext, dass die Last³² auf den Pool Elements möglichst gleichmäßig verteilt wird, und dass alle virtuellen Server möglichst direkt einen passenden Pool Element zugeordnet werden.

8 Implementierung

Die Struktur der Implementierung für die Verwaltung der virtuellen Server auf Xen-Basis wurde stark durch das rsplib[11] Framework geprägt.

²⁹In der aktuellen Implementierung der Hostname

³⁰wie die verwendete Least Used

³¹zumindest wenn man virtuelle Server im Allgemeinen betrachtet, in bestimmten Anwendungsszenarien kann es da durchaus Abhängigkeiten geben

³²vor allen die CPU-last, wenn ein Teil des RAMs des Servers nicht genutzt wird hat das keine positiven/negativen Effekte

Der Prototyp besteht aus vier Teilen:

- Den schon in Kapitel 7.3 besprochenen Nachrichtendefinitionen in der Datei Xen-Servpackets.h (siehe Anhang B.1).
- Dem Pool User in der Datei XenServ.cc.
- Der Registrierung des Xen-Verwaltungsdienst in der Datei server.cc
- Die Realisierung Xen-Verwaltungsdienst in den Dateien standardservices.cc und standardservices.h.

Auf die letzten drei wird im Folgenden näher eingegangen.

8.1 Pool User / Client

Der Pool User wurde unter der Nutzung der Enhanced Mode API[10, 12] des rsplib-Frameworks geschrieben. Dadurch müssen die Session-State Cookies nicht verwaltet werden, so dass im Client selber neben der Reliable Server Pooling Initialisierung nur die eigentliche Client-Server Kommunikation beinhaltet.

Der Pool User (siehe Anhang B.2) hat einen sehr einfachen Aufbau. In den Zeilen 18 bis 41 werden die die verwendeten Variablen deklariert.

In den Zeilen 45 bis 69 werden die übergebenen Parameter³³ geparkt und die Variablen damit belegt.

In den Zeilen 81 bis 96 wird das rsplib-Framework initialisiert, und der Socket geöffnet. In den Zeilen 98 bis 115 wird dann die Nachricht vom Typ `CreateServer` erstellt und versandt, welche die vorher geparkten Informationen für den zu erstellenden virtuellen Server enthält.

Bis zur Zeile 175 folgt dann die Schleife, welche bis zum Beenden der Verbindung zwischen dem Pool User und dem Pool³⁴ durchlaufen wird.

Dabei wird zuerst in den Zeilen 119 bis 125 versucht eine Nachricht zu empfangen, der Timeout beträgt dabei die Zeit bis zum Versand der nächsten periodischen „Heartbeat“-Nachricht.

In den Zeilen 127 bis 156 wird dann im Falle eines Nachrichtenerhalts je nach Nachrichtentyp für den Benutzer Informationen ausgegeben. Falls es sich um eine bestimmte RSerPool Nachricht handelt, wird außerdem (in Zeile 128) der durch das Framework implementierte Failover angestoßen.

Am Ende der Schleife (Zeile 158 - 173) wird, falls das Intervall zwischen zwei Heartbeat-Nachrichten verstrichen ist, die Heartbeat-Nachricht erstellt und versandt.

Der Code ab Zeile 179 wird nur beim Beenden ausgeführt, und schließt den Reliable Server Pooling Socket, und beendet das Framework.

³³Möglich sind die Parameter für den virtuellen Server und welche für das rsplib-Framework

³⁴also so lange bis der der Client beendet wird

8.2 Pool Element

Das rsplib-Framework ermöglicht die Aufteilung in zwei Teile.

Der erste Teil, die Registrierung in `server.cc`, beinhaltet nur die Registrierung des Dienstes in dem Serverprogramm. Dieses ruft dann den eigentlichen Dienst auf.

Der zweite Teil, in `standardservices.h` und `standardservices.cc`, beinhaltet die eigentliche Diensterbringung. Dazu wird eine Klasse von der Klasse `TCPLikeServer` abgeleitet.

Die Klasse `TCPLikeServer` ist im rsplib-Framework enthalten und implementiert einen generischen Dienst. Dieser Dienst startet für jede neue Verbindung einen eigenen Thread, und implementiert die notwendigen Verwaltungsnachrichten für Reliable Server Pooling, wie die Registrierung bei den Pool Registrars.

Die Nutzung der `libvirt` war zunächst nicht geplant. Da aber nach den ersten Versuchen deutlich wurde, dass das regelmäßige Aufrufen der Kommandozeilen-Programme einen relativ hohen Overhead erzeugte (siehe Kapitel 10.1 auf Seite 24), wurde die Nutzung der `libvirt` für die periodischen Aufgaben geplant. Da die `libvirt` auch andere Daten³⁵ zuverlässig & schnell, das heißt ohne das Parsen der Ausgabe von Kommandozeilenprogrammen, bereitstellen konnte, wurden auch die Methoden umgeschrieben.

8.2.1 Registrierung in `server.cc`

Bei der Registrierung in der Datei `server.cc` (siehe B.3.1) sind die folgenden Stellen zu beachten.

Zuerst wird in Zeile 12 für den Dienst eine eindeutige ID definiert.

In den Zeilen 158 bis 160 wird dann die Kommandozeile geparkt, und die interne Variable `service` auf den Xen-Dienst gesetzt.

In den Zeilen 182 bis 206 wird dann, falls der Xen-Service angeboten werden soll, noch einige Variablen vorbelegt, und dann ab Zeile 194 der Dienst gestartet.

8.2.2 Diensterbringung in `standardservices.cc` und `standardservices.h`

Das rsplib-Framework erlaubt es einen einfachen Dienst der vom Typ `TCPLikeServer` durch das Schreiben weniger Methoden in der abgeleiteten Klasse. Notwendig sind neben den Constructor und Destructor, Methoden zum starten und stoppen eines Threads, die Methoden zur Behandlung von Nutzdaten vom Pool User³⁶ und zur Behandlung von Nachrichten vom Typ Cookie-Echo³⁷. Diese Methode ist für den Failover zuständig wenn vom Pool User das Session-State Cookie gesendet wird.

Im Folgenden werden die Methoden in der Datei `standardservices.cc` erläutert, bei der ersten Verwendung dann, soweit notwendig, auch die in `standardservices.h` definierten Datenstrukturen.

³⁵Anzahl der CPUs, Menge an installierten RAM

³⁶Die Methode `handleMessage` siehe B.3.3 ab Zeile 224

³⁷Die Methode `handleCookieEcho` siehe B.3.3 ab Zeile 224

XenServer(...) - **Constructor** Im Konstruktor wird für die Verwaltung des virtuellen Servers der Status der Variablen `vserverSettings.domainRunning`, auf falsch gesetzt, damit eindeutig definiert ist, dass der von diesem Thread verwaltete virtuelle Server noch nicht gestartet wurde.

Des Weiteren wird der Variable `domainInfo` ein neues Objekt vom Typ `virDomainInfo` zugewiesen. Dieser Datentyp wird von der `libvirt` genutzt um Daten, wie die RAM-Zuweisung und verbrauchte CPU-Zeit eines virtuellen Servers zu übergeben.

~XenServer() - **Destructor** Im Destructor finden keine Aktionen statt.

XenServerFactory(...) In dieser Methode wird ein neues Objekt für die Erstellung eines neuen Threads an das `rsplib`-Framework geliefert.

XenServerinitializeService(...) Diese Methode wird vom `rsplib`-Framework beim Start des Pool Elements aufgerufen. Dabei werden Initialisierungen und Variablenbelegungen vorgenommen, die nur einmal pro Pool Element getätigt werden müssen, und nicht für jeden Thread erneut.

In Zeile 413 wird die `libvirt` initialisiert. In der Zeile 417 wird dann versucht eine Verbindung zum Xen-Hypervisor aufzubauen.

Falls das gelingt wird in Zeile 419 über die `libvirt`-Bibliothek versucht Informationen³⁸ über das System auf dem das Pool Element läuft zu bekommen.

In der Zeile 420 wird dann noch die `Stoneith`-Information gesetzt. Diese ist in dieser Implementierung der Hostname, den die Methode `getHostname()` liefert.

In der Zeile 421 wird der für die Verwaltung mittels `libvirt` notwendige Pointer für der Domain-0 (siehe Kapitel 4.1 auf Seite 7) über die Methode `virDomainLookupByID` versucht zu erstellen. Der Lookup über die ID ist möglich, da die Domain-0 immer die ID 0 hat, daher auch der Name `Domain-0`.

In den Zeilen 423 bis 425 wird dann die Variable `dom0Settings.info` ein neu erstelltes Objekt vom Typ `virDomainInfo` zugewiesen, und dieses danach aktualisiert. Danach wird die bisher verbrauchte CPU-Zeit der Domain-0 in der Variablen `dom0Settings.CPU_Seconds` gespeichert.

In der Zeile 426 wird dann noch die Variablen `freeRAM` mit dem aktuell frei zur Verfügung stehenden RAM durch einen Aufruf der Methode `getFreeRAM()` belegt.

XenServerfinishService(...) Diese Methode wird vom `rsplib` Framework beim Beenden des Pool Elements aufgerufen.

Es wird die Verbindung der `libvirt` geschlossen.

finishSession(...) Die Methode `finishSession()` wird aufgerufen, wenn ein Thread beendet wird, also eine Verbindung zu einem Pool User endet³⁹.

³⁸Anzahl der Prozessoren / Prozessorkerne, vorhandener RAM, weitere wie die Taktfrequenz vorhanden.

³⁹unabhängig ob es ein Verbindungsabbruch oder ein vom Nutzer gewolltes Ende ist.

Für den Fall, dass der verwaltete virtuelle Server laufen könnte, dementsprechend die Variable `vserverSettings.domainRunning` den Wert wahr hat, wird der Server mit den Kommandozeilentools heruntergefahren.

Dazu wird zuerst in Zeile 70 dem virtuellen Server über ein Signal die Möglichkeit gegeben sich von alleine zu beenden. Falls dies nicht, innerhalb eines Timeouts⁴⁰, erfolgreich ist, wird in Zeile 76 der virtuelle Server durch den Entzug sämtlicher Betriebsmittel zwangsweise beendet.

Zum Schluss wird der freie RAM aktualisiert, indem der von dem nun beendeten virtuelle Server vorher belegte RAM bei der Variablen `freeRAM` hinzu addiert wird.

getHostname(void) Die Methode liefert den Hostname des aktuellen Systems zurück. Dazu wird die Ausgabe von dem Befehl `hostname` verwendet⁴¹.

getFreeRAM(void) Diese Methode hat als Rückgabewert die Menge des aktuell für virtuelle Server verfügbaren RAM in Kilobyte.

Um den den verfügbaren RAM zu berechnen wird zuerst in Zeile 442 der gesamte im System vorhandene RAM in eine Hilfsvariable (hier `ram`) kopiert, und danach der RAM-Bedarf⁴² des Hypervisors in Zeile 445 abgezogen.

In Zeile 446 wird dann die Zahl der laufenden virtuellen Server ermittelt, und die lokalen IDs in ein Array kopiert. In der Schleife ab Zeile 450 wird dann für jeden im Array stehenden virtuellen Server der belegte RAM ermittelt und von dem verfügbaren RAM abgezogen.

Zum Schluss wird noch der RAM für die Domain-0 in der Zeile 455 abgezogen, bevor der Wert zurückgegeben wird, und die Methode beendet ist.

loadUpdateHook(const double load) Diese Methode wird von dem `rsplib`-Framework aufgerufen, und kann die Last ändern die an die Pool Registrars gesendet wird. Als Parameter bekommt die Methode die Summe der Last der laufenden virtuellen Server. Die Last wird zwischen 0 und 1 abgebildet, wobei 1 eine 100% Auslastung des Systems abbilden soll.

Zunächst wird ab Zeile 355, falls seit dem letzten Update mehr Zeit (in Sekunden) verstrichen ist, als in der Konstanten `loadUpdateIntervall` vorgesehen ist, die CPU-Last der Domain-0 ermittelt.

Die CPU-Last ist definiert als Anteil der gesamten CPU-Zeit seit dem letzten Update. Um kurzfristige Peaks zu glätten, wird in Zeile 370 noch ein gleitendes Mittel errechnet, und danach die Menge des freien RAM aktualisiert.

In der Zeile 379 wird die Variable `temp_Load_raise` auf falsch gesetzt. Diese Variable wird genutzt um kurzfristig⁴³ die Last auf 1 anzuheben. Falls ein virtueller Server aufgrund zu wenig RAM nicht starten kann, darf der Server nicht wieder wegen seiner geringen Last

⁴⁰Wird in der Xen-Konfiguration definiert

⁴¹Falls, wieder erwarten mehrere Zeilen, ausgegeben werden, wird die letzte Zeile als Hostname angenommen.

⁴²Bei der verwendeten Xen-Version 12MB für den Hypervisor und 7 KB pro MB (siehe [14])

⁴³bis zum nächsten Update der Domain-0 Last

ausgewählt werden. Durch die Erhöhung der Last auf den Wert 1 gelangt der Server ans Ende der von dem Pool Registrar gelieferten Liste, und ein anderer Server wird probiert.

Der Code ab Zeile 383 wird bei jedem Durchlauf ausgeführt. Zunächst wird kontrolliert ob `temp_Load_raise` auf den Wert wahr gesetzt wurde, falls dem so ist wird sofort die Last 1 zurückgeliefert.

In Zeile 392 wird dann die CPU-Last der Domain-0 und die CPU-Last der anderen virtuellen Server in der Variablen `cpuload` zusammengeführt.

In der Zeile 397 wird die RAM-Auslastung ermittelt.

