Technische Universität Dortmund
Fakultät Informatik, Lehrstuhl 12
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Markus Buschhoff
Dipl.-Inf. Boguslaw Jablkowski

PG 574: CyPhyControl

- Endbericht -

2. Juni 2014

Inhaltsverzeichnis

1. Zielbeschreibung 6

2. Einführung 8

2.1. Einführung in CPS . . . 8

2.2. Grundlegende Begrifflichkeiten . . . 9

2.2.1. Live Migration . . . 9

2.2.2. Hochverfügbarkeit . . . 10

2.2.3. Echtzeitfähigkeit . . . 12

3. Anforderungsanalyse 13

4. Bewertung vorhandener Technologien 14

4.1. Virtualisierung allgemein . . . 14

4.1.1. Hypervisor . . . 15

4.1.2. Vollständige Virtualisierung und Paravirtualisierung . . . 15

4.2. Virtualisierungs-Lösungen . . . 16

4.2.1. KVM (Proxmox VE) . . . 18

4.2.2. XEN . . . 21

4.3. Gast-Systeme . . . 24

4.3.1. TinyCore . . . 25

4.3.2. CiAO . . . 25

4.3.3. Mini-OS . . . 26

4.4. Monitoring-Systeme . . . 27

4.5. Benutzeroberfläche . . . 29

4.5.1. Anpassung vorhandener Software oder eigene Entwicklung . . . 29

4.5.2. Webbasierte Anwendung oder Desktop-Anwendung . . . 32

4.6. Lokales Scheduling . . . 32

4.6.1. Vorinstallierte Scheduler . . . 33

4.6.2. RT-XEN . . . 35

4.7. Globales Scheduling . . . 38

4.7.1. Vergleich von Lastverteilungsalgorithmen . . . 38

4.7.2. Real-Time-Calculus . . . 41

4.8. Implementierungs-Sprachen . . . 46

5. Systementwurf 47

5.1. Überblick . . . 47

5.1.1. Logischer Aufbau . . . 48

5.1.2. Physikalischer Aufbau . . . 50

5.2. Hochverfügbarkeitskonzept . . . 52

5.2.1. Single Master . . . 52

5.2.2. Master + Backup . . . 53
5.2.3. Master + Backup + Node . . . 54
5.3. Frontend . . . 55
5.4. LMU . . . 56
5.4.1. Kriterien zur Verteilung virtueller Maschinen . . . 57
5.4.2. Dynamische Anpassung des Systemzustandes . . . 58
5.4.3. Konfigurations- und Erweiterungsmöglichkeiten . . . 60
5.5. Monitoring . . . 60
5.6. Lokales Scheduling . . . 63
5.7. Gastsystem . . . 64
5.8. Systemstart . . . 67

6. Endprodukt **68**
6.1. Host-Konfiguration . . . 68
6.1.1. Xen als Hypervisor . . . 68
6.1.2. Systemarchitektur . . . 69
6.1.3. Fehlertoleranz mit Remus . . . 70
6.2. Frontend . . . 71
6.2.1. Genutzte Technologien . . . 72
6.2.2. Struktur und Konfiguration . . . 73
6.3. Load Management Unit . . . 74
6.3.1. Schnittstellenbeschreibung . . . 75
6.3.2. Komponentenbeschreibung . . . 78
6.3.3. Funktionsbeschreibung . . . 81
6.4. Monitoring . . . 88
6.4.1. Schnittstellenbeschreibung . . . 89
6.4.2. Funktionsbeschreibung . . . 91
6.5. Lokales Scheduling . . . 99
6.5.1. Fixed-Priority Scheduling . . . 100
6.5.2. Xen Scheduler Architektur . . . 100
6.5.3. Rate-Monotonic Scheduler - Architektur . . . 101
6.5.4. Hypervisor Änderungen . . . 107
6.5.5. libxc und xend/xm Schnittstelle . . . 110
6.6. Gastsystem . . . 111

7. Beispielapplikation **112**
7.1. Beschreibung des Distanzschutzes . . . 112
7.2. Das Szenario . . . 113
7.3. Implementierung . . . 114

8. Testkonzepte **116**
8.1. Frontend . . . 117

8.2. Systemtests . . . 119
8.3. Load Management Unit . . . 121

9. Evaluation **124**
9.1. Lokales Scheduling . . . 124
9.1.1. Task Example . . . 124
9.1.2. Ablauf der VMs . . . 124
9.1.3. Das Zeitquantum . . . 129
9.1.4. Zeit in der do_schedule . . . 130
9.1.5. Evaluation . . . 131
9.2. Ausfallzeiten der verschiedenen Redundanzkonzepte . . . 133
9.2.1. Versuchsaufbau . . . 134
9.2.2. Darstellung der Ergebnisse und Auswertung . . . 137
9.3. Auslösezeiten der Beispielapplikation . . . 141
9.3.1. Versuchsaufbau . . . 142
9.3.2. Ablauf der Messungen . . . 142
9.3.3. Auswertung . . . 143
9.4. Auslösezeiten der Beispielapplikation in CiAO . . . 145

10. Ungelöste Probleme **148**
10.1. Load Management Unit . . . 148
10.2. CiAO . . . 148

11. Projektorganisation **150**

12. Zusammenfassung **154**
12.1. Mögliche Erweiterungen . . . 154
12.1.1. N-Version Programming . . . 155
12.1.2. Verwendung von RT-Xen . . . 155
12.1.3. Abhängige Tasks . . . 156

A. Dokumentation der LMU-Funktionen **156**

B. Liste aller Fehlerfälle **240**

C. Liste aller Host-Logging-Daten **243**

D. Liste aller Application-Logging-Daten **244**

E. Dokumentation der Frontend-Implementierung **245**
E.1. Verzeichnisstruktur . . . 245
E.2. global_config.php . . . 245

F. Funktionstests für LMU **246**

G. Ergebnisse zur Evaluation der Remus-Replikation 264

H. Literaturverweise 297

1. Zielbeschreibung

(Heng Liu)

Motivation In den letzten Jahren waren Cyber-physikalische Systeme (CPS) im Bereich der Industrie und Informatik ein zentrales Thema. Viele große und komplexe technische Systemen werden heutzutage durch CPS überwacht und gesteuert. Der Begriff der CPS wurde von Edward Lee geprägt. Eine seiner frühen Definitionen ist [1]:

> „Cyber-Physical Systems are integrations of computation with physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa."

CPS entstehen aus der Verschmelzung von vernetzten eingebetteten Rechensystemen mit mechanischen und elektronischen Elementen der pyhsischen Umgebung. Wenn man allerdings die häufig genannten Beispiele betrachtet, z.B. die dezentrale Erzeugung und Verteilung von Energie („Smart Grid"), Fahrerassistenz- und Verkehrssysteme auf der Basis einer Vernetzung von Fahrzeugen untereinander und mit ihrer Umgebung („Car-to-X"), vernetzte medizitechnische Systeme mit Kopplung von körpernaher Sensorik und Fernüberwachung („eHealth"), oder andere moderne Produktionsanlagen, wird es erkannt, dass CPS mehr als klassische eingebettete Systeme sind.

Es wird deutlich, dass eine erhebliche Herausforderung in der Zukunft aufkommen würde, weil die Komplexität der CPS sich kontinuierlich erhöht. Deswegen erwartet man, dass künftige physische Systeme verstärkt durch Software überwacht und gesteuert werden sollen und einen höheren Integrationgrad haben werden. Dazu benötigt man eine entsprechende Technik für Systemintegration, die Virtualisierung. Mithilfe der Virtualisierung ist es möglich, dass mehrere Anwendungen als verschiedene virtuelle Maschinen auf einer Ausführungsplattform ausgeführt werden können. Außerdem sollte die Virtualisierung noch verschiedene Fehlertoleranzmechanismen bieten.

Die Zuverlässigkeit des Betriebs von Überwachung- und Steuerung ist abhängig von mindestens zwei wichtigen Voraussetzungen: Fehlertoleranz und Echtzeitfähigkeit. Aus diesem Grund soll eine virtualisierte Ausführungsplattform für Steuerung der CPS entworfen werden, auf der die echtzeitkritische Anwendungen ausgeführt werden können.

Zielsetzung Das grundlegende Ziel der Projektgruppe ist der Entwurf und die Implementierung einer virtualisierten Ausführungsplattform für Cyber-physikalische Anwendungen, für die die geeignete Virtualisierungslösung zur Verfügung gestellt werden soll. Des Weiteren soll diese Plattform gut skalierbar und erweiterbar sein.

Die entworfene Virtualisierungsinfrastruktur soll aus mehreren Host-Systemen bestehen. Anschießend sollen minimale Gastsysteme erstellt werden, die auf den Host-Systemen ausgeführt werden können. Des Weitern soll es möglich sein, die Gastsysteme von einem Host zu einem anderen zu migrieren sowie mehrere Gastsysteme parallel auszuführen.

Um die Host-Systeme zu steuern und zu überwachen, ist der Entwurf und die Implementierung eines Monitoring-Systems unbedingt notwendig, das wichtige Daten und Werte von gesamtem System sammeln und verarbeiten kann. Des Weiteren soll eine grafische Benutzeroberfläche erstellt werden, auf der alle relevanten Daten dargestellt werden können und von der die komplette Infrastruktur gesteuert werden kann.

Ein wichtiger Schwerpunkt liegt darauf, ein geeignetes Fehlererkennungskonzept und Fehlertoleranzmechanismen zu entwickeln. Bei einem Fehler soll dieser Zustand schnellstmöglich erkannt werden, z.B. bei einem Ausfall eines Host- oder Gastsystems. Falls es zu einem solchen Fehler kommt, kann das Gastsystem zum Beispiel auf einem Host mit geringer Auslastung neu gestartet werden. Somit kann eine bessere Konfiguration beim Fehlerfall für das Gesamtsystem festgestellt werden.

Schließlich soll ein Prototyp der Plattform implementiert und evaluiert werden. Während des Projektes soll die vollständige Dokumentation angefertigt werden, das abschließend in einer Abschlusspräsentation vorgestellt werden soll.

2. Einführung

2.1. Einführung in CPS

(Sebastian Struwe) In diesem Kapitel wird der Begriff eines Cyber Physikalischen Systems (CPS) definiert. Nach der Definition wird ein Beispiel für ein CPS gegeben. Insbesondere soll der Bezug zur Projektgruppe (PG) herausgestellt werden.
Ein CPS besteht aus drei Teilen:

1. dem physikalischen Teil, der typischerweise den mechanischen Teil beschreibt
2. der Plattform, die aus Sensoren, Aktoren und Rechnern besteht,
3. und dem Netzwerk, welche die Aufgabe hat, die verschiedenen Plattformen miteinander zu verbinden und so eine Kommunikation zu ermöglichen.

Dabei kann das Netzwerk kabelgebunden oder auch drahtlos sein.
Ein Beispiel für ein CPS stellt ein intelligentes Stromnetz dar: Bisher war das Stromnetz eher zentral aufgebaut, was z.B. der Fall ist, wenn ein Kraftwerk eine Stadt versorgt. Dies hat sich jedoch durch Nutzung von Sonnen-, Wind-, und Wasserkraft geändert. Hierbei wird die Energie an vielen verschiedenen Stellen eingespeist. Um diese Energie vernünftig steuern und auch verwerten zu können, sind intelligente Stromnetze nötig. In einem Haushalt bestehen diese aus einem intelligenten Steuergerät, das bestimmt, wann welches Gerät laufen darf. Dies funktioniert nur bei Geräten, die auf Vorrat arbeiten können, da bei den erneuerbaren Energiequellen nicht immer gleich viel Energie zur Verfüngung steht. So kann z.B. das Elektroauto geladen werden oder der Kühlschrank kann kühlen, wenn viel Energie vorhanden ist. In diesem Beispiel finden sich alle Teile des CPS wieder.
In der Projektgruppe (PG) wird nur der Plattform- und Netzwerkteil betrachtet. Hierbei wird ein hochverfügbares System entwickelt, das echtzeitfähig auf Daten reagiert, die von einem simulierten Sensor zur Verfügung gestellt werden. Dabei wird ein simulierter Aktor geschaltet. Da dies auf mehreren Rechnern geschieht, wird hierbei auch der Netzwerkteil betrachtet.
Wiederholend kann gesagt werden, dass ein CPS aus drei Teilen besteht: dem physikalischen Teil, der Plattform und dem Netzwerk. Bei der PG wird nur die Plattform und der Netzwerkteil betrachtet.