Danach wird je nach gewählten Algorithmus aus den beiden Werten die Last berechnet und zurückgegeben.

createDomain(void) Diese Methode versucht einen virtuellen Server zu starten. Die Parameter sind in den Variablen des Objektes enthalten.

Zuerst wird in Zeile 89 die Variable `vserverSettings.domainRunning` auf wahr gesetzt, da im folgenden versucht wird den virtuelle Server zu startet wird, und es deshalb möglich ist, dass der virtuelle Server ab diesen Zeitpunkt läuft. Diese Information ist für das Stonith beim Failover wichtig.

Damit die geänderte Information auch beim Pool User gespeichert wird, wird ein Session State Cookie versendet (Zeile 92 bis 103), welche die Daten für den Failoverfall enthält.

In den Zeilen 106 und 107 wird die Startzeit vermerkt, und die bislang verbrauchte CPU-Zeit mit 0 initialisiert, damit die Lastberechnung sinnvolle Ausgangsdaten hat.

In den Zeilen 109 bis 128 wird der Befehl zum Starten eines virtuellen Servers mit dem Kommandozeilentool `xm` erstellt und ausgeführt. Hier wurde nicht die von libvirt angebotene Funktion genutzt, da für die notwendige XML-Struktur die Dokumentation nicht ausreichend war, und bei der seltenen Ausführung der Performanceunterschied als nicht besonders relevant angesehen wurde.

In der Zeile 129 wird dann der Zeiger für die libvirt ermittelt. Dieser wird für den direkten Zugriff auf die Verwaltungsstrukturen benötigt.

Falls dies erfolgreich war, dann wird in den Zeilen 133 bis 143 eine Nachricht vom Typ `XSPT_create_success` an den Client gesandt. Diese informiert den Client, dass der virtuelle Server nun gestartet ist.

In der Zeile 147 wird die Last des gerade gestarteten virtuellen Servers auf den Schätzwert⁴⁴ der RAM-Belegung gesetzt. Dies ist notwendig, damit die Last des Pool Elements schnell angepasst wird, und nicht mehrere zeitnah gestartete virtuelle Server auf dem gleichen Pool Element landen, auch wenn nach dem ersten andere geeigneter wären.

In der Zeile 149 wird dann noch der verfügbare RAM um die gerade dem virtuellen Server zugewiesenen Menge reduziert, und dann der Status `EHR_Okay` zurückgeliefert.

Falls der Zeiger für die libvirt nicht ermittelt werden konnte⁴⁵, dann wird dem Pool User das über eine Nachricht vom Typ `XSPT_create_failed` (Zeile 155-164) mitgeteilt,

⁴⁴da der CPU-Bedarf noch nicht bekannt ist, wurde das als Näherung verwendet, andere Annahmen wie eine konstante Last wären auch denkbar

⁴⁵häufigster Grund: der virtuelle Server konnte aufgrund eines Fehlers nicht gestartet werden

und der Rückgabewert ist dann `EHR_Abort`.

handleCookieEcho(const char* buffer, size_t bufferSize) Diese Methode wird von dem `rsplib`-Framework aufgerufen, falls ein Session State Cookie empfangen wird, also ein Failover ausgeführt werden muss.

In den Zeilen 179 bis 183 wird eine einfache Plausibilitätsprüfung durchgeführt, ob es sich bei der empfangenen Nachricht um eine Nachricht im passenden Format handelt.

In den Zeilen 186 bis 189 werden dann die Variablen des Objektes belegt.

In den Zeilen 204 bis 212 wird überprüft ob eine ausreichende Menge an RAM frei ist um den virtuellen Server zu starten. Falls nicht wird `EHR_Abort` zurückgeliefert, und falls das Feature für die temporäre Lasterhöhung beim RAM-Mangel aktiviert ist, die Last erhöht.

In Zeile 217 wird überprüft ob der virtuelle Server auf dem Pool Element auf dem bislang die Verbindung endete vermutlich lief. In dem Fall werden die Stoneith-Instruktionen ausgeführt⁴⁶.

In der Zeile 221 wird dann versucht den virtuellen Server zu starten, und das Ergebnis des Aufrufes wird zurückgeliefert.

handleMessage(...) Diese Methode behandelt die Nutznachrichten vom Pool User. Es werden zwei Nachrichtentypen behandelt. Den Nachrichtentyp `XSPT_createServer` in den Zeilen 231 bis 300 welcher eine Anfrage zur Erstellung eines virtuellen Servers darstellt, und ab Zeile 301 bis 343 den `XSPT_CheckRunning`, welcher zur periodischen Kontrolle dient.

Falls keiner der beiden Nachrichtentypen erkannt wird, wird ein `EHR_Abort` zurückgeliefert, welches die Sitzung beendet.

XSPT_createServer Zuerst werden ab Zeile 238 die Parameter aus der Nachricht in die Variablen des Objektes kopiert. Danach wird für den Betreiber die Anfrage als Text ausgegeben.

Ab Zeile 255 wird dann an den Pool User ein Session State Cookie gesendet, dies sorgt dafür, dass falls das gewählte Pool Element die Anfrage nicht bedienen kann, der Failover funktioniert. Bei diesem Session State Cookie ist die Variable `running` noch auf falsch gesetzt, da zu dem Zeitpunkt noch kein Startversuch stattgefunden hat. Deshalb wird bei einem Failover auch kein Stonith versucht.

In der Zeile 270 wird kontrolliert, ob genügend unbelegter RAM für den angeforderten virtuellen Server vorhanden ist. Falls genügend RAM vorhanden ist, wird in Zeile 297 versucht den virtuellen Server zu starten, und das Ergebnis wird zurückgeliefert.

Falls nicht genügend RAM frei ist, dann wird in den Zeilen 272 bis 284 eine entsprechende Meldung zur Information des Pool Users erstellt, falls aktiv die Last temporär erhöht und ein `EHR_Abort` als Rückgabewert geliefert.

⁴⁶In der Implementierung nur eine Ausgabe.

XSPT_CheckRunning In dieser Hälfte ab Zeile 301 werden einige Bedingungen abgefragt, bevor in Zeile 338 die positive Rückmeldung über den laufenden virtuellen Server zurückgegeben wird.

Zunächst wird in Zeile 303 versucht den Status mit der libvirt-Funktion `virDomainGetInfo()` zu aktualisieren. Falls dies erfolgreich ist, wird ab Zeile 305 dem Pool User ein Antwortpaket gesendet.

Falls die Prüfung auf einem normalen Betriebszustand des virtuellen Servers in Zeile 317 erfolgreich ist, dann wird in Zeile 338 das `EHR_Okay` zurückgeliefert.

Falls seit der letzten Lastaktualisierung mehr als in der Konstanten `loadUpdateIntervall` definierten Sekunden verstrichen sind, wird zuvor in den Zeilen 321 bis 337 die CPU-Last des virtuellen Servers berechnet. Dabei wird in Zeile 328 der Anteil an der in dem Intervall theoretisch nutzbaren CPU-Zeit berechnet, und in Zeile 333 ein gleitendes Mittel bestimmt, wobei die Konstante `loadUpdateFaktor` festlegt wie stark die vergangenen Werte Einfluss haben.

Teil IV

Tests

9 Versuchsaufbau

Es gab zwei Versuchsreihen, deren Aufbau von der verwendeten Infrastruktur identisch waren. Die genutzten virtuellen Server und die Testmethoden unterschieden sich.

Die erste Versuchsreihe beinhaltete Tests zur Lastverteilung, während die zweite Versuchsreihe den Failoverfall untersuchte.

Im folgenden wird zunächst die gemeinsam genutzte Infrastruktur vorgestellt, und danach die für die jeweilige Testreihe spezifischen Anpassungen.

9.1 verwendete Hardware

Für die Versuche wurden 10 identisch ausgestattete PCs verwendet, welche über einen DualCore Prozessor und 1024 MB RAM verfügten.

Alle PCs waren per Ethernet miteinander verbunden.

9.2 genutzte Software

- Auf den PCs Ubuntu 6.06
- nbd als „SAN“ (siehe auch Kapitel 5.3)
- Xen 3.0.3 mit selbstkompilierten Kernel für die Domain-0
- rsplib 2.1 - Das Reliable Server Pooling Framework

- sctplib 1.0.6 eine Userspace SCTP-Implementierung
- socketapi 1.9.2
- libvirt 0.2.2
- stress als Lastgenerator und apache2 als Dienst.
- heartbeat-2

9.3 Konfiguration

Die 10 zur Verfügung stehenden PCs wurden wie folgt aufgeteilt:

- 1 PC als „SAN“-Server und Pool Registrar. Auf den PC wurden die Diskimages als Dateien angelegt, und vor den Testläufen mit dem nbd-server für die Clients verfügbar gemacht.
Die Diskimages wurden mit einem minimalen Debian-System versehen, und den jeweiligen für den Test notwendigen Programmen versehen.
- 1 PC zum Starten der Pool User und als Pool Registrar.
- 8 PC bildeten die Pool Elements, und liefen mit dem Xen-Hypervisor.

Auf den Pool Elements wurde vor den Testläufen die Blockdevices für NBD (siehe 5.3) angelegt und konfiguriert.

Es wurde ein Template für einen virtuellen Server unter `/etc/xen/` erstellt, welches die für alle virtuellen Server konstanten Parameter enthielt⁴⁷.

Der Netzwerktyp für Xen ist gerouted.

9.3.1 Lastverteilung

Die virtuellen Server, die gestartet werden, erzeugen eine fest definierte Last und belegen eine bestimmte Menge RAM.

Die Last wird dabei durch den Aufruf des Programms `stress` erzeugt, welches durch mathematische Berechnungen die CPU eine definierte Zeit lang versucht zu belegen. Um die bei echten Systemen auftretenden Schwankungen zu modellieren, wird die Laufzeit in $\frac{2}{3}$ der Schleifendurchläufe zufällig um 1 Sekunde verlängert oder verkürzt (siehe Anhang A.4).

Für die Test zur Lastverteilung wurden 4 Typen von virtuellen Servern definiert, die verschiedene Lastprofile abbilden sollen.

„**Last-2**“ nutzt 64 MB RAM und 2 Sekunden von 30 Sekunden wird Last für einen CPU-Kern erzeugt.

„**Last-5**“ nutzt 300 MB RAM und 5 Sekunden von 30 Sekunden wird Last für einen CPU-Kern erzeugt.

⁴⁷In dieser Implementierung: Kernel, Ramdisk, root-Device

„**Last-10**“ nutzt 150 MB RAM und 10 Sekunden von 30 Sekunden wird Last für einen CPU-Kern erzeugt.

„**Last-25**“ nutzt 64 MB RAM und 25 Sekunden von 30 Sekunden wird Last für einen CPU-Kern erzeugt.

Durch die Inhomogenität der verschiedenen virtuellen Servern ist die Lastdefinition wie schon in Kapitel 7.4 erwähnt nicht trivial, aber für eine gute Verteilung entscheidend.

Bei den Lastverteilungstest wurden jeweils acht virtuelle Server vom Typ Last-2, Last-5 und Last-25 gestartet. Vom Typ Last-10 wurden jeweils 16 Stück gestartet, so dass bei gleichmäßiger Belegung jeder der acht Pool Elements eine CPU-Auslastung von ca. 87%⁴⁸ erreichen würde.

9.3.2 Failovertests

Einfache qualitativen Failovertests wurden mit vorhandenen virtuellen Servern für die Lastverteilung durchgeführt.

Für die quantitative Messung der Failoverzeiten wurde ein virtueller Server im SAN angelegt, welcher als Dienst einen Apache-Webserver automatisches beim booten startet. Dieser liefert eine einfache Test-Webseite aus.

Um die IP-Adresse für den Test im Netz erreichbar zu machen und nach einem Failover schnell die ARP-Tabellen zu aktualisieren wurde das vif-Route Skript, wie in Anhang A.3 abgebildet, geändert.

9.4 Messung

Die Erhebung der Messdaten unterscheidet sich zwischen den beiden Tests sehr deutlich.

Lastverteilung Bei den Lastverteilungstests wurde, nachdem alle virtuellen Server gestartet wurden, auf allen Pool Elements dreimal im Abstand von jeweils etwa 5 Minuten die laufenden virtuellen Server und die CPU-Zeit die sie verbraucht haben ermittelt. Dies geschah mit dem Kommandozeilenutility `xm`.

Aus diesen Daten liessen sich dann die theoretische CPU-Last und die RAM-Auslastung anhand der Anzahl und des Typs der virtuellen Server berechnen. Anhand der der zwischen den Messungen verbrauchten CPU-Zeit wurde auch die tatsächliche CPU-Last ermittelt.

Failovertest Für den Failovertest wurde von einem Client im Netz eine Schleife ausgeführt, welche zuerst Versucht die Webseite zu laden, mit einen Timeout von einer Sekunde, und danach eine Sekunde wartet. Bei jeden Durchlauf wird ein Zeitstempel in einer Logdatei vermerkt, und bei erfolgreichen Downloads auch das Ergebnis.

⁴⁸Da jedem Pool Element 2 CPU-Kerne zur Verfügung stehen können theoretisch 60 CPU-Sekunden in den 30s eines Zyklus abgearbeitet werden. Da insgesamt $(25 + 2 * 10 + 5 + 2) = 52$ Sekunden angefordert werden ergibt sich $\frac{52}{60} = 0,8\bar{6} = 86,6\%$

Für den Failovertest waren jeweils zwei Pool Elements verfügbar. Nachdem der virtuelle Server auf einem der beiden Pool Elements gestartet wurde, wurde die Überwachung auf dem Client gestartet.

Nun wurde das Pool Element auf dem der virtuelle Server lief ausgeschaltet⁴⁹ um einen Ausfall zu simulieren. Nachdem der virtuelle Server wieder erreichbar war, wurde dafür gesorgt, dass wieder zwei Pool Element verfügbar waren, und der Test wiederholt.

Aus der Logdatei lässt sich dann der Zeitraum ist bis der IP-Stack wieder läuft & konfiguriert ist und der Zeitraum bis der Dienst wieder erreichbar ist ableiten. Dieser Zeitraum beinhaltet die Fehlererkennungszeit⁵⁰, und den Zeitraum zum Starten des virtuellen Servers und des darauf laufenden Dienstes.

10 Auswertung der Lasttests

Die Auswertung der Lasttests wird in drei Teile gegliedert.

Zunächst werde einige allgemeine Beobachtungen & Ergebnisse vorgestellt.

Danach werden die Ergebnisse der Messungen mit den drei unterschiedlichen Lastdefinitionen vorgestellt, und zum Schluss ein Fazit daraus gezogen.

10.1 Beobachtungen

Notwendigkeit der temporären Lasterhöhung Bei allen getesteten Lastdefinitionen kam es bei der Belegung vor, dass ein Server ein relativ geringe Last aufwies, und somit als Ziel gewählt wurde, aber nicht genug RAM für den zu startenden virtuellen Server mehr frei hatte. Dieses Verhalten war besonders beim Start der virtuellen Server mit 300MB Speicherbedarf⁵¹ zu beobachten, aber auch bei anderen virtuellen Servern.

Im günstigen Fall löst sich das Problem dadurch, dass sich durch die wiederholten Startversuche die Last der Domain-0 des Pool Elements erhöht, und ein geeigneter Server dann an die Spitzenposition kommt. Im ungünstigen Fall kann das zweitbeste Pool Element den virtuellen Server auch nicht starten, so dass in dem Zeitraum, der benötigt wird um die Last ausreichend zu erhöhen, bereits die Last des zuerst gewählten Pool Elements wieder soweit gesunken sein kann, um das Ziel für den Failover zu werden.

Wie schon im Kapitel 8.2.2 erwähnt, wurde, um das Problem zu lösen, der Mechanismus eingebaut, die Last temporär auf den Wert 1 zu setzen. Dies sorgt dafür, dass ein Pool Element kurzfristig ganz weit an das Ende der Liste gelangt. Dadurch findet dann pro Pool Element nur noch ein⁵² Startversuch statt.

Dies bewirkt dann, dass zuverlässig⁵³ ein passendes Pool Element gefunden wird.

⁴⁹durch einen Reset, kein normales Herunterfahren

⁵⁰Der Zeitraum den das Reliable Server Pooling Framework mit den gewählten Parametern benötigt einen Ausfall festzustellen.

⁵¹Typ Last-5

⁵²bei zeitlich ungünstigen Reset der Variable, direkt nach dem Setzen, auch evtl. ein zweiter

⁵³vorausgesetzt es sind auf mindestens einem Pool Element genügend Ressourcen vorhanden

Performancevorteil bei der Nutzung von libvirt Die Nutzung der libvirt wurde nach dem zweiten Lauf implementiert.

Lauf	PC72	PC73	PC74	PC75	PC76	PC77	PC78	PC79	Mittel
2	12,31%	15,62%	11,69%	13,27%	14,61%	9,83%	13,5%	12,45%	12,91%
3	6,93%	5,4%	5,66%	5,52%	6,35%	3,54%	6,35%	6,1%	5,73%

Tabelle 2: Last der Domain-0

In der Tabelle 2 kann man die Last der Domain-0 in den Läufen zwei und drei vergleichen.

Die Last, die für die Verwaltung genutzten Domain-0 betrug im zweiten Lauf durchschnittlich fast 13% eines Systems. Es wurde also etwa ein viertel eines Prozessorkernes für die Verwaltung genutzt. Im dritten Lauf betrug die Last durchschnittlich weniger als 6% eines Systems. Es lässt sich also feststellen, dass der Verwaltungsoverhead mit der Nutzung der libvirt um mehr als 50% reduziert wurde.

Abweichungen der gemessenen CPU-Last von der theoretischen CPU-Last Bei den detaillierten Auswertungen der einzelnen Läufe⁵⁴ gibt es häufig eine Differenz zwischen der gemessenen Last und der theoretischen Last. Diese lassen sich in die folgenden drei Kategorien einteilen:

geringe theoretische CPU-Last: bei nur einer geringen theoretischen CPU-Last ist die gemessene CPU-Belastung meist etwas höher. Dies liegt daran, dass in die theoretische CPU-Last nicht die Last der Domain-0, die zur Verwaltung benötigt, wird einfließt.

hohe theoretische CPU-Last: bei einer höheren theoretischen CPU-Last, die aber noch unter 100% liegt, wird teilweise eine etwas zu geringe CPU-Last gemessen. Dies liegt daran, dass die virtuellen Server in diesen Test nur zu bestimmten Zeiten versuchen die CPU zu beanspruchen, und falls dies mehr virtuelle Server als CPU-Kerne vorhanden sind versuchen, kann nicht jeder virtuelle Server die gewünschte CPU-Zeit bekommen.

theoretische Last >1: Bei einer theoretischen CPU-Auslastung größer als 100% ist klar, dass der real gemessene Wert $\leq 100\%$ liegen muss.

Die Abweichungen betragen in den Fällen wo keine Überlastsituation vorlag meist weniger als 10%.

Bedeutung der Startreihenfolge für die Verteilung In den achten und neunten Lauf wurden jeweils alle virtuellen Server eines Typs jeweils hintereinander gestartet (siehe Anhang A.1), so dass jedes Pool Element die gleiche Lasterhöhung hatte.

⁵⁴aufgrund des Umfangs nicht in dieser Ausarbeitung abgedruckt

Dadurch wurde die ideale Lastverteilung erreicht, jedes Pool Element hatte genau die geplante Last von 87%⁵⁵, die sich aus zwei virtuellen Server vom Typ Last-10, jeweils einem vom Typ Last-2, Last-5 und Last-25 besteht.

Bei den anderen Läufen wurden die virtuellen Server in einer anderen Reihenfolge gestartet, achtmal hintereinander Last-10, Last-2, Last-5, Last-25, Last-10, so dass durch die Lastverteilung zwischenzeitlich unterschiedliche Lastsituationen auftraten, die am Ende in keinen der Läufe zur gleichmäßigen Lastverteilung führten. Ein Vergleich der

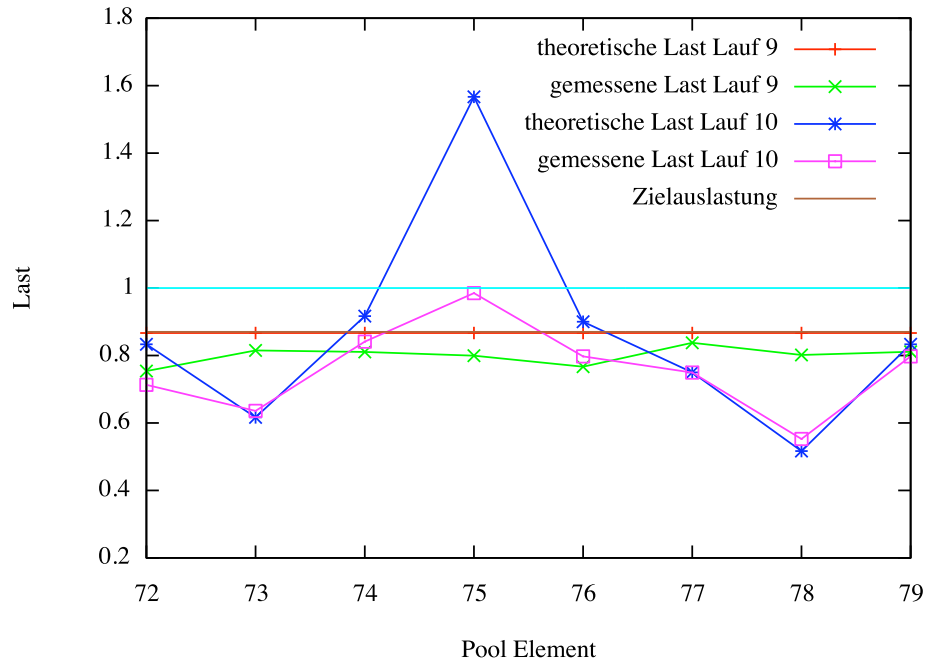


Abbildung 8: Vergleich der Lastverteilung von Lauf 9 und Lauf 10

Verteilung auf die Pool Elements der Läufe 9 und 10 sind in Abbildung 8 zu sehen.

Es trat sogar der Fall auf⁵⁶, dass die Lastverteilung so ungünstig erfolgte, dass auf keinem Pool Element mehr genügend freier RAM vorhanden war um den virtuellen Server Last-5-8⁵⁷ zu starten, obgleich in der Summe noch ausreichen vorhanden gewesen wäre.

10.2 Ergebnisse der Messungen

Maximum aus CPU-Last und RAM-Auslastung als Last Bei den Testläufen fünf bis sieben wurde als Lastdefinition das Maximum aus der CPU-Belastung und der aktuellen RAM-Auslastung genommen.

Die Lastverteilung ist in Abbildung 9 zu sehen.

⁵⁵theoretische Last

⁵⁶im Versuchslauf Nr. 3

⁵⁷der letzte gestartete virtuelle Server der 300 MB RAM anforderte

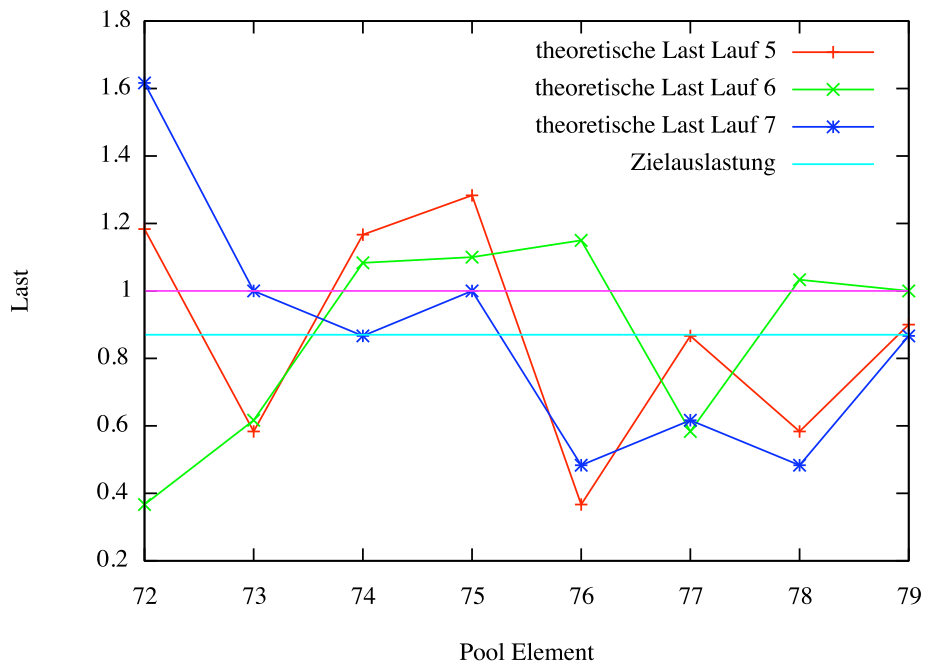


Abbildung 9: theoretische CPU-Last der Pool Elements bei den Läufen 5 bis 7

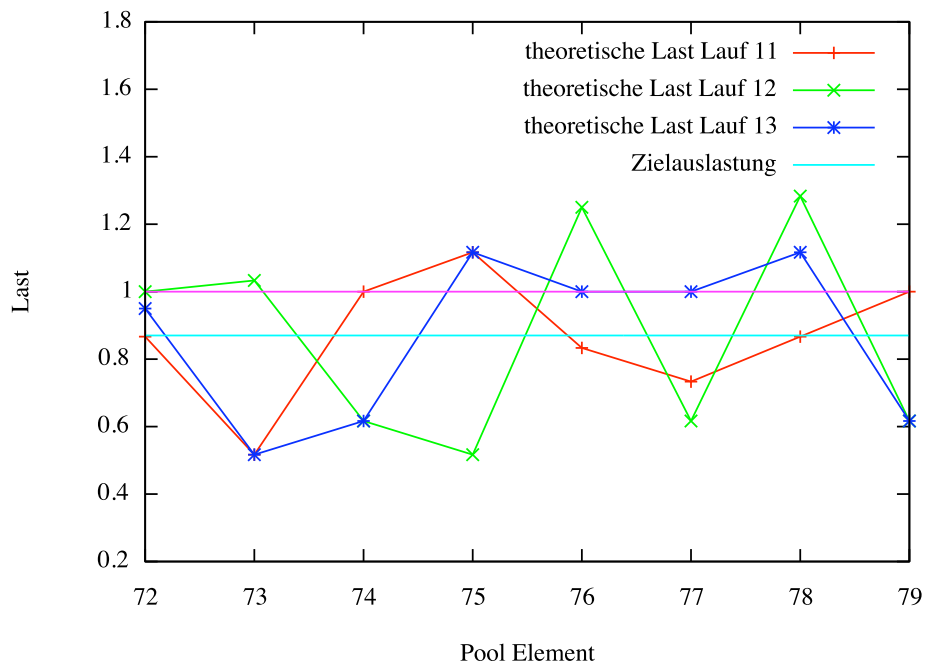


Abbildung 10: theoretische CPU-Last der Pool Elements bei den Läufen 11 bis 13

Gewichtetes Mittel aus CPU- und RAM-Auslastung Bei den Testläufen 11 bis 13 wurde zur Berechnung ein gewichtetes Mittel aus CPU-Last und RAM-Auslastung gewählt, wobei die CPU-Auslastung zu 75% eingeht, und die RAM-Auslastung zu 25%.