2.2. Grundlegende Begrifflichkeiten

(Sebastian Struwe) In diesem Abschnitt werden grundlegende Begriffe, die in der Anforderungsanalyse verwendet werden, geklärt.

- Virtualisierung: Virtualisierung bezeichnet das Erzeugen von nicht physikalischer Hardware, welches durch emulation erfolgt. In unserem Fall wird für das Gastsystem die komplette Hardware virtualisiert. Dadurch können vorhandene Ressourcen effektiver genutzt werden.
- Gastsystem: Darunter versteht man das Betriebssystem, das auf der emulierten Hardware läuft.
- Virtuelle Maschine: Als virtuelle Maschine wird ein Rechner bezeichnet, der aus dem Gastsystem und der emulierten Hardware besteht.
- Hypervisor: Als Hypervisor wird eine Software bezeichnet, die das Ausführen von virtuellen Maschinen erlaubt.
- Echtzeit: Es kann garantiert werden, dass eine Anwendung innerhalb einer bestimmten Zeit antwortet.
- Deadline: Als Deadline wird ein Zeitpunkt bezeichnet, bis zu dem eine Anwendung spätestens ein Ergebnis liefern muss.
- Monitoring-System: Das Monitoring-System überwacht die echtzeitkritischen Anwendungen und reagiert auf auftretende Deadline-Überschreitungen.
- Host-System: Das Host-System setzt sich aus dem Hypervisor und der Hardware zusammen.
- Fehlertoleranz: Die Fehlertoleranz beschreibt die Robustheit gegen auftretende Fehler. So soll das System trotz eines auftretenden Fehlers in der Hard- oder Software weiterhin verfügbar sein.
- Dienst/Applikation: Der Dienst oder die Applikation beschreibt eine menge von virtuellen Maschienen, die das gleiche Systemabbild ausführen.

2.2.1. Live Migration

(Tim Harde) Unter Live-Migration versteht man im Allgemeinen den Transfer einer virtu-

ellen Maschine von einem physischen Rechner zu einem anderen unter Beibehaltung des aktuellen Systemzustandes (Betriebssystem und Anwendungen). Durch die Verwendung von Live-Migration wird dabei die Ausfalldauer der durch diese virtuelle Maschine bereitgestellten Dienste minimiert.

Mögliche Einsatzgebiete sind dabei insbesondere die Wartung oder die Lastverteilung innerhalb eines Pools an Virtualisierungsservern. Zu Wartungszwecken können so beispielsweise alle auf einem Server aufgeführten virtuellen Maschinen auf einen anderen verschoben werden, ohne dabei die Bereitstellung der Dienste zu unterbrechen.

Der eigentliche Migrationsprozess von virtuellen Maschinen kann auf Basis von unterschiedlichen Kriterien bewertet werden. Neben den offensichtlichen Bewertungskriterien, wie der Dauer des gesamten Migrationsprozesses oder der tatsächlichen Ausfalldauer (Downtime) der bereitgestellten Dienste existieren noch weitere Kriterien, um die Qualität zu bewerten. Auf diese Kriterien (Endbenutzertransparenz und CPU-Zeit) soll im folgenden Abschnitt kurz eingegangen werden.

Bei der Durchführung einer Live-Migration stellt sich immer die Frage, wie transparent ein solcher Prozess für den Endbenutzer ist. Negative Auswirkungen wie Session-Unterbrechungen oder Nichtverfügbarkeit von Diensten sollten vermieden beziehungsweise minimiert werden. Des Weiteren bringt die Durchführung eines Migrationsprozesses noch weitere Nebeneffekte mit sich: unter anderem wird ein nicht unerheblicher Anteil an CPU-Zeit für die Migration benötigt, der im regulären Betrieb nicht angefallen wäre – ähnliches gilt für die benötigte Netzwerkbandbreite zur Synchronisation des Systemzustandes. Somit kann die Migration einer Maschine zu einer unbeabsichtigten Latenzerhöhung und immensen Beeinträchtigung anderer Dienste führen.

2.2.2. Hochverfügbarkeit

(Tim Harde, Vasco Fachin) Als Hochverfügbarkeit bezeichnet man die Fähigkeit, ein System (bzw. die durch das System bereitgestellten Dienste) mit einer erwarteten Verfügbarkeit von mindestens 99,999% kontinuierlich auszuführen. [26].

Der Begriff der Hochverfügbarkeit wird allerdings im IT-Bereich nicht immer homogen verwendet. Als Beispiel kann man hier die Veröffentlichungen [23] und [24]

heranziehen: die Begriffe Hochverfügbarkeit und Fehlertoleranz werden in den beiden Quellen komplett gegensätzlich definiert. Hochverfügbarkeit erfüllt in [23] höhere Anforderungen als Fehlertoleranz; in [24] ist die Hierarchie genau umgekehrt. Des Weiteren nimmt die Firma Citrix Systems eine Unterscheidung zwischen unterschiedlichen Hochverfügbarkeitsstufen für XenServer vor [6]:

- **Level 1** (core failover, recovery and restart): Diese Stufe besteht aus der Erkennung einer ausgefallenen virtuellen Maschine und dem Neustart der VM auf einem anderen Virtualisierungsserver.
- **Level 2** (high availability with component-level fault tolerance): Diese Stufe besteht in der Erkennung von fehlerhaften Netzwerk- oder Speicherkomponenten und kann dabei insbesondere den Ausfall von einzelnen Komponenten kompensieren.
- **Level 3** (continuous availability with system-level fault tolerance): Die höchste Verfügbarkeitsstufe ermöglicht die Hochverfügbarkeit eines Systems selbst bei einem kompletten Systemausfall. Diese Stufe zeichnet sich insbesondere durch die Erhaltung des Applikations- und Speicherzustandes aus, so dass ein Failover ohne Serviceunterbrechung möglich ist.

In [25] wird eine Abgrenzung der beiden Begrifflichkeiten vorgenommen. Hochverfügbarkeit wird hier definiert als:

> "[...] a system that is designed to avoid the loss of service by reducing or managing failures as well as minimizing planned downtime for the system [...]. Continuous availability means non-stop service, that is, there are no planned or unplanned outages at all [...] HA [High Availability] does not imply continuous availability [...]" [1]

und Fehlertoleranz als:

> "[...] is not a degree of availability so much as a method for achieving very high levels of availability. A fault-tolerant system is characterized by redundancy in most hardware components, including CPU, memory, I/O subsystems, and other elements. A fault-tolerant system is one that has the ability to continue service in spite of a hardware or software

[1] Ibid., S. 5

failure. However, even fault-tolerant systems are subject to outages from human error. Note that High Availability does not imply fault tolerance [...]" [2]

Im Rahmen der Projektgruppe ist nach dieser Definition ein Fehlertoleranz-Mechanismus von zentraler Bedeutung.

2.2.3. Echtzeitfähigkeit

(Sebastian Struwe) In der Domäne der Cyber-physikalischen Systeme sind harte Deadlines allgegenwärtig und die Garantie von Echtzeitverhalten eine große Herausforderung. Um die Einhaltung solcher Garantien gewährleisten zu können, ist es extrem wichtig, die genaue Auslastung der verwendeten Systemkomponenten zu kennen. Die Auslastung des gesamten Systems ist dabei von unterschiedlichen Faktoren abhängig. Im Bezug auf die Projektgruppe können drei Bereiche ausgemacht werden, die für die Echtzeitfähigkeit ausschlaggebend sind.

Der erste Bereich bezieht sich auf das verteilen der Gastsysteme auf die verschiedenen Hosts. Dabei darf ein Host durch die Gastsysteme nur so stark belastet werden, dass jedes Gastsystem dem ihm zugesicherte CPU-Zeit erhält.

Der zweite Bereich bezieht sich auf den Scheduler des Host-Systems. Dieser muss gewährleisten, dass das Gastsystem genug Rechenzeit erhält. Wenn hier ungünstige Werte gewählt werden, das Gastsystem also nicht ausreichend CPU-Zeit zugewiesen bekommt, ist ein Einhalten der Schranke nicht möglich.

Der dritte Bereich beinhaltet das Gastsystem, welches dafür sorge tragen muss, dass der Task mit Echtzeitanforderungen genug CPU-Zeit zugeteilt bekommt. Dies bedeutet, dass ein Echtzeitfähiges Gastsystem benötigt wird.

Nur wenn diese drei Eigenschaften erfüllt sind, kann eine Einhaltung der Deadlines garantiert werden. Außerdem müssen für das Netzwerk realistische Annahmen getroffen werden, die durch die Verwendung einer entsprechenden Infrastruktur zu

[2] Ibid., S. 6

gewährleisten ist.

3. Anforderungsanalyse

(Daniel Stoller) Durch den steigenden Integrationsdruck im Bereich cyber-physikalischer Systeme ergibt sich der Bedarf nach einer Infrastruktur, in der eine Anwendung auf einem beliebigen Host-System ausgeführt werden kann und in der auch mehrere Anwendungen gleichzeitig auf dem selben Host-System ausgeführt werden können. Eine Lösung für diese Anforderung ist dabei die Virtualisierung (vorgestellt im Kapitel 4.1), weshalb in Kapitel 4.2 unterschiedliche Virtualisierungslösungen evaluiert werden. Die verschiedenen Anwendungen laufen auf Gastsystemen, welche hinsichtlich ihrer Effizienz, ihrem Funktionsumfang und auch des Speicherplatzbedarfs ausgewählt werden müssen. Dies geschieht in Kapitel 4.3.

Eine wichtige Anforderung im Rahmen der Projektgruppe ist die Fehlertoleranz und die damit verbundene Hochverfügbarkeit. Um diesen Anforderungen gerecht zu werden, bedarf es entsprechender Monitoring-Systeme (siehe Kapitel 4.4), die ein plötzlich auftretendes Fehlverhalten einer laufenden Anwendung erkennen sowie eine entsprechende Fehlerbehandlung, die von einer weiteren Komponente durchgeführt werden muss, einleiten.

Um dem Benutzer die aktuellen Ausführungszeiten und eventuelle Deadline-Überschreitungen der Anwendungen mitzuteilen, müssen diese Daten vom Monitoring-System aufgezeichnet werden.

Eine weitere Anforderung ist die Echtzeitfähigkeit des Systems, aus der sich weitere Anforderungen an verschiedene Bereiche des Systems ergeben. Bei Einsatz mehrer Host-Systeme müssen zunächst die auszuführenden virtuellen Maschinen so auf die Host-Systeme verteilt werden, dass deren Rechenkapazität groß genug ist, um die Einhaltung der Deadlines zu gewährleisten. Diese Verteilung wird im nachfolgenden als „globales Scheduling“ bezeichnet und wird in Kapitel 4.7 diskutiert. Des Weiteren muss jeder virtuellen Maschine auf einem Host-System genug Rechenzeit zugewiesen werden. Lösungen für dieses im Weiteren „lokales Scheduling“ genannte Problem finden sich in Kapitel 4.6. Außerdem muss der Anwendung, die auf dem Gastsystem innerhalb der virtuellen Maschine ausgeführt wird, ausreichend Rechen-

zeit zukommen. Diese Echtzeitfähigkeit ist eine weitere Anforderung an die Gastsysteme, welche in Kapitel 4.3 untersucht werden. Alle drei genannten Bedingungen müssen erfüllt werden, um eine echtzeitfähige Ausführungsplattform zu realisieren.

Des weiteren muss die Benutzeroberfläche für mehrere solcher Host-Systeme speziellen Anforderungen genügen: Deadline-Überschreitungen sowie Anwendungslaufzeiten müssen dem Benutzer zeitnah mitgeteilt werden und das System muss auch bei vielen physikalischen Hosts und virtuellen Maschinen noch gut zu überblicken sein. Eine Bewertung möglicher Technologien zur Realisierung einer periodisch aktualisierten Benutzeroberfläche finden sich in Kapitel 4.5.

Alle Technologien, welche für die Realisierung der oben genannten Bestandteile des Systems verwendet werden, müssen zusätzlich hinsichtlich ihrer Zuverlässigkeit bewertet werden. Bei eigenständig entwickelten Lösungen spielt dabei die Programmiersprache eine wesentliche Rolle, weshalb einige in Kapitel 4.8 verglichen werden. Diese kann dabei zusätzliche Auswirkungen auf die Skalierbarkeit sowie die Performanz und damit auch die Echtzeitfähigkeit haben.