Die Lastverteilung ist in Abbildung 10 zu sehen.

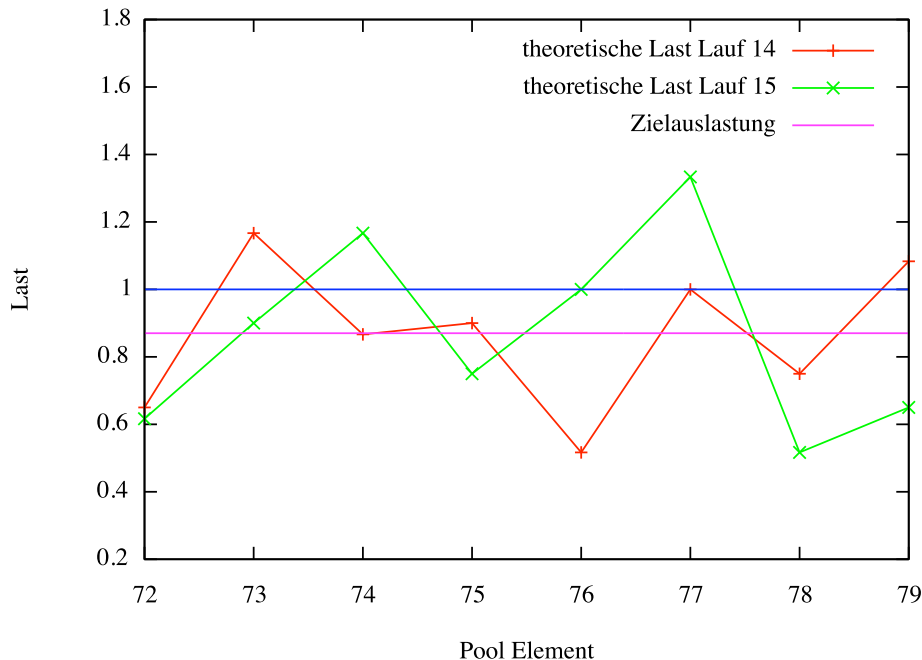


Abbildung 11: theoretische CPU-Last der Pool Elements bei den Läufen 14 bis 15

Nur CPU-Last Bei den Testläufen 14 und 15 wurde als Lastdefinition nur die CPU-Belastung genommen.

Die Lastverteilung ist in Abbildung 11 zu sehen.

Es konnte im Vergleich zu den anderen beiden Lastdefinitionen deutlich mehr Failover beim Start aufgrund von RAM-Mangel beobachtet werden.

Gründe für besonders hohen oder niedrigen Lastwerte von Pool Elements Für die Lastdefinition „Maximum“ und „gewichtetes Mittel“ konnte bei den Pool Elements mit einer sehr hohen theoretischen, und damit auch praktischen, Last der folgende Verlauf in der Belegung festgestellt werden: Nach dem Start des ersten virtuellen Servers vom Typ Last-25 wurde aufgrund der gestiegenen Last meist zwei „Runden“⁵⁸ Zeit keine virtuellen Server gestartet, und im späteren Verlauf wurden neben anderen virtuellen Servern auch ein zweiter⁵⁹ virtueller Server vom Typ Last-25 allokiert.

⁵⁸Schleifendurchläufen im Startskript

⁵⁹in einem Lauf auch ein dritter

Bei den Pool Elements mit sehr geringer Last, lässt sich bei den Lastdefinition „Maximum“ und „gewichtetes Mittel“, kein so deutliches Schema erkennen. Es besteht vor allen aus der Abwesenheit des Typs Last-25 und mindestens einem virtuellen Server vom Typ Last-5, welcher viel RAM bei geringer CPU-Last belegt.

Bei der Lastdefinition nur anhand der CPU-Last können die hohen & niedrigen (theoretische) CPU-Lastwerte über die RAM-Verteilung erklärt werden. Bei den niedrig belasteten Pool Elements wurden mehrere virtuelle Server gestartet, die im Vergleich zur CPU-Last viel RAM belegen, und für virtuelle Server vom Typ Last-25 kein RAM mehr verfügbar war. Diese mussten dann auf Pool Elements gestartet werden, die noch freien RAM hatten, so dass dort die Last erhöht wurde.

10.3 Vergleich der Lastdefinitionen

	1.Lauf	2. Lauf	3. Lauf	Mittel	90% min	90% max	95% min	95% max
Maximum	27%	26%	25%	26,00%	25,33%	26,67%	25,20%	26,80%
gew. Mittel	13%	28%	21%	20,67%	10,59%	30,74%	8,66%	32,67%
CPU	17%	23%		20,00%	15,07%	24,93%	14,12%	25,88%
Max.-Mittel	14%	-2%	4%	5,33%	-5,41%	16,08%	-7,47%	21,01%
Mittel-CPU	-4%	5%		0,50%	-6,90%	7,90%	-8,32%	7,70%
Max.-CPU	10%	3%		6,50%	0,74%	12,26%	-0,36%	13,36%

Tabelle 3: Vergleich der Lastdefinitionen anhand durchschnittlicher Abweichung von der idealen Verteilung

Für den Vergleich der Lastdefinitionen wurde als Qualitätsmaßstab die durchschnittliche Abweichung der berechneten CPU-Last von den Pool Elements von dem Mittelwert⁶⁰ definiert. Diese wird als einfache (absolute) Differenz berechnet und dann über die 8 Pool Elements eines Laufes gemittelt.

Qualitative Betrachtung: Für einen rein qualitativen Vergleich sind die folgenden Punkte festzuhalten:

- Alle drei Lastdefinitionen schafften in den hier verglichenen Läufen alle virtuellen Server zu starten.
- Die Lastdefinition nur über die CPU-Last hatte beim Starten deutlich mehr Failover wegen RAM-Mangel. Falls die virtuellen Server eine sehr kurze Startzeit haben müssen kann das u.U. störend wirken.
- Alle drei Lastdefinitionen liefern bei der getesteten Startreihenfolge ein Ergebnis, welches vom Ideal deutlich abweicht.

⁶⁰der ja auch die Ideale Verteilung darstellt

Quantitativer Vergleich: Für einen quantitativen Vergleich wurde in Tabelle 3 für jede Lastdefinition die durchschnittliche Abweichung der berechneten CPU-Last für jeden Lauf, den Mittelwert, und die Grenzen der 90% und 95% Konfidenzintervalle⁶¹ berechnet. Zum Vergleich der Lastdefinitionen wurde dies auch jeweils mit den Differenzen von zwei Lastdefinitionen gemacht.

Aufgrund der relativ geringen Zahl an durchgeführten Testreihen⁶² und der teilweise recht hohen Standardabweichung sind die Konfidenzintervalle relativ groß.

Es fällt auf, dass in den Werten der Maximum-Lastdefinition nur eine sehr geringe Streuung der Werte vorkommt, diese aber zu den schlechtesten gehören.

Um zwei Verfahren zu vergleichen, kann man (nach [15]) die Differenzen bilden, und falls der Wert 0 in dem gewählten Konfidenzintervall vorhanden ist, dann sind die Systeme vermutlich nicht unterscheidbar. Aufgrund der großen Konfidenzintervalle⁶³, lässt sich mit einer 90% Wahrscheinlichkeit⁶⁴ feststellen, dass das rein CPU-basierende Verfahren einen leicht bessere Verteilung erzielt, als die nach dem Maximum gehende Lastdefinition. Bei den anderen Vergleichen und bei dem 95% Wahrscheinlichkeit lassen sich mit dieser Technik keine Aussagen mehr treffen.

10.4 Bewertung

Keiner der drei getesteten Lastdefinitionen konnte in dem durchgeführten Test, bei den bezüglich des Verhältnis von RAM- und CPU-Bedarfs sehr inhomogenen, virtuellen Servern eine gute Verteilung erzielen.

Die einzige Ausnahme waren in den Läufen acht und neun zu sehen, wo durch die spezielle Startreihenfolge für eine homogene Lastverteilung erreicht worden ist. In allen anderen Läufen wurden Pool Elements überlastet mit einer theoretischen Last über eins, und andere Pool Elements hatten dementsprechend geringe Auslastungen.

Dies lässt die folgenden Schlussfolgerungen zu:

- die getesteten Lastdefinitionen sind für produktiven Einsatz, bei den getesteten inhomogenen Verhältnis von CPU-Last und RAM-Bedarf, nicht geeignet, falls man eine hohe Auslastung ohne Überlast erreichen will.
- Für virtuelle Server die bezüglich des Verhältnis von CPU-Last und RAM-Bedarf relativ homogene Gruppe bilden gilt diese Einschränkung nicht, da es in dem Fall vergleichsweise leicht ist, die Auslastung zu definieren. Dies konnte auch im achten und neunten Lauf gezeigt werden, wo jeweils homogene Gruppen von virtuellen Servern gestartet wurden.

11 Auswertung der Failovertests

Bei dem Failovertest wurden die in der Tabelle 4 notierten Werte für die Unterbrechung

⁶¹unter der Annahme einer Exponentialverteilung der Werte

⁶²Aufgrund des hohen Zeitbedarfes der Testdurchführung

⁶³diese könnten mit längeren Versuchsreihen vermutlich reduziert werden

⁶⁴unter der Annahme einer Exponentialverteilung der Messwerte

Lauf	1	2	3	4	5	6	7	8
Zeit in s	70	138	69	64	163	69	167	67

Tabelle 4: Gemessene Zeiten bei den Failovertests

des Dienstes gemessen.

Auffallend ist, dass die Werte sich auf zwei Bereiche konzentrieren. Die erste Häufung liegt bei etwa 69 Sekunden Failoverzeit, und die zweite Häufung bei etwa 150 Sekunden.

Einzelne Beobachtungen zeigten, dass das Starten des `init-Daemon`⁶⁵ teilweise, im Vergleich zu anderen Systemen, ungewöhnlich viel Zeit in Anspruch nahm. Da dies das erste Programm ist, welches vom SAN bezogen wird, könnten Probleme im NBD-Setup dafür verantwortlich sein.

Bewertung der Zeiten Die gemessenen Zeiten liegen in einem Bereich, der die Reaktionszeiten eines menschlichen Operators, im Normalfall unterbietet.

Für hochverfügbare Systeme sind die Failover und Bootzeiten zu hoch, so dass dort entweder eine deutliche Verbesserung erzielt werden⁶⁶, oder der Failover auf Applikationsebene stattfinden muss.

Teil V

Zusammenfassung

12 Zusammenfassung & Ausblick

12.1 Zusammenfassung

In dieser Arbeit wurde ein Prototyp zur Verwaltung von virtuellen Servern mittels Reliable Server Pooling auf der Basis der `rsplib` realisiert.

Die durchgeführten Testläufe zeigten, dass die im Kapitel 2 formulierten Ziele, Reliable Server Pooling zur Lastverteilung und Erhöhung der Verfügbarkeit zu nutzen, zum aktuellen Zeitpunkt, nicht vollständig erfüllt werden.

Verfügbarkeit: Das Ziel die Verfügbarkeit von virtuellen Servern zu verbessern konnte gut erfüllt werden. Der automatische Failover wurde in den Tests zuverlässig und in akzeptablen Zeiten durchgeführt (siehe Kapitel 11). Dadurch werden die Ausfallzeiten eines virtuellen Servers durch Defekte bei einem Server gering genug, dass diese für sehr viele Anwendungen tolerabel sind.

⁶⁵Das Programm welches vom Kernel gestartet wird, und dann den Start der Dienste übernimmt.

⁶⁶welche gerade bei den Bootzeiten nur begrenzt möglich ist

Lastverteilung/Verwaltung: Die virtuellen Server werden gut verwaltet, wenn die Verteilung der CPU-Last möglichst homogen ist. Dies konnte in den Test nur erzielt werden, wenn homogene⁶⁷ Gruppen von virtuellen Servern gestartet wurden. Gruppen von virtuellen Server die ein gleiches/ähnliches Verhältnis von CPU-Last und RAM-Belegung auf den Pool Elements erzeugen lassen sich auch gut Verwalten, da dort sehr leicht ein eindimensionaler Lastindikator gebildet werden kann.

Bei den Testläufen mit Gruppen von virtuellen Server die ein inhomogenes Verhältnis von CPU-Last und RAM-Belegung auf den Pool Elements erzeugen, konnte keine der verwendeten Lastdefinitionen überzeugen (siehe Kapitel 10.4).

12.2 Offene Themen

Im folgenden werden einige Themen vorgestellt, die im besprochenen Themengebiet noch zu untersuchen wären.

Lastverteilung Es sollte geklärt werden, ob eine Lastdefinition existiert, die eine gleichmäßige Auslastung der Pool Elements auch bei den getesteten Lastprofilen ermöglicht. Diese Fragestellung kann verallgemeinert werden: Wie kann eine gleichmäßige Lastverteilung bei Reliable Server Pooling erzielt werden, wenn die Pool Elements für den Dienst verschiedene Ressourcen zur Verfügung stellen, deren Nutzungsgrad für jede Session (relativ) unabhängig voneinander ist⁶⁸.

Da Testläufe und Messungen an aufgebauten Systemen sehr zeitintensiv sind, wäre hier ein simulativer Ansatz mit späterem Validieren durch Testläufe sinnvoll.

Integration des persistenten Speichers In diesen Projekt wurde der persistente Speicher eines virtuellen Servers als einfach verfügbar angesehen.

Für eine vollständige Verwaltung von virtuellen Servern müsste dieser aber auch verwaltet werden. Da aufgrund der großen Datenmenge dort weder ein Verteilen auf alle Pool Elements noch die Session-State-Cookies in Frage kommt, wäre hier der Business-Card Ansatz für das Failover zu evaluieren, und evtl. mit einem passenden Backend, welches den Datenabgleich vornimmt, zu implementieren.

Verhalten beim Ausfall des Pool Users Da der Pool User der einen virtuellen Server gestartet hat, nicht in jedem Szenario auch dessen (alleiniger) Nutzer ist⁶⁹, besteht in der aktuellen Architektur das Problem, dass falls ein Pool User ausfällt⁷⁰ der virtuelle Server beendet wird. Dies verlagert die Verfügbarkeitsproblematik in diesen Fällen von dem Server auf den Client wo der Pool User läuft.

Ob und wie das Problem vermeidbar wäre, ist noch zu untersuchen.

⁶⁷Durch die eingebaute leichte Variation in der Last waren die getesteten virtuellen Server von der Last nicht vollkommen identisch.

⁶⁸In diesem Projekt waren die Ressourcen CPU-Zeit und RAM

⁶⁹z.B. ein Administrator startet einen virtuellen Server der einen Webserver enthält der Daten für Dritte bereitstellt

⁷⁰also nicht absichtlich vom Pool User heruntergefahren

Dynamische Lastverteilung Da die CPU-Last eines virtuellen Servers über die Zeit nicht konstant sein muss, wären auch Ansätze zu überprüfen in wie weit man zu späteren Zeitpunkten⁷¹ die Lastverteilung optimieren kann, indem virtuelle Server auf andere Pool Elements verlagert werden.

Sicherstellen der Eindeutigkeit eines virtuellen Servers Zu prüfen wäre auch, in wie weit man zuverlässig verhindern könnte, dass ein böartiger Pool User virtuelle Server so startet, dass Ressourcen⁷² unzulässigerweise mehrfach genutzt werden.

12.3 Kommentar

Durch das rsplib Framework war es dem Autor möglich, den Prototyp in relativ kurzer Zeit zu erstellen.

Dadurch kommt der Autor noch zu der Feststellung, dass die Entwicklung eines Programmes, welches auf Reliable Server Pooling/rsplib aufsetzt, mit geringen Aufwand verbunden ist⁷³. Die größten Probleme sind dabei die die konzeptionellen Fragen, wie sich gezeigt hat: Was muss bei einem Failover beachtet werden, wie wird die Last (bei dynamischen Pool Policys) definiert?

⁷¹also nach dem Start

⁷²das Diskimage, die „eindeutige“ ID, die IP-Adresse...

⁷³Dies war ein Ziel bei der Entwicklung des Reliable Server Pooling Frameworks.

Teil VI

Anhang

A Läufe zum Testen der Lastverteilung im PC-Pool

Im folgenden sind die Startreihenfolge und die genutzten Features dokumentiert.

A.1 Startreihenfolge und Wartezeiten

Für den achten und neunten Lauf wurde das folgende Startskript verwendet:

```
1 #!/bin/bash
  waitin=27
3 waitout=45
  for nr in 1 2 3 4 5 6 7 8 ; do
5     screen -d -m ./XenDemoClient -ram=300 -serverid=last -5-$nr \
      -blockdevice=nd$[ 40 + $nr ] -iplist=10.0.5.$nr
7     sleep $waitin
  done
9     sleep $waitout
  for nr in 1 2 3 4 5 6 7 8 ; do
11    screen -d -m ./XenDemoClient -ram=64 -serverid=last -2-$nr \
      -blockdevice=nd$[ 30 + $nr ] -iplist=10.0.2.$nr
13    sleep $waitin
  done
15    sleep $waitout
  for nr in 1 2 3 4 5 6 7 8 ; do
17    screen -d -m ./XenDemoClient -ram=64 -serverid=last -25-$nr \
      -blockdevice=nd$[ 20 + $nr ] -iplist=10.0.25.$nr
19    sleep $waitin
  done
21    sleep $waitout
  for nr in 1 2 3 4 5 6 7 8 ; do
23    screen -d -m ./XenDemoClient -ram=150 -serverid=last -10-$nr \
      -blockdevice=nd$[ $nr ] -iplist=10.0.10.$nr
25    sleep $waitin
  screen -d -m ./XenDemoClient -ram=150 -serverid=last -10-$[ $nr +8 ] \
27    -blockdevice=nd$[ $nr + 8 ] -iplist=10.0.10.$[
      $nr + 8 ]
  sleep $waitin
29 done
  sleep $waitout
```

Für die anderen Läufe (1-7, 10-15) wurde das folgendes Startskript verwendet, die verwendeten Zeiten für waitin und waitout variierten teilweise etwas.

```
#!/bin/bash
2 waitin=47
  waitout=60
4 for nr in 1 2 3 4 5 6 7 8 ; do
  screen -d -m ./XenDemoClient -ram=150 -serverid=last -10-$nr \
6     -blockdevice=nd$[ $nr ] -iplist=10.0.10.$nr
  sleep $waitin
8     screen -d -m ./XenDemoClient -ram=64 -serverid=last -2-$nr \
      -blockdevice=nd$[ 30 + $nr ] -iplist=10.0.2.$nr
10    sleep $waitin
  screen -d -m ./XenDemoClient -ram=300 -serverid=last -5-$nr \
```

```

12         -blockdevice=nd$[ 40 + $nr ] -iplist=10.0.5.$nr
    sleep $waitin
14    screen -d -m ./XenDemoClient -ram=64 -serverid=last-25-$nr \
        -blockdevice=nd$[ 20 + $nr ] -iplist=10.0.25.$nr
16    sleep $waitin
    screen -d -m ./XenDemoClient -ram=150 -serverid=last-10-$[ $nr + 8 ] \
18        -blockdevice=nd$[ $nr + 8 ] -iplist=10.0.10.$[
            $nr + 8 ]
    sleep $waitout
20 done

```

A.2 Weitere Parameter der Läufe

In der Tabelle 5 sind weitere Parameter bei den Läufen angegeben.

Lauf	Tempraiseload aktiv	Lastberechnung
1	Nein	Maximum aus CPU-Auslastung und RAM-Auslastung
2	Nein	Maximum aus CPU-Auslastung und RAM-Auslastung
3	Nein	Maximum aus CPU-Auslastung und RAM-Auslastung
4	Nein	Maximum aus CPU-Auslastung und RAM-Auslastung
5	Ja	Maximum aus CPU-Auslastung und RAM-Auslastung
6	Ja	Maximum aus CPU-Auslastung und RAM-Auslastung
7	Ja	Maximum aus CPU-Auslastung und RAM-Auslastung
8	Ja	Maximum aus CPU-Auslastung und RAM-Auslastung
9	Ja	Maximum aus CPU-Auslastung und RAM-Auslastung
10	Nein (fehlerhaft)	$0,75 * \text{CPU-Auslastung} + 0,25 * \text{RAM-Auslastung}$
11	Ja	$0,75 * \text{CPU-Auslastung} + 0,25 * \text{RAM-Auslastung}$
12	Ja	$0,75 * \text{CPU-Auslastung} + 0,25 * \text{RAM-Auslastung}$
13	Ja	$0,75 * \text{CPU-Auslastung} + 0,25 * \text{RAM-Auslastung}$
14	Ja	CPU-Auslastung
15	Ja	CPU-Auslastung

Tabelle 5: Parameter bei den Läufen

A.3 vif-Route

Das `/etc/xen/scripts/vif-route` Skript wurde bei den Failovertest wie folgt geändert.

```

#!/bin/bash
2 #
# /etc/xen/vif-route
4 #
# Script for configuring a vif in routed mode.
6 # The hotplugging system will call this script if it is specified either in
# the device configuration given to Xend, or the default Xend configuration
8 # in /etc/xen/xend-config.sxp. If the script is specified in neither of those
# places, then vif-bridge is the default.
10 #
# Usage:
12 # vif-route (add/remove/online/offline)

```

```

#
14 # Environment vars:
# vif      vif interface name (required).
16 # XENBUS_PATH path to this device's details in the XenStore (required).
#
18 # Read from the store:
# ip      list of IP networks for the vif, space-separated (default given in
20 #      this script).
#-----
22   dir=$(dirname "$0")
24   . "$dir/vif-common.sh"

26 main_ip=$(dom0_ip)

28 case "$command" in
    online)
30     ifconfig ${vif} ${main_ip} netmask 255.255.255.255 up
        echo 1 >/proc/sys/net/ipv4/conf/${vif}/proxy_arp
32     if [ "${ip}" ] ; then
        for addr in ${ip} ; do
34         arp -Ds ${addr} eth0 pub
            /usr/lib/heartbeat/send_arp -i 50 -r 5 -p /tmp/send_arp.pid eth0 $ip
            auto 10.0.10.255 ffffffff &
36         done
        fi
38     ipcmd='add'
        cmdprefix=''
40     ;;
    offline)
42     do_without_error ifdown ${vif}
        ipcmd='del'
44     cmdprefix='do_without_error'
        ;;
46 esac

48 if [ "${ip}" ] ; then
    # If we've been given a list of IP addresses, then add routes from dom0 to
50 # the guest using those addresses.
    for addr in ${ip} ; do
52     ${cmdprefix} ip route ${ipcmd} ${addr} dev ${vif} src ${main_ip}
    done
54 fi

56 handle_iptable

58 log debug "Successful_vif-route_$command_for_${vif}."
    if [ "$command" == "online" ]
60 then
        success
62 fi

```

A.4 Lastgenerator

Bei den Lastverteilungsläufen wurde die Last in den virtuellen Servern durch automatischen Aufruf des folgenden Shell-Skriptes erzeugt.

```

1 #!/bin/bash
  last=x
3 # x=2,5,10 oder 25, je nach Typ

```

```

while true; do
5   var=$(( $RANDOM / 11000 ));
    sleep $( 30 - $last + $var -1 );
7   stress -t $( $last - $var +1 ) -c 2
done

```

B Quelltext

B.1 Pakete - XenServpackets.h

```

#ifndef XENSERV_H
2 #define XENSERV_H

4 #include <stdint.h>

6
// Ensure consistent alignment!
8 #pragma pack(push)
#pragma pack(4)
10

12 #define PPID_PPP 0x29097602

14 #define XSPT_createServer 0x01
#define XSPT_create_failed 0x02
16 #define XSPT_create_success 0x03
#define XSPT_CheckRunning 0x10
18 #define XSPT_CheckRunningResponse 0x11

20 struct XenServCommonHeader
{
22     uint8_t Type;
    uint8_t Flags;
24     uint16_t Length;
};
26
struct CreateServer
28 {
    struct XenServCommonHeader Header;
30     char ServerID[32];
    char blockdevice[256];
32     char IP_List[256];
    unsigned int requested_ram;
34 };

36 struct Create_response
{
38     struct XenServCommonHeader Header;
    char ServerID[32];
40     short int ErrorCode;
// -1 unspecified error
42     // 0 erfolgreich gestartet
// 11 not enough free RAM
44     // 12 cant find/connect blockdevice
// 13 ID is allready running
46     // 50 Normal beendet
    unsigned int avaiable_ram;
48 };

```

```

50 struct CheckRunning
51 {
52     struct XenServCommonHeader Header;
53     char ServerID [32];
54     short int intervall;
55 };
56
57 struct CheckRunningResponse
58 {
59     struct XenServCommonHeader Header;
60     char ServerID [32];
61     short int responseCode;
62 };
63
64 #define XS_COOKIE_ID "<PP-TD1>"
65
66 struct XScookie
67 {
68     char ID [8];
69     char ServerID [32];
70     char blockdevice [256];
71     char IP_List [256];
72     unsigned int requested_ram;
73     bool running;
74     char Stonith [128];
75 };
76
77 #pragma pack(pop)
78
79 #endif

```

B.2 Client - XenServ.cc

```

#include "rserpool.h"
2 #include "XenServpackets.h"
#include "breakdetector.h"
4 #include "timeutilities.h"
#include "netutilities.h"
6 #include "randomizer.h"
#include "netutilities.h"
8 #include <iostream>
#include <string>
10 #ifndef ENABLE_CSP
#include "componentstatuspackets.h"
12 #endif

14
15 /* ##### Main program ##### */
16 int main(int argc, char** argv)
17 {
18     const char* poolHandle = "Xen";
19     unsigned int requested_ram = 96;
20     union rserpool_notification* notification;
21     struct rsp_info info;
22     struct rsp_sndrcvinfo rinfo;
23     char buffer [65536 + 4];
24     char str [128];
25     struct pollfd ufds;
26     struct CreateServer* create;
27     struct CheckRunning* ping;
28     char serverID [32] = "";