Für die abschließend durchzuführende Evaluation muss das System anhand verschiedener Qualitäts-Metriken bewertet werden. Um die nötigen Messwerte zur Bestimmung dieser Metriken zu erhalten, bedarf es einer echtzeitkritischen Beispiel-Anwendung, die auf dem entwickelten System ausgeführt wird. Eine solche Beispielapplikation kann auch während der Entwicklung als Grundlage für System-Tests zum Einsatz kommen, die mögliche Fehler aufdecken beziehungsweise die Korrektheit von System-Funktionen sicherstellen. Dafür wurde Im Rahmen der Projektgruppe die Funktion eines Distanzschutzes entwickelt, auf die in Kapitel 7 eingegangen wird.

Da die Virtualisierung den wichtigsten Bestandteil des benötigten Systems ausmacht, wird diese nun näher erläutert.

4. Bewertung vorhandener Technologien

4.1. Virtualisierung allgemein

(Vasco Fachin) In diesem Unterkapitel wird zunächst auf grundlegende Virtualisierungs-

techniken eingegangen. Dabei werden zuerst die unterschiedlichen Typen von Hypervisoren gegenübergestellt und anschließend die Unterschiede zwischen Voll- und Paravirtualisierung näher erläutert.

4.1.1. Hypervisor

Die Basis einer Virtualisierungsplattform ist ein Hypervisor (auch *Virtual Machine Monitor* genannt). Die Aufgabe eines Hypervisors ist die Erstellung und Verwaltung von virtuellen Maschinen. Hierzu zählen insbesondere die Zuweisung von Hauptspeicher und Rechenzeit. Es existieren zwei Arten von Hypervisoren (siehe Abbildung 1):

- Typ 1 (*native* oder *bare-metal*): der Hypervisor läuft direkt auf der Hardware (wie beispielsweise Xen oder VMWare ESX);
- Typ 2 (*hosted*): der Hypervisor läuft als Applikation eines darunterliegenden Betriebssystems (wie z.B Java Virtual Machine oder VirtualBox).

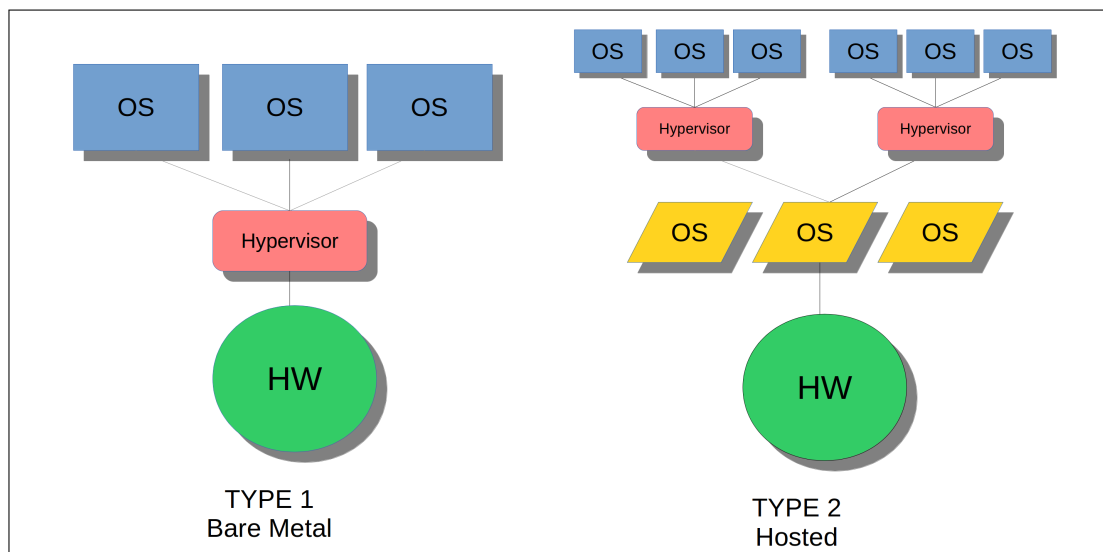


Abbildung 1: Arten von Hypervisoren

4.1.2. Vollständige Virtualisierung und Paravirtualisierung

Es existieren unterschiedliche Ansätze zur Virtualisierung von Systemen. Dabei werden zwei Virtualisierungsarten am häufigsten verwendet: Voll- und Paravirtualisie-

rung.

Vollständige Virtualisierung, oder auch *Hardware Virtual Machine* (HVM), bedeutet, dass der Hypervisor die Hardware-Ebene simuliert. Das Betriebssystem in der VM kann dabei ohne Anpassung ausgeführt werden. Wenn eine VM versucht auf ein Gerät zuzugreifen, fängt der Hypervisor diesen Zugriff ab und simuliert das angesprochene Gerät [16].

Die Softwaresimulation der Hardware ist dabei relativ langsam; aus diesem Grund wurde ein alternativer Ansatz zur Virtualisierung - die *Paravirtualisierung* - entwickelt. Paravirtualisierung bedeutet, dass die Hardware-Ebene nicht simuliert, sondern nur eine partielle Abstraktion der Hardware dem Betriebssystem zur Verfügung gestellt wird [16]. Das Ziel ist die Verringerung des E/A-Overheads [15], der beispielsweise bei Festplatten- und Netzwerkoperationen anfällt. In diesem Fall ist das laufende Betriebssystem der VM so angepasst, dass Zugriffe auf E/A-Geräte über eine spezielle Schnittstelle direkt an den Hypervisor weitergeleitet werden; die VM sendet E/A-Anforderungen: dadurch wird der Emulationsaufwand erheblich verringert.

Kernel-Instruktionen werden zu *Hypercalls* umgewandelt und ebenfalls auf der Hypervisor-Ebene durchgeführt. Das bedeutet, dass auch hier das virtualisierte Betriebssystem modifiziert werden muss. Diese Modifikation wird in allen gängigen Linux-Distributionen beispielsweise durch die so genannte *paravirt extension operations* (kurz: PVOPS) realisiert.

Bei der Verwendung von *Vollvirtualisierung* benötigt die verwendete CPU eine besondere Virtualisierungsunterstützung (AMD-V bzw. Intel VT). Bei paravirtualisierten Betriebssystemen ist eine derartige Virtualisierungsunterstützung auf der Hardware-Ebene nicht notwendig, was der wichtigste Vorteil dieser Virtualisierungstechnik ist.

4.2. Virtualisierungs-Lösungen

(Tim Harde, Vasco Fachin) Wie bereits erwähnt wurde, existieren eine Vielzahl von unterschiedlichen Virtualisierungslösungen. Allerdings existiert meist kein Fehlertoleranz-Mechanismus, wie er beispielsweise in Xen (in Form von Remus) integriert ist. Ohne

eine vergleichbare Implementierung ist der Einsatz der entsprechenden Virtualisierungslösung allerdings nicht möglich oder mit extremem Aufwand verbunden, da ein entsprechender Mechanismus sonst für den Einsatz in der Projektgruppe entwickelt, implementiert und getestet werden müsste.

Zu Beginn der Projektgruppe wurden einige Virtualisierungsumgebungen evaluiert. Xen und KVM (*Kernel-based Virtual Machine*) wurden besonders intensiv getestet, die Ergebnisse sind deshalb in einen eigenen Kapitel (4.2.1 bzw. 4.2.2) vorgestellt. Anschließend erfolgt eine Kurzvorstellung von unterschiedlichen Lösungsansätzen, die nach der Evaluation für den Einsatz in der Projektgruppe aufgrund unterschiedlicher Gründe nicht weiter in Betracht gezogen wurden.

Nova Nova ist ein Micro-Hypervisor, der 2010 an der TU Dresden entwickelt wurde. Das Ziel war es, eine minimale Virtualisierungslösung zu entwickeln, die einen hohen Anspruch an den Aspekt der Sicherheit legt [22]. Jedoch ist das System noch relativ neu, weshalb bisher keine praktische Anwendung entwickelt wurde. Außerdem existiert für NOVA kein Fehlertoleranz-Mechanismus; aus diesem Grund wurde das System nicht weiter betrachtet.

Hosted-Hypervisor Wie bereits erwähnt wurde, kann ein Hypervisor auch auf einem Betriebssystem basieren. Hosted Hypervisors sind beispielsweise QEMU, VMWare Workstation und VirtualBox.

Diese Virtualisierungslösungen sind aber ungeeignet, um ein verteiltes Echtzeitsystem aufzubauen, da in diesem Fall die VMs als normaler Prozess des Betriebssystems behandelt werden. Aus diesem Grund kann man keinerlei Garantien für eventuelle Reaktionszeiten oder einen Anteil der CPU-Zeit abgeben.

Hyper-V Microsoft vertreibt seit 2008 seinen eigenen Hypervisor (bare-metal), der nicht Open-Source ist. Dies macht die notwendige Entwicklung von zusätzlichen Komponenten zur Erweiterung innerhalb der Projektgruppe unmöglich.

VMWare ESX VMWare ist einer der führenden Hersteller, wenn es um die Entwicklung von Virtualisierungslösungen jeglicher Art geht. Problematisch ist hierbei

allerdings erneut, dass es sich bei der ESX-Technologie ausschließlich um kommerzielle Closed-Source-Produkte handelt, was eine Entwicklung von zusätzlichen Komponenten oder den Austausch des Scheduling-Verfahrens unmöglich macht. Insgesamt existieren ausgereifte Mechanismen zur Fehlertoleranz, Hochverfügbarkeit und Failover. Problematisch ist allerdings, dass sich mit VMWare ESX keinerlei Echtzeitgarantien realisieren lassen und die Nichtverfügbarkeit des Quellcodes die Implementierung der notwendigen Erweiterungen verhindert. Aufgrund der oben genannten Einschränkungen kommt ein Einsatz von VMWare ESX allerdings innerhalb der Projektgruppe nicht in Frage.

4.2.1. KVM (Proxmox VE)

(Tim Harde) Zur Evaluation von KVM als Virtualisierungstechnik wurde die Virtualisierungsplattform Proxmox Virtual Environment in der Version 2.3 ausgewählt.

Proxmox VE ist eine von der Firma Proxmox Server Solutions GmbH entwickelte Open Source-Virtualisierungsumgebung, die auf Debian Linux basiert. Als Virtualisierungstechnik bietet Proxmox dabei neben KVM ebenfalls Unterstützung für OpenVZ (eine Virtualisierungslösung für Linux Container), was aufgrund der eingeschränkten Auswahl an möglichen Betriebssystemen (nur Linux-Systeme werden von OpenVZ als Gast-Betriebssysteme unterstützt) für die weitere Evaluation nicht in Frage kam.

Das System verwendet dabei den Kernel 2.6.32 (inklusive KSM[3]) sowie die aktuellste Version von KVM.

Installation Die Installation von KVM gestaltet sich insgesamt als unkompliziert. Auf der Homepage wird ein CD-Image zum Download angeboten, mit dem das Debian-Grundsystem sowie die integrierten Erweiterungen für Proxmox VE problemlos installiert werden können.

Während der Installation müssen lediglich grundlegenden Konfigurationsparameter (Zielpartition, Land, Zeitzone, Tastaturlayout, Administratorkennwort, Hostname,

[3] **Kernel Samepage Merging**: eine Technik zur Deduplizierung von Speicherseiten, die es ermöglicht, identische Speicherinhalte unterschiedlicher Virtueller Maschinen zu nur einer Instanz zusammenzufassen.

IP, Subnetzmaske, Default-Gateway und DNS-Server) festgelegt werden.

Die Evaluation von Proxmox VE wurde mit einem Cluster aus zwei physikalischen Servern durchgeführt.

Administration Die Konfiguration und Verwaltung des Systems erfolgt über ein integriertes Webinterface, über das die meisten administrativen Aufgaben (Erstellung von Virtuellen Maschinen, Live-Migration, HA-Konfiguration, Logfile-Analyse, etc.) durchgeführt werden können. Die beispielhafte Ansicht der Administrationsoberfläche findet sich in Abbildung 2.