```

```

30     char                blockdevice[256] = "";
31     char                ipList[256] = "";
32     ssize_t             received;
33     ssize_t             sent;
34     int                 result;
35     int                 flags;
36     int                 sd;
37     int                 i;
38     // ### fuer die Ping-Pong check-Alive-Sachen
39     unsigned long long  pingInterval = 1000000;
40     unsigned long long  lastPing;
41     unsigned long long  now;
42     unsigned long long  nextPing;
43
44
45     rsp_initinfo(&info);
46     for(i = 1; i < argc; i++) {
47         std::cout << "argv[" << i << "] = " << argv[i] << "\n";
48         if(rsp_initarg(&info, argv[i])) {
49             /* rsplib argument */
50         }
51         else if(!(strcmp(argv[i], "-poolhandle=" ,12))) {
52             poolHandle = (char*)&argv[i][12];
53         }
54         else if(!(strcmp(argv[i], "-ram=" ,5))) {
55             requested_ram = (unsigned int)atol((const char*)&argv[i][5]);
56         } else if(!(strcmp(argv[i], "-serverid=" ,10))) {
57             strcpy(serverID, (char*)&argv[i][10]);
58         } else if(!(strcmp(argv[i], "-interval=" ,10))) {
59             pingInterval = 1000000ULL * atol((const char*)&argv[i][10]);
60         } else if(!(strcmp(argv[i], "-blockdevice=" ,13))) {
61             strcpy(blockdevice, (char*)&argv[i][13]);
62         } else if(!(strcmp(argv[i], "-iplist=" ,8))) {
63             strcpy(ipList, (char*)&argv[i][8]);
64         }
65         else {
66             fprintf(stderr, "ERROR: Bad argument %s\n", argv[i]);
67             exit(1);
68         }
69     }
70 #ifdef ENABLE_CSP
71     info.ri_csp_identifier = CID_COMPOUND(CID_GROUP_POOLUSER, random64());
72 #endif
73
74     puts("Xen_Service_User_-_Version_0.1");
75     puts("=====\n");
76     printf("Pool_Handle = %s\n", poolHandle);
77     printf("SerberID = %s\n", serverID);
78     std::cout << "RAM = " << requested_ram << "\n";
79
80     if(rsp_initialize(&info) < 0) {
81         fputs("ERROR: Unable to initialize rsplib\n", stderr);
82         exit(1);
83     }
84
85     sd = rsp_socket(0, SOCK_SEQPACKET, IPPROTO_SCTP);
86     if(sd < 0) {
87         perror("Unable to create RSerPool socket");
88         exit(1);
89     }
90 }

```

```

92     if(rsp_connect(sd, (unsigned char*)poolHandle, strlen(poolHandle)) < 0) {
93         perror("Unable_to_connect_to_pool_element");
94         exit(1);
95     }
96     installBreakDetector();
97
98     // Send Message
99     snprintf((char*)&str, sizeof(str),
100             "Create_Nachricht");
101     size_t dataLength = strlen(str);
102
103     create = (struct CreateServer*)&buffer;
104     create->Header.Type = XSPT_createServer;
105     create->Header.Flags = 0x00;
106     create->Header.Length = htons(sizeof(struct CreateServer));
107     strcpy(create->ServerID, serverID);
108     strcpy(create->blockdevice, blockdevice);
109     strcpy(create->IP_List, ipList);
110     create->requested_ram = requested_ram;
111     sent = rsp_sendmsg(sd, (char*)create, sizeof(struct CreateServer), 0,
112                       0, htonl(PPID_PPP), 0, 0, 0);
113     if(sent > 0) {
114         printf("Message_sent\n");
115     }
116
117     lastPing = 0;
118     while(!breakDetected() ) {
119         ufds.fd = sd;
120         ufds.events = POLLIN;
121         nextPing = lastPing + pingInterval;
122         now = getMicroTime();
123         result = rsp_poll(&ufds, 1,
124                          (nextPing <= now) ? 0 : (int)((nextPing - now) / 1000));
125
126         /* ##### Handle Pong message ##### */
127         if(result > 0) {
128             if(ufds.revents & POLLIN) {
129                 flags = 0;
130                 received = rsp_recvmmsg(sd, (char*)&buffer, sizeof(buffer),
131                                        &rinfo, &flags, 0);
132                 if(received > 0) {
133                     if(flags & MSG_RSERPOOL_NOTIFICATION) {
134                         notification = (union rserpool_notification*)&buffer;
135                         printf("\x1b[39;47mNotification:_");
136                         rsp_print_notification(notification, stdout);
137                         puts("\x1b[0m");
138                         if((notification->rn_header.rn_type == RSERPOOL_FAILOVER) &&
139                            (notification->rn_failover.rf_state ==
140                             RSERPOOL_FAILOVER_NECESSARY)) {
141                             puts("FAILOVER...");
142                             rsp_forcefailover(sd);
143                         }
144                     } else if (((const XenServCommonHeader*)buffer)->Type ==
145                               XSPT_create_failed){
146                         const Create_response* response = (const Create_response*)
147                             buffer;
148                         std::cout << "Failed_to_create_" << response->ServerID << "\n";
149                     } else if (((const XenServCommonHeader*)buffer)->Type ==
150                               XSPT_create_success){

```

```

        const Create_response* response = (const Create_response*)
        buffer;
148     std::cout << "Sucessfull_created_" << response -> ServerID <<
        "\n";
    }else if (((const XenServCommonHeader*)buffer)->Type ==
    XSPT_CheckRunningResponse){
150     const CheckRunningResponse* response = (const
        CheckRunningResponse*)buffer;
        std::cout << "Domain_" << response -> ServerID << "_lieferte_"
        Statuscode_" <<response ->responseCode << "\n";
152     }
    }
154 }
156 }

158 /* ##### Send CheckRunning message #####
    */
    if(getMicroTime() - lastPing >= pingInterval) {
160     lastPing = getMicroTime();

162     snprintf((char*)&str, sizeof(str),
        "Zeitstempel:_%llu", (unsigned long long)lastPing);
164
        ping = (struct CheckRunning*)&buffer;
166     ping->Header.Type = XSPT_CheckRunning;
        ping->Header.Flags = 0x00;
168     ping->Header.Length = htons(sizeof(struct CheckRunning) );
        strcpy(ping->ServerID, serverID);
170     ping->intervall=(short int)(pingInterval/1000000);
        rsp_sendmsg(sd, (char*)ping, sizeof(struct CheckRunning), 0,
172     0, htonl(PPID_PPP), 0, 0, 0);
    }
174 }
176

178 puts("\x1b[0m\nTerminated!");
180 rsp_close(sd);
    rsp_cleanup();
182 rsp_freeinfo(&info);
    return 0;
184 }

```

B.3 Server - Pool Element

B.3.1 Registrierung in server.cc

Datei gekürzt dargestellt. Die anderen Services sind nicht im Abdruck enthalten.

```

#include "rserpool.h"
2 #include "breakdetector.h"
#include "tagitem.h"
4 #include "netutilities.h"
#include "standardservices.h"
6 #include "fractalgeneratorservice.h"
#include "calcappservice.h"
8 #ifdef ENABLE_CSP
#include "componentstatuspackets.h"
10 #endif
...

```

```

12 #define SERVICE_XenSERV 8
   /* ##### Main program ##### */
14 int main(int argc, char** argv)
   {
16     struct rsp_info      info;
       struct rsp_loadinfo loadInfo;
18     struct TagItem      tags[8];
       double              degradation;
20     unsigned int        identifier;
       unsigned int        reregInterval = 30000;
22     unsigned int        runtimeLimit  = 0;
       unsigned int        service      = SERVICE_XenSERV;
24     const char*         poolHandle    = NULL;
       /* ===== Read parameters ===== */
26     rsp_initinfo(&info);
       tags[0].Tag = TAG_PoolElement_Identifier;
28     tags[0].Data = 0;
       tags[1].Tag = TAG_DONE;
30 #ifdef ENABLE_CSP
       info.ri_csp_identifier = CID_COMPOUND(CID_GROUP_POOLELEMENT, 0);
32 #endif
       memset(&loadInfo, 0, sizeof(loadInfo));
34     loadInfo.rli_policy = PPT_LEASTUSED;
       for(int i = 1; i < argc; i++) {
36         if(rsp_initarg(&info, argv[i])) {
           /* rsplib argument */
38         }
           if(!(strcmp(argv[i], "-identifier=", 12))) {
40             if(sscanf((const char*)&argv[i][12], "0x%x", &identifier) == 0) {
               if(sscanf((const char*)&argv[i][12], "%u", &identifier) == 0) {
42                 fputs("ERROR: _Bad_registrar_ID_given!\n", stderr);
                   exit(1);
44             }
           }
           tags[0].Data = identifier;
46 #ifdef ENABLE_CSP
           info.ri_csp_identifier = CID_COMPOUND(CID_GROUP_POOLELEMENT, tags[0].Data
48             );
       #endif
50     }
       else if(!(strcmp(argv[i], "-poolhandle=" ,12))) {
52         poolHandle = (char*)&argv[i][12];
       }
54     else if(!(strcmp(argv[i], "-rereginterval=" ,15))) {
           reregInterval = atol((char*)&argv[i][15]);
56         if(reregInterval < 10) {
           reregInterval = 10;
58         }
       }
60     else if(!(strcmp(argv[i], "-policy=" ,8))) {
           double dpf;
62         if(!(strcmp((const char*)&argv[i][8], "RoundRobin"))) {
           loadInfo.rli_policy = PPT_ROUNDROBIN;
64         }
           else if(!(strcmp((const char*)&argv[i][8], "Random"))) {
66             loadInfo.rli_policy = PPT_RANDOM;
           }
68         else if(!(strcmp((const char*)&argv[i][8], "LeastUsed"))) {
           loadInfo.rli_policy = PPT_LEASTUSED;
70         }
           else if(!(strcmp((const char*)&argv[i][8], "RandomizedLeastUsed"))) {
72             loadInfo.rli_policy = PPT_RANDOMIZED_LEASTUSED;

```

```

74     }
75     else if(sscanf((const char*)&argv[i][8], "LeastUsedDegradation:%lf",
76                 &degradation) == 1) {
77         loadInfo.rli_load_degradation = (unsigned int)rint(degradation
78                 * (double)PPV_MAX_LOAD_DEGRADATION);
79         if(loadInfo.rli_load_degradation < 0) {
80             fputs("ERROR: _Bad_LUD_degradation_value!\n", stderr);
81             exit(1);
82         }
83         else if(loadInfo.rli_load_degradation > PPV_MAX_LOAD_DEGRADATION) {
84             fputs("ERROR: _Bad_LUD_degradation_value!\n", stderr);
85             exit(1);
86         }
87         loadInfo.rli_policy = PPT_LEASTUSED_DEGRADATION;
88     }
89     else if(sscanf((const char*)&argv[i][8], "PriorityLeastUsed:%lf",
90                 &degradation) == 1) {
91         loadInfo.rli_load_degradation = (unsigned int)rint(degradation
92                 * (double)PPV_MAX_LOAD_DEGRADATION);
93         if(loadInfo.rli_load_degradation < 0) {
94             fputs("ERROR: _Bad_PLU_degradation_value!\n", stderr);
95             exit(1);
96         }
97         else if(loadInfo.rli_load_degradation > PPV_MAX_LOAD_DEGRADATION) {
98             fputs("ERROR: _Bad_PLU_degradation_value!\n", stderr);
99             exit(1);
100        }
101        loadInfo.rli_policy = PPT_PRIORITY_LEASTUSED;
102    }
103    else if(!(strcmp((const char*)&argv[i][8], "RandomizedPriorityLeastUsed")
104            )) {
105        loadInfo.rli_policy = PPT_RANDOMIZED_PRIORITY_LEASTUSED;
106    }
107    else if(sscanf((const char*)&argv[i][8], "LeastUsedDPF:%lf", &dpf) == 1)
108    {
109        if((dpf < 0.0) || (dpf > 1.0)) {
110            fputs("ERROR: _Bad_LU-DPF_DPF_value!\n", stderr);
111            exit(1);
112        }
113        loadInfo.rli_load_dpf = (unsigned int)rint(dpf * (double)
114                PPV_MAX_LOADDPF);
115        loadInfo.rli_policy = PPT_LEASTUSED_DPF;
116    }
117    else if(sscanf((const char*)&argv[i][8], "LeastUsedDegradationDPF:%lf:%lf
118    ",
119                &degradation, &dpf) == 2) {
120        if((dpf < 0.0) || (dpf > 1.0)) {
121            fputs("ERROR: _Bad_LU-DPF_DPF_value!\n", stderr);
122            exit(1);
123        }
124        loadInfo.rli_load_dpf = (unsigned int)rint(dpf * (double)
125                PPV_MAX_LOADDPF);
126        loadInfo.rli_load_degradation = (unsigned int)rint(degradation
127                * (double)PPV_MAX_LOAD_DEGRADATION);
128        if(loadInfo.rli_load_degradation < 0) {
129            fputs("ERROR: _Bad_LU-DPF_degradation_value!\n", stderr);
130            exit(1);
131        }
132        else if(loadInfo.rli_load_degradation > PPV_MAX_LOAD_DEGRADATION) {
133            fputs("ERROR: _Bad_LU-DPF_degradation_value!\n", stderr);
134            exit(1);
135        }
136    }

```

```

130     loadInfo.rli_policy = PPT_LEASTUSED_DEGRADATION_DPF;
132     }
132     else if(sscanf((const char*)&argv[i][8], "WeightedRoundRobin:%u",
134         &loadInfo.rli_weight) == 1) {
134         loadInfo.rli_policy = PPT_WEIGHTED_ROUNDROBIN;
136     }
136     else if(sscanf((const char*)&argv[i][8], "WeightedRandom:%u",
138         &loadInfo.rli_weight) == 1) {
138         loadInfo.rli_policy = PPT_WEIGHTED_RANDOM;
140     }
140     else if(sscanf((const char*)&argv[i][8], "WeightedRandomDPF:%u:%lf",
142         &loadInfo.rli_weight, &dpf) ==
142         2) {
142         if((dpf < 0.0) || (dpf > 1.0)) {
144             fputs("ERROR: _Bad_WRAND-DPF_DPF_value!\n", stderr);
144             exit(1);
146         }
146         loadInfo.rli_weight_dpf = (unsigned int)rint(dpf * (double)
148             PPV_MAX_WEIGHTIDPF);
148         loadInfo.rli_policy = PPT_WEIGHTED_RANDOM_DPF;
150     }
150     else {
150         fprintf(stderr, "ERROR: _Bad_policy_setting_<%s>!\n", argv[i]);
152         exit(1);
152     }
154 }
154 else if(!(strcmp(argv[i], "-runtime=" ,9))) {
156     runtimeLimit = atol((const char*)&argv[i][9]);
156 }
158 ...
158 else if(!(strcmp(argv[i], "-Xen"))) {
160     service = SERVICE_XenSERV;
160 }
162 }
162 /* ===== Print startup message ===== */
162 printf("Starting_service_");
164 if(service == SERVICE_ECHO) {
164     printf("Echo");
166 }
168 ...
168 else if(service == SERVICE_XenSERV) {
170     printf("Xen_Server_Service");
170 }
172 puts("...");
172 /* ===== Start requested service ===== */
172 if(service == SERVICE_ECHO) {
174     EchoServer echoServer;
174     echoServer.poolElement("Echo_Server_-_Version_1.0",
176         (poolHandle != NULL) ? poolHandle : "EchoPool",
176         &info, &loadInfo,
178         reregInterval, runtimeLimit,
178         (struct TagItem*)&tags);
180 }
182 ...
182 else if(service == SERVICE_XenSERV) {
184     size_t maxThreads = 20;
184     XenServer::XenServerSettings settings;
184     settings.FailureAfter = 0;
186     for(int i = 1; i < argc; i++) {
188         if(!(strcmp(argv[i], "-pppfailureafter=", 17))) {
188             settings.FailureAfter = atol((const char*)&argv[i][17]);
188         }