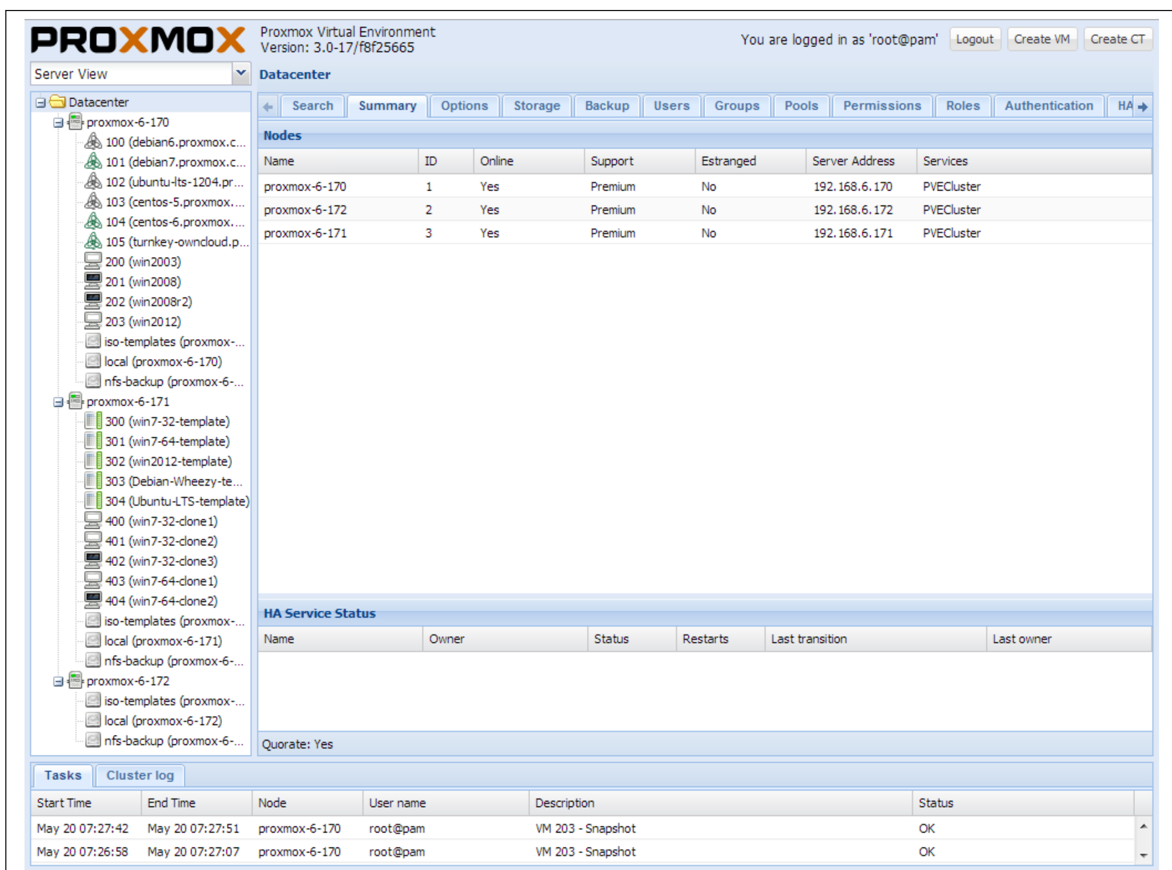


Abbildung 2: Administratoroberfläche Proxmox VE. (Quelle: [2])

Das bereitgestellte Webinterface eignet sich im Großen und Ganzen gut für die Administration der Systeme. Auf den ersten Blick sind alle relevanten Konfigurationsparameter logisch gruppiert. Die Administrationsoberfläche lässt sich zudem bei

Bedarf erweitern, der Hersteller stellt diverse API-Clients auf seiner Webseite bereit [5].

Evaluation Während der Evaluationsphase wurden unterschiedliche Experimente durchgeführt, um die Virtualisierungsumgebung zu testen. Hierbei wurde insbesondere auf die Unterstützung von CiAO und TinyCore als Gast-Betriebssystem sowie die Funktionalität von Live-Migration und Hochverfügbarkeit großen Wert gelegt. Nachfolgend finden sich die entsprechenden Ergebnisse der Evaluation.

Gast-Betriebssysteme Bei der Unterstützung der unterschiedlichen Gast-Betriebssysteme existieren keinerlei Einschränkungen. Die Test-VMs (Ubuntu, CiAO und TinyCore) konnten alle problemlos auf dem Proxmox VE-Cluster (Type: HVM, also Vollvirtualisierung) ausgeführt werden. Dabei gab es keinen bemerkenswerten Konfigurationsaufwand.

Live-Migration Die Live-Migration konnte bei allen unterschiedlichen Test-VMs ebenfalls erfolgreich getestet werden. Test-Applikationen setzten die Ausführung nach der Migration auf dem neuen Host-System ohne Probleme fort - auf eine Messung der tatsächlichen Downtime wurde an dieser Stelle allerdings verzichtet.

Hochverfügbarkeit Proxmox VE verfügt über ein integriertes Hochverfügbarkeitskonzept. Dieses Konzept sieht vor, dass im Falle eines Ausfalls eines Hostsystems die darauf ausgeführten (und als hochverfügbar konfigurierten) virtuellen Maschinen auf einem anderen physikalischen Server neu gestartet werden.

Mag dieses Konzept im Bezug auf Hochverfügbarkeit von Mailservern, Datenbankservern, etc. ausreichend sein, so gilt dies für das hier vorgesehene Szenario in der Domäne der Cyber-Physical Systems nicht. Im Falle des Ausfalls eines Hostsystems (und dem daraus resultierenden Neustart der betroffenen VMs auf einem anderen Host) können die harten Deadlines und entsprechenden Reaktionszeiten nicht mehr garantiert werden.

Mit dem *Kemari Project* [4] existieren zwar Ansätze, Synchronisationsmechanismen für KVM und Qemu zu implementieren, um im Falle eines Ausfalls eines Hostsys-

tems oder eines auftretenden Fehlers in einer VM einen automatischen Failover auf einen weiteren Host bei minimaler Downtime zu ermöglichen - leider ist dieses Projekt allerdings bereit im Jahre 2011 aufgegeben und seitdem nicht weiterentwickelt worden.

Fazit Aufgrund der einfachen Konfiguration und der vorhandenen Unterstützung für viele unterschiedliche Gastsysteme ist Proxmox VE eine vielversprechende Virtualisierungsumgebung. Aufgrund der nicht vorhandenen Fehlertoleranzmechanismen und einem für den Einsatz im Bereich der Cyber-Physikalischen Systeme nicht ausreichenden Hochverfügbarkeitskonzeptes kommt PVE für den Einsatz im Rahmen der Projektgruppe allerdings nicht in Frage.

4.2.2. XEN

(Vasco Fachin) In Rahmen der Entwicklung einer Virtualisierungsplattform wurde auch *Xen* als Hypervisor evaluiert. Xen wurde als Open Source-Projekt an der *Cambridge University* im Jahre 2003 entwickelt und veröffentlicht [18].

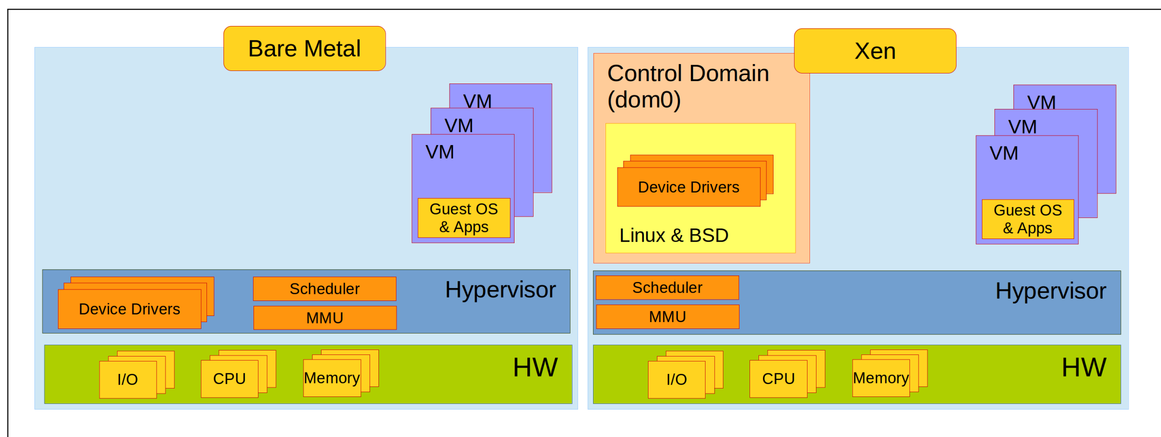


Abbildung 3: Bare-Metal Hypervisor- und Xen-Architektur nach [19]

Architektur Xen ist ein *bare-metal* Hypervisor und wird somit direkt auf der Hardware ausgeführt. Der wichtigste Unterschied zum einem typischen *bare-metal Hypervisor* ist die Tatsache, dass die Gerätetreiber nicht direkt im Hypervisor implementiert sondern stattdessen in eine hochprivilegierte virtuelle Maschine - die Dom0 -

ausgelagert sind (siehe Abbildung 3). Innerhalb des Hypervisors befinden sich lediglich die notwendigsten Mechanismen wie Speicherverwaltung, VM-Scheduling und Schnittstellen zu den einzelnen virtuellen Maschinen.

- **VM-Scheduling**: Standardmäßig sind in Xen zwei Scheduling-Verfahren (Credit Scheduler und Simple Earliest Deadline First (SEDF) , für eine detaillierte Beschreibung siehe 4.6) enthalten. Diese Verfahren können über Kernel-Parameter beim Booten ausgewählt werden.
- **VM-Schnittstellen**: Xen stellt zur Kommunikation mit den einzelnen VMs zwei Methoden zur Verfügung. Einerseits Interrupts, die von einer VM ausgelöst werden, werden in Hypercalls umgewandelt und vom Hypervisor bearbeitet. Andererseits nach der Bearbeitung wird die entsprechende Benachrichtigung durch einen asynchronen Event-Mechanismus an die VM ausgeliefert.
- **Speicherverwaltung**: Jede VM erhält einen eigenen Adressraum, somit ist jede virtuelle Maschine von den anderen strikt getrennt.

Diese Methoden sind Teil der sogenannten Isolation. Sämtliche Geräte (Speicher, I/O Geräte und CPU) sind voneinander getrennt, die VMs können sich somit nicht gegenseitig direkt beeinflussen.

Domain 0 In Xen wird eine virtuelle Maschine auch als Domäne bezeichnet. Generell lassen sich zwei Arten von Domänen unterscheiden: DomUs und Dom0. DomUs (*unprivileged domains*) sind nicht-privilegierte VMs, die über keinen direkten Zugriff auf die Hardware verfügen.

Die Dom0 hingegen ist ein Gastbetriebssystem, die von Xen direkt nach dem Start des Hypervisors geladen wird und direkten Zugriff auf E/A-Geräte hat. Eine Hauptaufgabe der Dom0 ist es, die Gerätetreiber bereitzustellen, da der Xen-Hypervisor keine dedizierten Gerätetreiber enthält (siehe Abbildung 3).

Xen implementiert ein sogenanntes *Split Device Driver* Modell: die DomU enthält die Front-End Treiber, bei einem Gerätezugriff werden die jeweiligen Daten in ein von Xen verwaltetes Shared-Memory-Segment geschrieben. Anschließend liest die Dom0 diese Pakete, die vom Back-End Treiber entgegengenommen wurden und leitet sie dann durch die “echten” Gerätetreiber an die physikalischen Geräte weiter [15].

Als Dom0 werden von Xen derzeit diverse Linux und BSD Distributionen unterstützt.

Virtualisierungsarten Xen erlaubt den Einsatz von unterschiedlichen Virtualisierungsarten. Die bekanntesten sind vollständige Virtualisierung und Paravirtualisierung (HVM und PV) (siehe Kapitel 4.1.2). Des Weiteren wurden auch noch zwei zusätzliche Lösungsansätze entwickelt:

- **PV on HVM**: Interrupts, Disk- und Netzwerktreiber sind paravirtualisiert, Hauptspeicher und sämtliche privilegierte Operationen laufen direkt auf der Hardware;
- **PVH**: (wird in Xen 4.4 integriert); nur die privilegierten Operationen werden vollvirtualisiert, alle anderen Hardwarezugriffe sind paravirtualisiert.

Migration Der Begriff der Migration wurde bereits in Kapitel 2.2.1 eingeführt. Xen unterstützt zwei unterschiedliche Migrationsarten:

- **Cold Migration**: Die VM wird pausiert und der Systemzustand auf einen anderen Virtualisierungsserver repliziert. Anschließend wird die Domäne auf dem neuen Host fortgesetzt.
- **Live-Migration**: Die VM wird nicht pausiert. Der Speicherzustand wird während des Migrationsprozesses zwischen den beiden beteiligten Servern synchronisiert, bis dem VM-Zustand komplett übertragen wurde. Diese Migrationsvariante hat eine Downtime, die stark von dem Speicherverhalten der migrierten Anwendung abhängt und zwischen 40 und 400 Millisekunden betragen kann [21].

Hochverfügbarkeit Seit 2009 ist *Remus* Teil des Xen-Projekts [20]. Remus bietet einen Fehlertoleranz-Mechanismus, der auf die Live-Migration basiert. Die Speicherseiten werden bis zu alle 25 Millisekunden auf einen Backup repliziert.

Bei der Replikation wird ein sogenanntes *Checkpointing Verfahren* verwendet: die Backup-VM wird nicht ausgeführt; sie ist pausiert und erhält nur die Zustansänderungen der primären VM. Bei einem Ausfall des ersten Hosts oder einem Ausfall der Master-VM wird die Backup-VM fortgesetzt und übernimmt die Aufgabe der ausgefallenen Maschine.

Bei der Verwendung von Remus können auch die TCP-Sessions erhalten werden. Dies geschieht durch die Verwendung eines Netzwerkpuffers, der bei der Replikation ebenfalls synchronisiert wird; weitere Details zu Remus finden sich im Kapitel 6.1.3.

Evaluation In den ersten Wochen der Projektgruppe wurde Xen intensiv getestet. Als Dom0 wurde dabei Ubuntu 12.10 mit zwei unterschiedlichen Xen-Versionen (4.1.4 und 4.2.1) eingesetzt.

Für den Aufbau eines echtzeitfähigen Systems war die Stabilität von Remus von zentraler Bedeutung. Aus diesem Grund wurde bei der Evaluation ein besonderes Augenmerk auf die Live-Migration und Remus-Replikation gelegt. Wegen eines Fehlers in der Checkpointing-Implementierung von Xen kam die Verwendung der Version 4.2.1 innerhalb der Projektgruppe nicht in Frage. Deshalb haben wir uns für den Einsatz der Version 4.1.4 entschieden (Details zur Installation und Konfiguration der Hostsysteme finden sich in Kapitel 6.1).

4.3. Gast-Systeme

(Sebastian Struwe) Ein Gastsystem ist ein Betriebssystem, das auf emulierter Hardware läuft, die von dem Hostsystem zur Verfügung gestellt wird.
An das Gastsystem werden verschiedene Anforderungen gestellt, so z.B. die der Echtzeitfähigkeit. Eine weitere untergeordnete Anforderung betrifft die Leistungsfähigkeit. Da eine Instanz des Gastsystems einen Webserver beherbergen soll, muss es einigermaßen leistungsfähig sein. Dabei sollte es jedoch möglichst schonend mit den Ressourcen umgehen, da mehrere Gastsysteme auf einem Host laufen sollen. Des Weiteren soll eine andere Instanz des Gastsystems die Load Management Unit (LMU) beherbergen. Die Aufgabe der LMU ist es, Aktionen mit den Tasks vorzunehmen und auf eintretende Ereignisse zu reagieren (siehe 5.4). Außerdem wäre es wünschenswert, wenn die Beispielapplikation (siehe 7) auf dem Gastsystem laufen kann.
Im Rahmen der Auswahl eines Gastsystems wurden verschiedene Betriebssysteme miteinander verglichen. Es gibt eine große Anzahl von Betriebssystemen für eingebettete Systeme, die eine Echtzeitanforderung unterstützen. Jedoch beschränken sich die meisten auf spezielle Aufgaben, wie z.B. das Ansteuern von Maschinen. Ein

Beispiel hierfür ist das Betriebssystem RTLinux.[4]
Ein Betriebssystem, das fast allen Anforderungen gerecht wird, ist TinyCore, welches jedoch nicht echtzeitfähig ist.

4.3.1. TinyCore

(Sebastian Struwe) Als Gastbetriebssystem wurde TinyCore [5] in Betracht gezogen. Das Gastbetriebssystem läuft auf den Hostrechnern und führt in einer Instanz den Webserver und in einer anderen Instanz die LMU aus. TinyCore ist ein schmales und leichtgewichtiges Betriebssystem, das eine große Verbreitung hat. Es wird in drei verschiedenen Ausstattungsvarianten angeboten, die eine Größe von 9-72 MB haben.

TinyCore beinhaltet einen Paket Manager, über den das Nachinstallieren von Software möglich ist. Es residiert komplett im Arbeitsspeicher und ist daher ausreichend leistungsfähig.

Um den Anforderungen gerecht zu werden, muss es möglich sein, einen Webserver in TinyCore zu betreiben. Die dafür benötigten Pakete lassen sich über den Paketmanager nachinstallieren.
Eine weitere Anforderung bezieht sich auf die LMU. Diese benötigt Python Unterstützung, welches sich aber auch durch den Paket Manager installieren lässt.

Die Echtzeitfähigkeit kann durch den PREEMT_RT Patch [6] realisiert werden.

Zusammenfassend kann gesagt werden, dass wir mit TinyCore ein schmales, lauffähiges Betriebssystem haben, das die meisten Anforderungen erfüllt.

4.3.2. CiAO

(Gregor Kotainy) In diesem Abschnitt geht es das Gastbetriebssystem *CiAO* (*CiAO is*

[4] http://en.wikipedia.org/wiki/RTLinux
[5] http://tinycorelinux.net/
[6] https://rt.wiki.kernel.org

Aspect-Oriented), das vor allem in eingebetteten Systemen eingesetzt wird und viele vorteilhafte Eigenschaften aufweisen kann [32]. Mit *CiAO* wurde ein Betriebssystem geschaffen, das von Grund auf durch Verwendung von Aspekten das Entwurfsprinzip *Separation of Concerns*[7] realisiert und hochgradig konfigurierbar ist. Seine Ursprünge hat *CiAO* in der *Universität Erlangen* und wird dort auch weiterentwickelt. Das Betriebssystem ist maßgeschneidert, wodurch es sehr klein ist und sich für aktuelle Forschung eignet. Der Programmierer hat volle Kontrolle über die definierten Prozesse. Einige der Entwickler sind an der *Technischen Universität Dortmund* ansprechbar, was als weiterer Vorteil gewertet werden kann. Zudem verfügt *CiAO* über einen kleinen IP-Stack, der sowohl *UDP*, als auch *TCP* versteht und Effizienz verspricht. Es sind viele Beispielanwendungen vorhanden, die es mit wenig Aufwand ermöglichen, schnell zu einem lauffähigen Betriebssystem zu kommen. Leider ist *CiAO* nicht paravirtualisiert, sodass der Umweg über *QEMU* in *XEN* genommen werden muss. Da *CiAO* ein sehr leichtgewichtiges Betriebssystem mit zunächst einem einzigen Task ist, kann eine echtzeitfähige Anwendung geschrieben werden, deren Antwortverhalten vorhergesagt werden kann. Die Vorhersagbarkeit eines minimal konfigurierten, maßgeschneiderten Systems ist erstrebenswert, da wichtige Hardware-Ressourcen eingespart und beim Einsatz mehrerer virtualisierter Maschinen optimal ausgelastet werden können.

4.3.3. Mini-OS

(Vasco Fachin) Mini-OS ist ein Betriebssystem, welches als Teil des Xen-Source Codes zur Verfügung gestellt wird. Die Idee dahinter war, ein Beispiel-Gastsystem zu haben, um die Minimalanforderungen einer paravirtualisierten virtuellen Maschine zu implementieren und beschreiben [36].

Da Mini-OS eines der kleinsten paravirtualisierten Betriebssysteme ist, wird es auch oft als sogenannte *Stub-Domain* verwendet. Eine Stub-Domain dient dazu, den sonst notwendigen *qemu*-Prozess zur Hardwareemulation für eine HVM zu ersetzen, so dass die virtuelle Maschine von der Dom0 getrennt werden kann und somit die E/A-Auslastung der Dom0 reduziert wird [15].

Das Betriebssystem bietet eine übersichtliche Schnittstelle zum Xen-Hypervisor. Die

[7] http://en.wikipedia.org/wiki/Separation_of_concerns

Hypercalls können direkt in einer Beispielapplikation ausgeführt werden. Außerdem beinhaltet Mini-OS einen paravirtualisierten Netzwerk-Treiber, der den minimalen TCP/IP Stack *lwIP* (Lightweight TCP/IP [37]) unterstützen kann.

Zusätzlich besteht die Möglichkeit, ohne großen Aufwand eine eigene Applikation zusammen mit den Xen-Bibliotheken zu kompilieren und in Mini-OS auszuführen.

Auf diesem Grund wurde auch Mini-OS als Gastsystem für die Beispielapplikation getestet. Allerdings ist eine Remus Replikation bzw. Migration von Mini-OS nach derzeitigem Stand nicht möglich, da der *Event Channel* nicht suspendiert werden kann. Der *Event Channel* wird von dem Hypervisor verwendet, um Benachrichtigungen an die VM auszuliefern (ähnlich wie *Signale* in UNIX Systemen): der Mini-OS Kernel implementiert nicht die Funktionalität, um die neuen physikalischen Speicheradressen speichern und die entsprechenden Benachrichtigungen aus dem Hypervisor annehmen zu können.

Details hierzu folgen im Kapitel 7.

4.4. Monitoring-Systeme

(Parinas Nassiri) In der Planung der Projektgruppe war sehr schnell offensichtlich, dass ein Monitoring-System benötigt wird. Ein Monitoring-System soll wichtige Daten und Werte eines Systems nach einem angegebenem Intervall ermitteln und diese veranschaulichen. Für die Projektgruppe war uns dabei wichtig, dass bestimmte Informationen, wie zum Beispiel IP, Name, Auslastung und noch weitere Daten der laufenden Hosts möglichst schnell abrufbar sind und übersichtlich dargestellt werden. Dies soll unter möglichst geringem Aufwand geschehen. Zusätzlich zu der Übersichtlichkeit und der Erleichterung der Datenerfassung soll das Monitoring auch als eine Überwachung dienen, um den Anwender bei eventuell kritischen Werten zu informieren. Zunächst musste die Entscheidung getroffen werden, ob ein selbst erstelltes Monitoring verwendet werden sollte oder ein fertiges Monitoring-System ausreicht, und falls ja, welches unseren Ansprüchen genügen könnte. Dabei ist wichtig, dass die Monitoring Komponente nicht das gesamte System stark auslastet oder gar verlangsamt. Es soll dauerhaft verfügbar sein und helfen, Fehler rechtzeitig zu erkennen und vorhandene Daten möglichst strukturiert darzustellen.

Zunächst wurden einige häufig verwendete und dadurch bekannte Monitoring Tools herausgesucht und näher betrachtet, um abzuwägen, ob diese unseren Anforderungen genügen. Es gibt eine große Anzahl an verschiedenen Monitoring-Systemen von diversen Herstellern mit unterschiedlichen Funktionen. Es wurden nur kostenlose Tools näher in Betracht gezogen, von denen hier nur auf drei eingegangen wird, um den Rahmen dieses Berichtes nicht zu sprengen.

Das erste Tool nennt sich *Nagios*[8]. Nagios kann in allen Unix-basierten Betriebssystemen eingesetzt werden. Für die Grundinstallation bedarf es aber einiger weiterer Komponenten, wie einen Apache HTTP Server mit PHP-Modul. Abgesehen von der Kernsoftware von Nagios müssen spezielle Plug-ins für bestimmte Zwecke eingesetzt werden. Diese Plug-ins übernehmen die eigentlichen Aufgaben des Monitorings. Sehr positiv ist auch die Möglichkeit eigene Plug-ins in den Sprachen Python, Perl, C, C++ und Java entwickeln zu können. Dabei gibt es einige hilfreiche Funktionen, die das fertige System anbietet, wie zum Beispiel die Alarmierung über SMS, Telefonanrufe, E-Mail. Die Einstellungen des Monitorings erfolgen über Konfigurationsdateien. Jedoch ist die Konfiguration von passiven *Netzwerküberwachungen* schwieriger. Unter passivem Monitoring versteht man ein Verfahren, dass unter Zuhilfenahme von vorhandenen Datenpaketen, den Traffic des Netzwerkes ermittelt. Bei dem aktiven Monitoring hingegen, werden neue Datenpakete erzeugt und gesendet, um die Auslastung des Netzwerkes zu ermitteln. Fehler werden bei passivem Monitoring erst entdeckt, nachdem diese schon aufgetreten sind.[9]

Als weiteres großes Monitoring Tool ist das bekannte und oft eingesetzte *Cacti*[10] zu erwähnen. Cacti kann Daten und Ergebnisse visualisieren. Der Zugriff erfolgt über eine Weboberfläche, wobei das Frontend in PHP geschrieben ist. Cacti ermöglicht den Anwendern und Benutzern eine klare Rollenverteilung. Auch hier werden Plugins benötigt, um bestimmte Funktionen einzusetzen. Wie auch bei Nagios gibt es zu erfüllende Voraussetzungen für die Nutzung, wie zum Beispiel PHP, Webserver, Datenbanken usw. Für eine Erhöhung des Funktionsumfanges müssen auch hier zusätzliche Erweiterungen integriert werden.