```

```

190     else if (!(strcmp(argv[i], "-pppmaxthreads=", 15))) {
191         maxThreads = atol((const char*)&argv[i][15]);
192     }
193 }
194 TCPLikeServer::poolElement("XEN_Server_-_Version_0.1",
195                             (poolHandle != NULL) ? poolHandle : "Xen",
196                             &info, &loadInfo,
197                             maxThreads,
198                             XenServer::XenServerFactory,
199                             NULL,
200                             XenServer::XenServerinitializeService,
201                             XenServer::XenServerfinishService,
202                             XenServer::loadUpdateHook,
203                             (void*)&settings,
204                             reregInterval, runtimeLimit,
205                             (struct TagItem*)&tags);
206 }
207 rsp_freeinfo(&info);
208 return(0);
209 }

```

B.3.2 standardservices.h

Datei gekürzt dargestellt. Die anderen Services sind nicht im Abdruck enthalten.

```

1 #ifndef STANDARDSERVICES_H
2 #define STANDARDSERVICES_H
3 #include "rserpool.h"
4 #include "tcplikeserver.h"
5 #include "udplikeserver.h"
6 #include <time.h>
7 #include <libvirt/libvirt.h>
8 class XenServer : public TCPLikeServer
9 {
10     // Parameter
11     static const int loadUpdateIntervall=30;
12     static const int loadUpdateFaktor=3;
13     static const bool temp_Load_raise=true;
14     static const int maxvServer=50;
15     static const float memory_weight=0.0F;
16     // Variablen
17     struct vserverSetting
18     {
19         unsigned long long CPU_time;
20         char ServerID [32];
21         char blockdevice [256];
22         char ipList [256];
23         time_t last_update;
24         bool domainRunning;
25         unsigned long requested_ram;
26     };
27     struct dom0Setting
28     {
29         time_t lastUpdate;
30         unsigned long long CPU_Seconds;
31         double Load;
32         virDomainPtr Ptr;
33         virDomainInfoPtr info;
34     };
35     virDomainInfoPtr domainInfo;

```

```

37  virDomainPtr          domainPtr;

39  static virConnectPtr virCon;
static virNodeInfo  nodeInfo;

41
static char          stonith[128];
43  static unsigned long freeRAM;
static dom0Seting  dom0Settings;
45  vserverseting  vserversettings;

47  public:
struct XenServerSettings
49  {
    size_t FailureAfter;
51  };
static double loadUpdateHook(const double load);
53  static bool  XenServerinitializeService(void* userData);
static void  XenServerfinishService(void* userData);
55  XenServer(int rserverpoolSocketDescriptor,
            XenServer::XenServerSettings* settings);
57  ~XenServer();
static TCPLikeServer* XenServerFactory(int sd, void* userData);
59

61  protected:
EventHandlingResult handleCookieEcho(const char* buffer, size_t bufferSize);
63  EventHandlingResult handleMessage(const char* buffer,
                                   size_t      bufferSize,
65                                   uint32_t   ppid,
                                   uint16_t   streamID);

67  private:
static bool  temp_Load_raise_active;
69  static unsigned int getFreeRAM(void);
static unsigned int getRAM(void);
71  static short int getCpuCount(void);
static char* getHostname(void);
73  EventHandlingResult createDomain(void);

75
XenServerSettings Settings;
77  uint64_t      ReplyNo;
size_t          Replies;
79

protected:
81  void finishSession(EventHandlingResult result);

83  };

```

B.3.3 standardservices.cc

Datei gekürzt dargestellt. Die anderen Services sind nicht im Abdruck enthalten.

```

1 #include "standardservices.h"
#include "netutilities.h"
3 #include "timeutilities.h"
#include "stringutilities.h"
5 #include <stdlib.h>
#include <iostream>
7 #include <string>
#include <stdio.h>
9 #include <time.h>

```

```

#include <libvirt/libvirt.h>
11 #include <libvirt/virterror.h>
/*
13 #####
##### Xen Server Service #####
15 #####
*/
17 #include "XenServpackets.h"
unsigned long XenServer::freeRAM=0;
19 char XenServer::stonith[128]="";
XenServer::dom0Seting XenServer::dom0Setings;
21 virConnectPtr XenServer::virCon;
virNodeInfo XenServer::nodeInfo;
23 bool XenServer::temp_Load_raise_active=false;
// ##### Constructor #####
25 XenServer::XenServer(int rserpoolSocketDescriptor ,
XenServer::XenServerSettings* settings)
27 : TCPLikeServer(rserpoolSocketDescriptor)
{
29 Settings = *settings;
vserverSetings.domainRunning=false;
31 domainInfo = new virDomainInfo();
}
33 // ##### Destructor #####
XenServer::~XenServer()
35 {
}
37 // ##### Hostname fuer stoneith ermitteln #####
char* XenServer::getHostname(void)
39 {
FILE *in;
41 char buff[128];
char* buff2;
43
if (!(in = popen("hostname", "r"))) {
45 return 0;
}
47 buff2=new char[128];
/* read the output of netstat, one line at a time */
49 while (fgets(buff, sizeof(buff), in) != NULL) {
delete buff2;
51 buff2=new char[128];
strcpy(buff2, buff);
53 }
/* close the pipe */
55 pclose(in);

57 return buff2;
}
59 // ##### Finish of thread #####
void XenServer::finishSession(EventHandlingResult result)
61 {
printf("Finish_Aufgerufen\n");
63 if (vserverSetings.domainRunning) {
char *Command = NULL;
65 int returncode;
// Vorerst so gelassen, wird selten aufgerufen, und waere schwer
nachzuprogrammieren
67 Command = new char[640];
strcpy(Command, "xm_shutdown_w_");
69 strcat(Command, vserverSetings.ServerID);
returncode=system(Command);
}
}

```

```

71     delete [] Command;
72     if ( returncode != 0 ) {
73         Command = new char[640];
74         strcpy(Command, "xm_destroy_");
75         strcat(Command, vserverSetings.ServerID);
76         system(Command);
77     }
78     freeRAM = freeRAM + vserverSetings.requested_ram;
79 }
80 }
81 // ##### Create a PingServer thread #####
TCPLikeServer* XenServer::XenServerFactory(int sd, void* userData)
82 {
83     return(new XenServer(sd, (XenServer::XenServerSettings*)userData));
84 }
85 // ##### createDomain #####
86 EventHandlingResult XenServer::createDomain(void) {
87     // Domain up&running
88     vserverSetings.domainRunning=true;

91     /* ===== Send cookie ===== */
92     struct XScookie newcookie;
93     strncpy((char*)&newcookie.ID, XS_COOKIE_ID, sizeof(newcookie.ID));
94     strcpy(newcookie.ServerID, vserverSetings.ServerID);
95     strcpy(newcookie.IP_List, vserverSetings.ipList);
96     strcpy(newcookie.blockdevice, vserverSetings.blockdevice);
97     strcpy(newcookie.Stonith, stonith);
98     newcookie.requested_ram=vserverSetings.requested_ram;
99     newcookie.running=vserverSetings.domainRunning;
100     rsp_send_cookie(RSerPoolSocketDescriptor,
101                   (unsigned char*)&newcookie, sizeof(newcookie),
102                   0, 0);
103     printf("Cookie_versendet\n");

104     // Startzeit festlegen
105     vserverSetings.last_update = time(NULL);
106     vserverSetings.CPU_time=0;

107     char *Command = NULL;
108     Command = new char[640];
109     int returncode;
110     if (Command)
111     {
112         strcpy(Command, "xm_create_template_");
113         strcat(Command, "memory=");
114         char buffer [33];
115         sprintf(buffer, "%d", vserverSetings.requested_ram);
116         strcat(Command, buffer);
117         strcat(Command, "_name=");
118         strcat(Command, vserverSetings.ServerID);
119         strcat(Command, "\"_vif='ip='");
120         strcat(Command, vserverSetings.ipList);
121         strcat(Command, "'_disk='phy:'");
122         strcat(Command, vserverSetings.blockdevice);
123         strcat(Command, ",hda1,w'");
124         std::cout << "xm-Aufruf_<_<< Command << "\n";
125         returncode=system(Command);
126         delete [] Command;
127         domainPtr=virDomainLookupByName(virCon, vserverSetings.ServerID);
128         // unterscheidung zwischen ok & Fehler
129         if (domainPtr != NULL && virDomainGetInfo(domainPtr, domainInfo) == 0 ) {
130             char create_successBuffer[sizeof(struct Create_response)];

```

```

133     Create_response* create_success = (Create_response*)&create_successBuffer;
create_success->Header.Type = XSPT_create_success;
135     create_success->Header.Flags = 0x00;
create_success->Header.Length = htons(sizeof(create_success));
137     strcpy(create_success->ServerID, vserverSetings.ServerID);
create_success->ErrorCode = 0;
139     create_success->avaiable_ram = freeRAM;
rsp_sendmsg(RSerPoolSocketDescriptor,
141         (char*)create_success, sizeof(create_success), 0,
0, htonl(PPID_PPP), 0, 0, 0);
143     printf("OK_versendet\n");

145     // Setze die Load auf einen "schaetzwert",
// damit nicht zuviele Am Anfang auf einen Server kopiert werden
147     setLoad((double)vserverSetings.requested_ram*1024/(double)nodeInfo.memory);
// reduziere die Menge an verfuegbaren RAM
149     freeRAM = freeRAM - (vserverSetings.requested_ram*1024);
return(EHR_Okay);
151 } else {
153     printf("Fehler_beim_Domainerstellen\n");
char create_failedBuffer[sizeof(struct Create_response)];
155     Create_response* create_failed = (Create_response*)&create_failedBuffer;
create_failed->Header.Type = XSPT_create_failed;
157     create_failed->Header.Flags = 0x00;
create_failed->Header.Length = htons(sizeof(create_failed));
159     strcpy(create_failed->ServerID, vserverSetings.ServerID);
create_failed->ErrorCode = -1;
161     create_failed->avaiable_ram = freeRAM;
rsp_sendmsg(RSerPoolSocketDescriptor,
163         (char*)create_failed, sizeof(create_failed), 0,
0, htonl(PPID_PPP), 0, 0, 0);
165     return(EHR_Abort);
167 }
169 }
171 return(EHR_Abort);
173 }
// ##### Handle cookie echo #####
175 EventHandlingResult XenServer::handleCookieEcho(const char* buffer,
size_t bufferSize)
177 {
const struct XSCookie* cookie = (const struct XSCookie*)buffer;
179 if( (bufferSize != sizeof(XSCookie)) ||
(strncmp((const char*)&cookie->ID, XS_COOKIE_ID, sizeof(cookie->ID))) ) {
181     puts("Received_bad_cookie_echo!");
return(EHR_Abort);
183 }
185 //Setze die Variablen
strcpy(vserverSetings.ServerID, cookie->ServerID);
187 strcpy(vserverSetings.ipList, cookie->IP_List);
strcpy(vserverSetings.blockdevice, cookie->blockdevice);
189 vserverSetings.requested_ram=(cookie->requested_ram);
191 // Ausgabe der Daten
std::cout << "Failover_mit_folgenden_Daten:_\n";
193 std::cout << "ServerID=_ " << vserverSetings.ServerID << "\n";
std::cout << "RAM=_ " << vserverSetings.requested_ram << "\n";