Zu guter Letzt soll hier noch das Monitoring Tool *Zabbix*[11] vorgestellt werden. Wie

[8] www.nagios.org
[9] http://www.slac.stanford.edu/comp/net/wan-mon/passive-vs-active.html
[10] www.cacti.net
[11] http://www.zabbix.com/

bei Cacti arbeitet auch Zabbix mit einer Weboberfläche, die in PHP geschrieben ist und MySQL, SQLite oder ähnliches zum Speichern der Daten benötigt. Zusätzlich muss ein Agent installiert werden, der Daten von den zu überwachenden Servern holt, um sie dann grafisch auf der Webseite darzustellen. Um die Funktionen des Tools zu erweitern, können externe Skripte ausgeführt werden.

Keines der Monitoring Tools entspricht genau den gestellten Anforderungen und bietet alle Funktionalitäten an, die benötigt und gewünscht werden. Um die erforderlichen Daten ermitteln zu können, müssten die Tools erweitert werden, wodurch keine große Zeit- und Aufwandsersparnis gewonnen werden würde. Es werden keine Performance-Einbußen durch unnötige Funktionen auftreten und keine unnötigen Systemvoraussetzungen, wie zum Beispiel eine Datenbank, erfüllt werden müssen. Daher fiel die Entscheidung auf eine eigene Implementierung eines Monitoring Tools, welches wir nach Bedarf entwickeln und auf Wunsch jederzeit erweitern können.

4.5. Benutzeroberfläche

(Hülya Kaplan) Zur Steuerung unseres CPS bedarf es einer Benutzeroberfläche, über die sich das System sowohl kontrollieren, als auch überwachen lässt. Die Leistungsanforderungen werden hierfür folgendermaßen aufgeteilt. Auf der einen Seite haben wir überwachende Funktionen wie beispielsweise das Anzeigen der laufenden Hosts und deren Auslastungen. Auch die Statusabfrage der VMs oder die Statusanzeige der Applikationen gehören hierzu. Auf der anderen Seite haben wir die steuernden Funktionen wie das Starten und Stoppen eines Dienstes, die Bestimmungen der Redundanz, Priorität, WCET etc.
Demnach ruft unsere GUI nicht nur kontinuierlich Daten ab, sondern konfiguriert diese auch. Dies geschieht mit der Steuerung des Hypervisors, worauf im nächsten Abschnitt näher eingegangen wird.

4.5.1. Anpassung vorhandener Software oder eigene Entwicklung

Die Steuerung von Xen verläuft über den Dienst *xend*, welcher innerhalb der Dom0 ausgeführt wird und sämtliche VMs verwaltet. Mithilfe dieses Dienstes kann der Benutzer alle virtuellen Maschinen steuern und auf deren Konsolen zugreifen. Doch

um *xend* bedienen zu können, wird das Kommandozeilenprogramm *xm* benötigt. Wird beispielsweise ein Webinterface als Schnittstelle benutzt, so kommuniziert diese mithilfe der *xm*-Befehle mit dem *xend*.

xm ist der *Xen Monitor* und stellt das Hauptprogramm für die Verwaltung von virtuellen Maschinen dar. Es dient als Werkzeug, mit dem der Benutzer interaktiv sämtliche Domains von der Kommandozeile aus steuern kann. *xm* besitzt alle Domain-Management-Funktionen und teilt *xend* die gewünschten Operationen mit, welche *xend* im Anschluss ausführt.[28]

Es wurden bereits diverse Hilfsprogramme entwickelt, mit denen sich Xen komfortabler steuern lässt. Hier werden einige davon aufgezählt.

Xen-Shell ist eine textbasierte Anwendung, die jedem Nutzer erlaubt, seine eigene VM ohne Zugriff auf die Dom0, zu verwalten. Nach der Einrichtung eines Zugangs via Login-Shell, öffnet sich bei jeder Anmeldung des Nutzers die Xen-Shell. Hierüber werden ihm nur beschränkte Rechte zugeteilt, so dass er *xm* und somit andere Programme innerhalb der Dom0 nicht starten kann. Welcher Nutzer über welche VM-Verwaltung verfügt, wird in der Konfigurationsdatei der VM gespeichert.
Von Vorteil ist, dass dadurch keinerlei Supportanfragen entstehen. Allerdings liegt der Nachteil auf der Hand: Die Benutzerfreundlichkeit leidet unter der fehlenden grafischen Bedienung.

Enomalism ist eine webbasierte Grafische Oberfläche zur Xen-Steuerung. Da sie sehr viele Anforderungsbereiche abdeckt und verschiedene Konfigurationsmöglichkeiten anbietet, wird sie meist von Unternehmen eingesetzt.
Die Erweiterbarkeit des Enomalism durch diverse Plugins ist recht vorteilhaft. Auch wenn dessen komplexe Installation viele weitere Programme erfordert, die für uns irrelevant wären, kommt diese Lösung unseren Vorstellungen am nächsten. Allerdings müssten wir dennoch zu viele Anpassungen durchführen, so dass sich dieser Ansatz doch nicht mehr lohnt.

XenMan ist ein grafisches Werkzeug, das mit Hilfe von *xm*-Befehlen die Xen-Domains steuert. Mir ihm lassen sich z.B. die VMs direkt in XenMan starten, zwi-

schenzeitig pausieren und auch anhalten. Neben den üblichen Verwaltungsmöglichkeiten bietet XenMan auch die Bedienung eines Serverpools, sprich entfernte Server, können zugefügt und ihre Domains von einer zentralen Stelle aus gemanagt werden. Im Gegensatz zu Enomalism ist die Installation des XenMan recht simpel und erfordert keine weiteren Pakete.

virt-manager Der *virtual-machine-manager*, kurz virt-manager, ist ein Desktop System mit einer ebenfalls grafischen Verwaltung. Ähnlich wie bei *XenMan* lassen sich hiermit nicht nur lokale VM bedienen sondern auch solche, die auf entfernten Systemen laufen. Der virt-manager setzt auf eine Programmbibliothek namens *libvirt*. Diese bietet dabei die Schnittstelle zu den Virtualisierungsfunktionen des Betriebssystems. Das heißt, es wird eine Zwischenschicht gebildet, um die Verwaltung der VMs, unabhängig von Xen, einheitlich zu gestalten. Dies wiederum bedeutet weniger administrativen Aufwand.

Eine eigene Implementierung scheint an dieser Stelle von Vorteil zu sein, da die erwähnten Beispiele zwar viele praktische Möglichkeiten bieten, aber die meisten davon für uns irrelevante sind, wie zum Beispiel die separate Verwaltung verschiedener Benutzeranmeldungen. So scheint es, dass keines der fertigen Lösung genau auf unsere Erwartungen und Anforderungen angepasst ist. Hierzu gehört z.B. die parallele Ausführung von Master+Backup+Node-Diensten.
Um die optimale Lösung für unser System zu erreichen, muss unsere Benutzeroberfläche so spezifisch wie möglich auf unsere Anwendung zugeschnitten sein. Daher wollten wir, inspiriert durch bereits vorhandene Ansätze, eine neue grafische Schnittstelle kreieren. Hierfür soll unser Programm primär die oben aufgelisteten Anforderungen abdecken. Wegen der Übersicht über zahlreiche Hosts und deren Möglichkeiten zur grafischen Darstellung der im Monitoring ermittelten Werte, mittels Diagrammen, sollte es eine grafische Oberfläche haben. Gleichzeitig sollte es auch eine schnelle Steuerung der VMs oder Hosts ermöglichen, wie beispielsweise das Einsehen von Fehlerprotokollen mit nur einem Klick.

4.5.2. Webbasierte Anwendung oder Desktop-Anwendung

Wir entschieden uns also für eine eigene Implementierung und überlegten welche Vor- und Nachteile eine webbasierte GUI gegenüber einer Desktop-Anwendung hat. Beim letzteren ist die Implementierungssprache beliebig wählbar, wohingegen wir bei einer Web-Gui auf einige wenige aber dafür etablierte Standards zurückgreifen können. Zudem ist eine HTML-GUI schnell implementiert und es lassen sich fertige Webserver-Software leicht einbetten, die wir beispielsweise für die Diagramme benötigen. Außerdem ist eine Web-GUI plattformunabhänig, was ein nicht ausschlaggebender aber dafür praktischer Nebeneffekt ist, und bieten einen globalen Zugriff. Gerade für eine PG, bestehend aus mehreren Mitgliedern, welche an verschiedenen Orten und Rechnern ständigen Zugang zur GUI brauchen, wird diese Flexibilität definitiv begrüßt. Daher entschieden wir uns, eine webbasierte GUI zu verwenden. Zwar benötigen wir dafür einen Webserver und sind stets auf einen Browser angewiesen, aber im Gegenzug wird unsere Arbeit um einiges erleichtert.

4.6. Lokales Scheduling

(Tim Harde, Hülya Kaplan) Als *lokales Scheduling* bezeichnen wir im Rahmen der Projektgruppe das Schedulingverfahren, das auf den einzelnen Virtualisierungsservern zur Zuteilung der CPU-Zeit an die einzelnen virtuellen Maschinen eingesetzt wird.

Für das Xen-Framework existieren unterschiedliche Verfahren, um die Virtualisierungsserver an die Anforderungen des jeweiligen Einsatzgebietes anzupassen. Während bei einem Einsatz im Rechenzentrum zur Virtualisierung von Serverkomponenten wie beispielsweise Datenbankservern, Mailservern oder VoIP-Servern eher auf die allgemeine Performance ankommt, sind solche Schedulingverfahren für die Virtualisierung von CPS häufig nicht geeignet, da sich meist keine Echtzeitgarantien für die einzelnen virtuellen Maschinen abgeben lassen.

In den folgenden zwei Unterkapiteln sollen zunächst die beiden standardmäßig in XEN integrierten Schedulingverfahren (SEDF und Credit2) vorgestellt werden, anschließend erfolgt eine kurze Vorstellung von RT-XEN.

4.6.1. Vorinstallierte Scheduler

(Hülya Kaplan) In Xen wird jeder VM eine virtuelle CPU, im Folgenden VCPU genannt, zugeteilt. Diese werden vom Gast-System als gewöhnliche CPUs erkannt. Für die Zuteilung der VCPUs auf echte CPUs wird ein Scheduling-Verfahren benötigt. In der Xen-Architektur läuft der Scheduler im Hypervisor, und bietet für verschiedene Scheduling-Verfahren ein Scheduling-Interface. Von dort aus wechselt er je nach Scheduling-Prinzip zwischen den VCPUs.

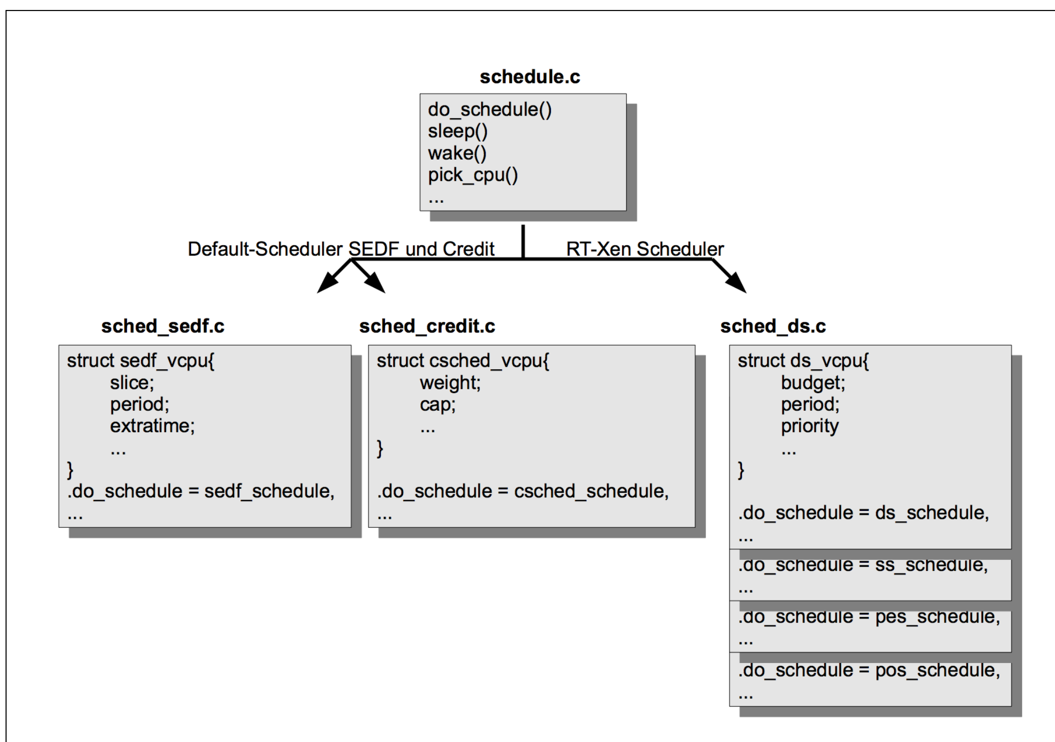
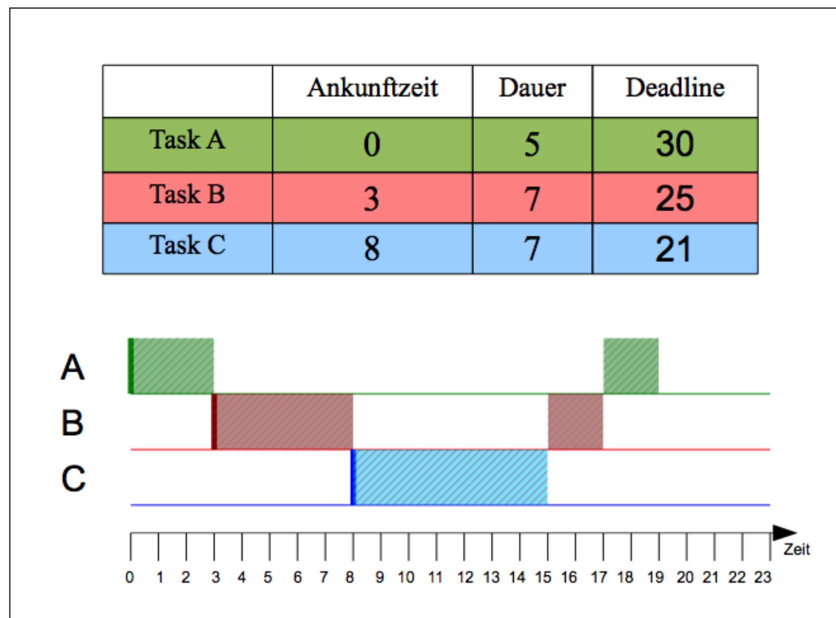


Abbildung 4: Scheduling Interface

Eine Datei namens *schedule.c* bildet dabei das Framework für den Scheduler und beinhaltet bereits einige gängige Funktionen. Soll ein bestimmter Scheduling-Algorithmus erstellt werden, muss zunächst eine *subscheduler* Datei angelegt werden, in der diese Funktionen definiert sind. Die zwei folgenden Methoden sind im Xen standardmäßig enthalten:

SEDF ist das *Simple-Earliest-Deadline-First*-Prinzip und war bis zur Version 3.0 der Default-Scheduler von Xen. Es basiert auf dem *Earliest-Deadline-First* (EDF),

welches dynamisch läuft und seine Entscheidungen erst zur Laufzeit trifft. Da die Prioritäten sich deswegen oft ändern, kann im EDF die Ausführung eines Tasks gegebenenfalls unterbrochen werden, um anderen Tasks den Vortritt zu gewähren (präemptives Scheduling). Dabei erhält jedes Mal die Task mit der nächsten Deadline den Vorrang, wie es die folgende Abbildung 5 zeigt.



	Ankunftzeit	Dauer	Deadline
Task A	0	5	30
Task B	3	7	25
Task C	8	7	21

Abbildung 5: EDF Schedule

Das SEDF arbeitet anstelle von Tasks mit Domains, genauer gesagt mit ihren VC-PUs, welche nach Deadlines sortiert werden. Die VCPU mit der nächsten Deadline, bekommt den Prozessor zugeteilt.

Bei diesem Scheduling-Verfahren erhält jede VCPU drei Parameter: *slice*, *period* und *extratime*. *slice* entspricht einem Guthaben, das die VCPU besitzt und verbraucht, so lange sie läuft. Falls sie nicht läuft, wird ihr *slice* beibehalten und beim Eintritt in die nächste Periode wieder aufgefüllt. Die *extratime* ermöglicht der VCPU auch nach Ablauf seiner *Slice* noch weiter zu laufen. Zu jedem echten CPU-Kern gehört eine Running-Queue (*RUNQ*) in der alle VCPUs mit positiver *slice* nach ihrer Deadline einsortiert sind. Ihre Deadline lässt sich anhand ihrer *period* errechnen.

CREDIT ersetzt das SEDF als Default-Scheduler im Xen. Auch hier hat jeder Kern eine RunQ, in der die ausführbaren VCPUs warten. Zu Beginn wird in jedem Kern

eine IDLE VCPU erstellt. Dies ist eine Art Pseudo-CPU, die stets ausführbereit am Ende der Schlange wartet. Wenn beim Scheduling die IDLE-VCPU drankommt, sind keine weiteren ausführbaren VCPUs vorhanden und der Kern gerät in einen Leerlauf.
Jede Domain verfügt über die Parameter *weight* und *cap*. *weight* stellt die Gewichtung der Ausführungszeit dar und *cap* dessen obere Schranke. Entsprechend seiner Gewichtung, wird jeder Domäne ein *credit* vergeben, welches sie an seine VCPUs verteilt (analog zu *slice* beim SEDF). Solange sie ausgeführt werden, verbrauchen die VCPUs diese *credits*.
In der RunQ werden die VCPUs in drei Kategorien unterteilt. Führen sie I/O-Operationen aus, gehören sie zu *boost*. Falls ihre *credits* aufgebraucht sind, zu *under*. Und solche die noch *credits* über haben, zu *over*. Das anschließende Scheduling läuft nach dem Round-Robin-Schema ab.
Eine Besonderheit der *Credit*-Methode ist allerdings, dass die vorgegebene Scheduling-Periode bei 30ms liegt. Nach dieser Zeit wird die gerade laufende VCPU abgebrochen und durch eine andere abgelöst. Diese harte Fairnis kann leider auch von Nachteil sein. Außerdem wird eine VCPU mit aufgebrauchten *credits* nicht mehr berücksichtigt, auch wenn sie noch weitere I/O-Aufgaben hätte.

4.6.2. RT-XEN

(Tim Harde) RT-Xen ist ein Scheduler für Xen, der derzeit (Stand: September 2013) in der Version 1.0 vorliegt und auf Xen 4.1.4 basiert. Ziel bei der Entwicklung von RT-Xen war es, den Xen-Hypervisor echtzeitfähig zu machen und somit die fehlende Unterstützung für virtuelle Maschinen mit Echtzeitanforderungen bereitzustellen.

RT-Xen erweitert im Prinzip SEDF (eine globale Prioritätswarteschlange für jeden physikalische Kern mit dynamischen Prioritäten). Der RT-Xen-Scheduler verwaltet die unterschiedliche VCPUs; jede VCPU verfügt dabei über Parameter: Priorität, Budget und Periode.

Die *Periode* entspricht dabei genau der Anzahl an Zeiteinheiten, die zwischen zwei Ausführungen eines Tasks liegt. Die *Priorität* wird statisch vergeben, das *Budget* ist die Rechenzeit, die einer VCPU während einer einzelnen Periode zugesichert wird; während der Ausführung wird dabei das Budget immer weiter verringert.

RT-Xen verwendet intern drei unterschiedliche Warteschlangen für jede physikalische CPU (siehe Abbildung 6):

- **RunQueue** (RunQ): In dieser Warteschlange befinden sich alle VCPUs, die derzeit über einen ausführungsbereiten Task verfügen. Dabei ist irrelevant, ob die jeweilige VCPU noch über ein Budget zur Ausführung verfügt. Die RunQ ist eine Prioritätswarteschlange, die nach den Prioritäten der VCPUs sortiert ist.
- **ReadyQueue** (RdyQ): In dieser Warteschlange befinden sich alle VCPUs, die derzeit keinen ausführungsbereiten Task haben.
- **ReplenishmentQueue** (RepQ): In dieser Queue werden Informationen verwaltet, die für das Auffüllen des Budgets für die unterschiedlichen VCPUs benötigt werden. Die RepQ ist dabei nach dem Zeitpunkt der nächsten Auffüllung des Budgets sortiert.

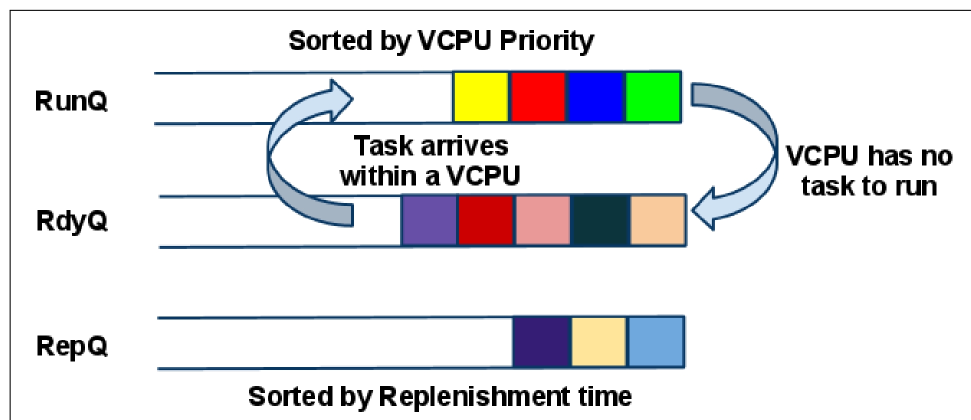


Abbildung 6: Wartenschlangen bei RT-Xen (Quelle: [3])

Die einzelnen VCPUs wechseln dabei von der RdyQ in die RunQ, sobald ein neuer Task in der zugeordneten Domäne (also ein Prozess innerhalb der virtuellen Maschine, der die VCPU zugeordnet ist) ausführbereit wird. Die Gegenrichtung (von der RunQ in die RdyQ) erfolgt dementsprechend genau dann, wenn ein Task komplett abgearbeitet wurde und kein weiterer Task auf die Ausführung wartet.

Insgesamt unterscheidet RT-Xen zwischen vier verschiedenen Servertypen. Diese Servertypen klassifizieren dabei im Prinzip das Verhalten einer virtuellen Maschine.

- **Deferrable Server** (DS, siehe Abbildung 7): Dieser Servertyp hat eine feste Periode. Die Ausführung erfolgt dabei solange, bis entweder das Budget erschöpft ist oder der Task erfolgreich abgearbeitet wurde. Eventuell verbleibendes Budget bleibt dabei auch nach der Ausführung erhalten und wird mit Beginn der nächsten Periode wieder aufgefüllt.

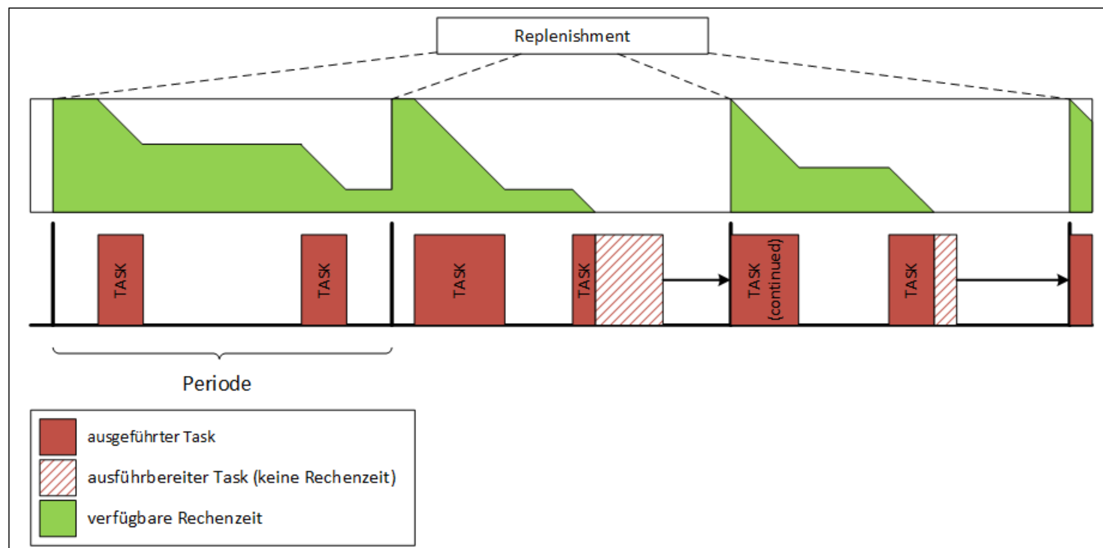


Abbildung 7: Beispielhaftes Scheduling mit RT-Xen: Deferrable Server.