```

```

195 std::cout << "Blockdevice_\n" << vserverSetings.blockdevice << "\n";
196 std::cout << "IPs_\n" << vserverSetings.ipList << "\n";
197 std::cout << "Stoneiths_\n" << cookie->Stonith << "\n";

199 /* ===== try to setup vServer ===== */

201 // aktualisiere die Menge an freien RAM
202 std::cout << "FreeRAM_\n" << freeRAM << "RequestedRAM:\n"
203 << vserverSetings.requested_ram * 1024 << "\n";
204 if ( freeRAM < (vserverSetings.requested_ram * 1024) ) {
205     printf("zuwenig_RAM_zweig\n");
206     // um die Load kurzfristig zu erhoehen
207     if (temp_Load_raise) {
208         printf("Erhoehetemporaer_Load_wegen_RAM_mangel\n");
209         temp_Load_raise_active=true;
210     }
211     return(EHR_Abort);
212 }
213 //optional weitere Tests

215 // da nun die offensichtlichsten Probleme ausgeschlossen sind, versuchen wir
    einen Start

217 if (cookie->running == true) {
218     std::cout << "Stoneith_mit_\n" << cookie->Stonith << "\n";
219 }

221 return(createDomain());
222 }
223 // ##### Handle message #####
EventHandlingResult XenServer::handleMessage(const char* buffer,
224 size_t bufferSize,
225 uint32_t ppid,
226 uint16_t streamID)
227 {
228     ssize_t sent;
229     if(bufferSize >= (ssize_t)sizeof(XenServCommonHeader)) {
230         if (((const XenServCommonHeader*)buffer)->Type == XSPT_createServer){
231             const CreateServer* request = (const CreateServer*)buffer;
232             if(request->Header.Type == XSPT_createServer) {
233                 printf("Request_angekommen\n");
234                 if(ntohs(request->Header.Length) >= (ssize_t)sizeof(struct CreateServer)) {
235
236                     //Setze die Variablen
237                     strcpy(vserverSetings.ServerID, request->ServerID);
238                     strcpy(vserverSetings.ipList, request->IP_List);
239                     strcpy(vserverSetings.blockdevice, request->blockdevice);
240                     vserverSetings.requested_ram=request->requested_ram;
241                     vserverSetings.domainRunning=false;
242
243                     // Ausgabe der Daten
244                     std::cout << "Request_mit_folgenden_Daten:\n";
245                     std::cout << "ServerID_\n" << vserverSetings.ServerID << "\n";
246                     std::cout << "RAM_\n" << vserverSetings.requested_ram << "\n";
247                     std::cout << "Blockdevice_\n" << vserverSetings.blockdevice << "\n";
248                     std::cout << "IPs_\n" << vserverSetings.ipList << "\n";
249
250                     /* ===== try to setup vServer ===== */

251
252                     // egal was passiert, erst einmal ein cookie senden, damit danach Failover
253                     klappt
254                     /* ===== Send cookie ===== */

```

```

255     struct XSCookie cookie;
256     strncpy((char*)&cookie.ID, XS_COOKIE_ID, sizeof(cookie.ID));
257     strcpy(cookie.ServerID, vserverSettings.ServerID);
258     strcpy(cookie.IP_List, vserverSettings.ipList);
259     strcpy(cookie.blockdevice, vserverSettings.blockdevice);
260     strcpy(cookie.Stonith, stonith);
261     cookie.requested_ram=vserverSettings.requested_ram;
262     cookie.running=vserverSettings.domainRunning;
263     rsp_send_cookie(RSerPoolSocketDescriptor,
264                   (unsigned char*)&cookie, sizeof(cookie),
265                   0, 0);
266     printf("Cookie_versendet\n");
267
268     std::cout << "FreeRAM_" << freeRAM << "RequestedRAM:_"
269               << vserverSettings.requested_ram * 1024 << "\n";
270     if ( freeRAM < (vserverSettings.requested_ram * 1024) ) {
271         printf("zuwenig_RAM_zweig\n");
272         char create_failedBuffer[sizeof(struct Create_response)];
273         Create_response* create_failed = (Create_response*)&create_failedBuffer;
274
275         create_failed->Header.Type = XSPT_create_failed;
276         create_failed->Header.Flags = 0x00;
277         create_failed->Header.Length = htons(sizeof(create_failed));
278         strcpy(create_failed->ServerID, vserverSettings.ServerID);
279         create_failed->ErrorCode = 1;
280         create_failed->available_ram = freeRAM;
281         printf("vorm_Senden_abort\n");
282         sent = rsp_sendmsg(RSerPoolSocketDescriptor,
283                          (char*)create_failed, sizeof(create_failed), 0,
284                          0, htonl(PPID_PPP), 0, 0, 0);
285         printf("ABORT, nicht genug RAM\n");
286         // um die Load kurzfristig zu erhoehen
287         if (temp_Load_raise) {
288             printf("Erhoehe_temporaer_Load_wegen_RAM_mangel\n");
289             temp_Load_raise_active=true;
290         }
291         return(EHR_Abort);
292     }
293     //optional weitere Tests
294
295     // da nun die offensichtlichsten Probleme ausgeschlossen sind, versuchen
296     // wir einen Start
297     return(createDomain());
298 }
299 }
300 }else if (((const XenServCommonHeader*)buffer)->Type == XSPT_CheckRunning){
301
302     if (virDomainGetInfo(domainPtr, domainInfo) == 0 ) {
303         // Sende "pong"
304         char responseBuffer[sizeof(struct CheckRunningResponse)];
305         CheckRunningResponse* response = (CheckRunningResponse*)&responseBuffer;
306         response->Header.Type = XSPT_CheckRunningResponse;
307         response->Header.Flags = 0x00;
308         response->Header.Length = htons(sizeof(response));
309         strcpy(response->ServerID, vserverSettings.ServerID);
310         response->responseCode = domainInfo->state;
311         rsp_sendmsg(RSerPoolSocketDescriptor,
312                  (char*)response, sizeof(response), 0,
313                  0, htonl(PPID_PPP), 0, 0, 0);
314     }

```

```

317 // Falls der Returncode != breche ab
    if (domainInfo->state == VIR_DOMAIN_NOSTATE
        || domainInfo->state == VIR_DOMAIN_RUNNING || domainInfo->state ==
319         VIR_DOMAIN_BLOCKED
        || domainInfo->state == VIR_DOMAIN_PAUSED ) {
        time_t jetzt=time(NULL);
321        if ( ( jetzt - vserverSetings.last_update ) >=loadUpdateIntervall) {
            // Load ermitteln
323            long difftime=jetzt - vserverSetings.last_update;
            if (difftime == 0 ){
325                difftime=1;
            }
327
            double new_load=(double)(domainInfo->cpuTime - vserverSetings.CPU_time
329                )
                /(double)(difftime* nodeInfo.cpus*1000000000LL);
            vserverSetings.last_update=jetzt;
331            vserverSetings.CPU_time=domainInfo->cpuTime;
            //Load gewichten
333            new_load=(new_load + (getLoad()*loadUpdateFaktor)) / (1+(
                loadUpdateFaktor));
            setLoad(new_load);
335            std::cout << "Neue_Load_fuer_" << vserverSetings.ServerID
                << "_ist_" << new_load << "\n";
337        }
        return(EHR_Okay);
339    }
341 }
343 }
    return(EHR_Abort);
345 }
// ##### Update Hook #####
347 // Reche Domain-0 und die RAM-Auslastung mit ein
double XenServer::loadUpdateHook(const double load)
349 {
    double cpuload;
351    double memoryLoad=0L;
    double returload;
353    time_t jetzt=time(NULL);

355    // Zeit seit dem letzten Update
    long timediff=(long) difftime(jetzt ,dom0Setings.lastUpdate);
357    if (timediff >=loadUpdateIntervall) {
        // Load der Dom0 ermitteln
359        if (virDomainGetInfo(dom0Setings.Ptr ,dom0Setings.info) != 0) {
            // fehler, einfach Load durchreichen
361            std::cout << "Fehler_beim_Ermitteln_der_Dom0-Load";
            return (load);
363        }

365        cpuload=(double)(dom0Setings.info->cpuTime
            - dom0Setings.CPU_Seconds)/(double)(timediff*nodeInfo.cpus
                *1000000000LL);
367        dom0Setings.lastUpdate=jetzt;
        dom0Setings.CPU_Seconds=dom0Setings.info->cpuTime;
369        //Load gewichten
        cpuload=(cpuload + (dom0Setings.Load*loadUpdateFaktor)) / (1+(loadUpdateFaktor
            ));
371        freeRAM = getFreeRAM();
        //Ausgabe

```

```

373     printTimeStamp(stdout);
374     std::cout <<"Dom0:_" << cpuload << "_RAM:_"
375             << ((double)(nodeInfo.memory - freeRAM) / (double)nodeInfo.memory <<
                 "\n");
376     //<< " Gesamt: " << (long long)ServerList->LoadSum / PPV_MAX_LOAD
377
378     // temp_Load_raise_active auf false setzen
379     temp_Load_raise_active=false;
380 } else {
381     cpuload=dom0Setings.Load;
382 }
383
384 // bei temp_Load_raise_active==true
385
386 if (temp_Load_raise_active==true ) {
387     return (1L);
388 }
389
390 if (cpuload >= 0L && cpuload <=1 ) {
391     dom0Setings.Load=cpuload;
392     cpuload=cpuload+load;
393 } else {
394     cpuload=load;
395 }
396 // Und nun der RAM
397 memoryLoad=((double)(nodeInfo.memory - freeRAM) / (double)nodeInfo.memory ;
398 returload = memoryLoad * memory_weight + (1 - memory_weight ) *cpuload;
399 // if ( memoryLoad > cpuload ) {
400 //     returload=memoryLoad;
401 // } else {
402 //     returload=cpuload;
403 // }
404
405 if (returload > 1L ) {
406     return (1L);
407 } else {
408     return(returload);
409 }
410 }
411 bool XenServer::XenServerinitializeService(void* userData)
412 {
413     if ( virInitialize() != 0) {
414         std::cout << "WARNING:_vir_Initializew_failed\n" ;
415         return false;
416     }
417     virCon=virConnectOpen(NULL);
418     if (virCon != NULL ) {
419         if (virNodeGetInfo(virCon, (virNodeInfo*)&nodeInfo) == 0) {
420             strcpy(stonith, getHostname());
421             dom0Setings.Ptr = virDomainLookupByID(virCon,0);
422             if ( dom0Setings.Ptr != NULL ) {
423                 dom0Setings.info = new virDomainInfo();
424                 virDomainGetInfo(dom0Setings.Ptr, dom0Setings.info);
425                 dom0Setings.CPU_Seconds = dom0Setings.info->cpuTime;
426                 freeRAM = getFreeRAM();
427                 return true;
428             }
429         }
430     }
431     std::cout << "WARNING:_Initialisierung_mislungen" ;
432     return false;
433 }

```

```

void XenServer::XenServerfinishService(void* userData)
435 {
    virConnectClose(virCon);
437 }
// ##### RAM-Ermittlung #####
439 unsigned int XenServer::getFreeRAM(void)
{
441     int liste[maxvServer+1];
    unsigned long ram=nodeInfo.memory;
443
    // der RAM fuer den Hypervisor
445 ram-=12*1024 + (ram /1024 ) * 7;
    int ret=virConnectListDomains(virCon , liste ,maxvServer+1) ;
447     if ( ret == -1) {
        return 0;
449     }
    for (int i=0; i<ret; i++) {
451         if (liste[i] != 0) {
            ram-=virDomainGetMaxMemory(virDomainLookupByID(virCon , liste [ i ] ) );
453         }
    }
455     ram=dom0Setings.info->memory;
    return ram;
457 }

```

Literatur

- [1] Computer Laboratory - Xen virtual machine monitor, 08 2007. URL <http://www.cl.cam.ac.uk/research/srg/netos/xen/>.
- [2] Xen Community, 08 2007. URL <http://xen.xensource.com/>.
- [3] Reliable Server Pooling (rserpool) Charter, 08 2007. URL <http://www.ietf.org/html.charters/rserpool-charter.html>.
- [4] XenStorage - Xen Wiki, 06 2006. URL <http://wiki.xensource.com/xenwiki/XenStorage>.
- [5] Xen v3.0 - Users' Manual, 08 2007. URL <http://bits.xensource.com/Xen/docs/user.pdf>.
- [6] the virtualization API, 08 2007. URL <http://libvirt.org/>.
- [7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.
- [8] G. Cockburn. Xen Live Migration with iSCSI, 07 2006. URL <http://www.performancemagic.com/iscsi-xen-howto/>.
- [9] T. Dreibholz. An Overview of the Reliable Server Pooling Architecture. In *Proceedings of the 12th IEEE International Conference on Network Protocols 2004*, Berlin/-Germany, Oct. 2004. URL <http://citeseer.ist.psu.edu/dreibholz04overview.html>. Poster presentation.

- [10] T. Dreibholz. *Reliable Server Pooling – Evaluation, Optimization and Extension of a Novel IETF Architecture*. PhD thesis, University of Duisburg-Essen, Faculty of Economics, Institute for Computer Science and Business Information Systems, Mar. 2007. URL <http://duepublico.uni-duisburg-essen.de/servlets/DerivateServlet/Derivate-16326/Dre2006-final.pdf>.
- [11] T. Dreibholz. Thomas Dreibholz’s Reliable Server Pooling (RSerPool) Page. URL <http://tdrwww.exp-math.uni-essen.de/dreibholz/rserpool/>.
- [12] T. Dreibholz. rsplib – Eine Open Source Implementation von Reliable Server Pooling. In *Proceedings of the Linuxtage in Essen*, Essen/Germany, Sept. 2006. URL <http://tdrwww.iem.uni-due.de/dreibholz/rserpool/rserpool-publications/Linuxtage2006.pdf>.
- [13] T. Dreibholz and E. P. Rathgeb. Implementing the Reliable Server Pooling Framework. In *Proceedings of the 8th IEEE International Conference on Telecommunications*, volume 1, pages 21–28, Zagreb/Croatia, June 2005. ISBN 953-184-081-4. URL <http://citeseer.ist.psu.edu/dreibholz05implementing.html>. ISBN 953-184-081-4.
- [14] J. Formann. RAM-Bedarf von XEN, 03 2007. URL <http://www.formann.de/2007/03/ram-bedarf-von-xen/>.
- [15] B. Müller-Clostermann. Diskrete Simulation. Vorlesungsskript zur gleichnamigen Veranstaltung, 10 2006.
- [16] M. Störchle. Virtualization in a Nutshell. Vortrag im Rahmen der IBM System z University, 05 2007. URL <http://sysmod.icb.uni-due.de/SysMod/zuniversity/ibmdocs/sessions/Virtualisierung.pdf>.