- **Periodic Server** (PES, siehe Abbildung 8): Dieser Servertyp hat ebenfalls eine feste Periode. Auch hier erfolgt die Ausführung solange, bis entweder das Budget erschöpft ist oder der Task erfolgreich abgearbeitet wurde. Im Gegensatz zu DS verbraucht dieser Servertyp aber auch dann das eigene Budget, wenn kein Task ausführungsbereit ist. Dies wird durch einen sogenannten Idle-Task realisiert, der stellvertretend eventuell vorhandenes Budget verbraucht. Dies sorgt dafür, dass der Virtualisierungsserver nicht durch häufige und kurze Berechnungen der virtuellen Maschine (durch hiervon verursachte häufige Kontextwechsel und dem daraus resultierenden Overhead) belastet wird.
- **Polling Server** (POS, siehe Abbildung 9): Dieser Servertyp ist eine Spezialform des PES. Der einzige Unterschied liegt darin, dass nach der Ausführung eines Tasks eventuell vorhandenes Budget komplett und sofort verworfen wird.
- **Sporadic Server** (SS): Dieser Servertyp verfügt über keine feste Periode, stattdessen erfolgt die Ausführung sporadisch und unvorhersehbar. Für die

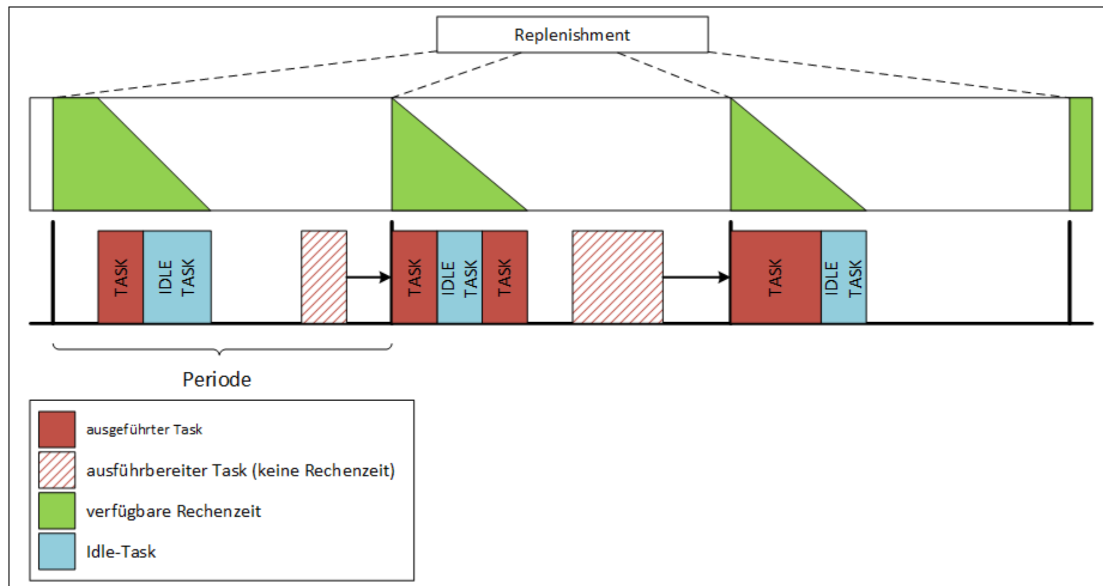


Abbildung 8: Beispielhaftes Scheduling mit RT-Xen: Periodic Server.

Auffüllung des Budgets existieren unterschiedliche und teilweise komplizierte Regeln. Da dieser Servertyp für die hier betrachteten Beispielapplikationen nicht relevant ist, wird an dieser Stelle auf weitere Erläuterungen verzichtet.

Insgesamt verfolgt RT-Xen einige sehr vielversprechende Ansätze und erfüllt die wesentlichen Anforderungen, die beim Einsatz in der Domäne der Cyber-Physikalischen Systeme an ein lokales Scheduling-Verfahren gestellt werden. Während der Evaluation konnte allerdings kein stabiles Systemverhalten mit mehr als zwei Domänen (Fedora 13, Xen 4.0,1, RT-Xen 1.0) beobachtet werden. RT-Xen kommt aus diesem Grunde für den Einsatz innerhalb der Projektgruppe nicht in Frage.

4.7. Globales Scheduling

4.7.1. Vergleich von Lastverteilungsalgorithmen

(Daniel Stoller) Das globale Scheduling hat die Aufgabe, die auszuführenden virtuellen Maschinen auf die einzelnen Hosts zu verteilen und somit jedem Host eine bestimmte Menge von virtuellen Maschinen zuzuordnen. Die Lastverteilung, die von einem entsprechenden Algorithmus berechnet werden soll, muss dabei insbesondere zwei Anforderungen erfüllen. Die erste Anforderung ist, dass auf jedem Host die Aus-

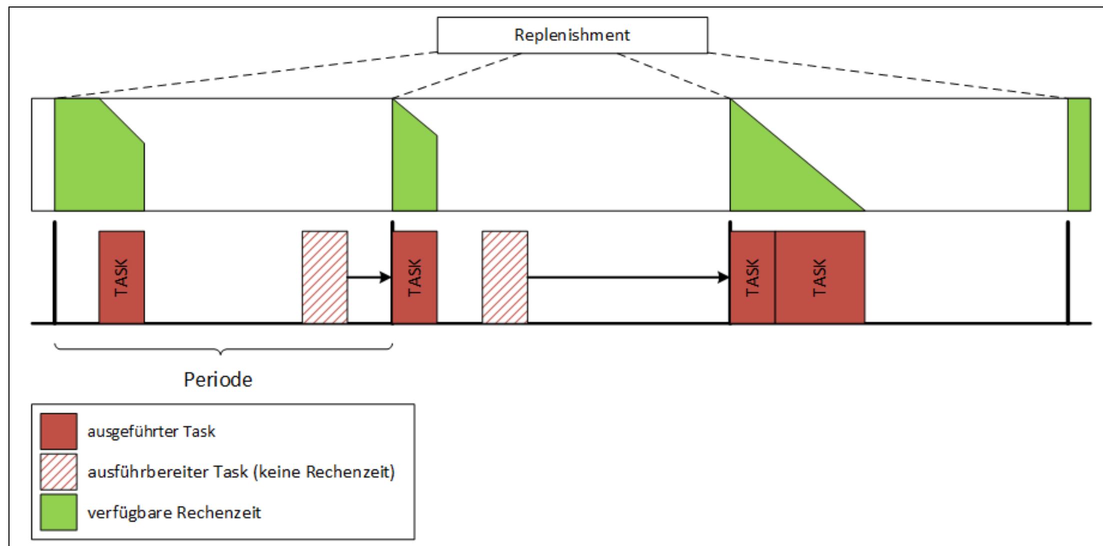


Abbildung 9: Beispielhaftes Scheduling mit RT-Xen: Polling Server.

lastung, also die Summe der Lastwerte aller auf diesem Host befindlichen virtuellen Maschinen kleiner ist als ein bestimmter Maximal-Lastwert $c \in [0, 1]$. Für jede virtuelle Maschine ergibt sich der Lastwert aus dem Quotienten von Worst-Case Laufzeit und der Periode des Dienstes. Die zweite Anforderung stellt die Fehlertoleranz sicher, indem redundant auszuführende virtuelle Maschinen eines Dienstes auf unterschiedlichen Hosts ausgeführt werden, das heißt keine zwei virtuellen Maschinen des selben Dienstes existieren, die dem selben Host zugeordnet sind. Andernfalls würde nämlich bei Ausfall eines Hosts auch die zur Absicherung zusätzlich ausgeführte virtuelle Backup-Maschine ausfallen können, womit die Fehlertoleranz nicht mehr gewährleistet wäre. Abgesehen von den obigen beiden Anforderungen gibt es somit einen Spielraum, in dem verschiedene Algorithmen mit unterschiedlichen Vor- und Nachteilen für den Einsatz innerhalb der Projektgruppe in Frage kommen. Daher werden im Folgenden drei verschiedene Ansätze zur Lastverteilung vorgestellt.

Worst-Fit Der Worst-Fit Algorithmus verteilt die n virtuellen Maschinen eines neu zu startenden Dienstes, indem aus insgesamt h Hosts die n am wenigsten ausgelasteten Hosts bestimmt werden. Anschließend wird für jeden dieser am wenigsten ausgelasteten Hosts überprüft, ob die Summe aus aktueller Auslastung und des Lastwerts der virtuellen Maschine den Maximal-Lastwert c übersteigen würde. Ist diese Überschreitung bei mindestens einem Host gegeben, so kann der Dienst mo-

mentan nicht gestartet werden, da mindestens ein Host überlastet werden würde. Der Dienst kann auch dann nicht gestartet werden, falls $n > h$ ist, also die Anzahl zu startender virtueller Maschinen größer als die Anzahl insgesamt zur Verfügung stehender Hosts ist. Gilt keine der oben genannten Bedingungen, werden diesen n am wenigsten ausgelasteten Hosts die virtuellen Maschinen in ihrer entsprechenden Redundanz zugeteilt.

Diese Vorgehensweise führt zu einer tendenziell gleichmäßigen Auslastung der verschiedenen Hosts, indem er die Auslastungsdifferenzen zwischen den Hosts minimiert. Die amortisierte Laufzeit ist bei Verwendung eines Fibonacci-Heaps für die Hosts mit ihren Auslastungen durch $\mathcal{O}(n \log h)$ beschränkt: Im schlimmsten Fall muss für jede der n virtuellen Maschinen jeweils einmal der aktuell am wenigsten ausgelastete von insgesamt h Hosts aus dem Heap extrahiert und am Ende wieder eingefügt werden. Da ein Fibonacci-Heap für die Extraktion des Minimums amortisiert $\mathcal{O}(\log h)$ Zeit und für das anschließende Einfügen amortisiert $\mathcal{O}(1)$ Zeit benötigt, ergibt sich für diese Vorgehensweise insgeamt $\mathcal{O}(n(\log h + 1)) = \mathcal{O}(n \log h)$. Dabei kann schon vorher abgebrochen werden, falls bei einem Host eine Überschreitung des Maximal-Lastwerts festgestellt wird, da darauffolgende Hosts nur noch stärker ausgelastet sind.

Best-Fit Der Best-Fit Algorithmus versucht im Gegensatz zum Worst-Fit Algorithmus, die am meisten ausgelasteten Hosts auszuwählen. Dazu wird zunächst der am stärksten ausgelastete Host bestimmt, bei dem keine Überschreitung des Maximal-Lastwerts auftritt. Falls dieser Host existiert, werden die $n - 1$ am nächst stärksten ausgelasteten Hosts bestimmt. Sind nun insgesamt n Hosts gefunden worden, wird ihnen jeweils eine virtuelle Maschine zugeteilt, andernfalls gibt es keine gültige Zuordnung.

Der Vorteil dieser Methode ist der namensgebende geringste Verschnitt, der dabei entsteht, wodurch die Hosts möglichst effizient ausgelastet werden. Außerdem gibt es bei Ausfall eines Hosts bei geringerer Auslastung eine einfache Lösung für das Umverteilungsproblem, da oft ein oder mehrere Hosts komplett unbenutzt gelassen werden und somit alle virtuellen Maschinen darauf umverteilt werden können. Der Algorithmus ist mit einer durch $\mathcal{O}(h \log h)$ beschränkten Laufzeit etwas größer als das Worst-Fit-Verfahren, da im schlimmsten Fall, bei dem alle Hosts zu stark aus-

gelastet sind, alle Hosts einmal aus dem Fibonacci-Heap extrahiert werden müssen, bevor die Unmöglichkeit einer Verteilung erkannt wird.

Integer Linear Programming Beim Integer Linear Programming werden Probleme als ein System aus einer Zielfunktion und einer Menge von Nebenbedingungen formuliert. Den Belegungen der gesuchten Variablen werden von dieser Zielfunktion einen Wert zugewiesen. Dabei wird die Belegung gesucht, welche den größten Zielfunktionswert erreicht, aber trotzdem alle geforderten als Gleichung formulierten Nebenbedingungen erfüllt. Im konkreten Fall müsste für jeden Host eine Nebenbedingung formuliert werden, welche die Summe der Auslastung der darauf laufenden virtuellen Maschinen auf den Maximal-Lastwert begrenzt.

Beim globalen Scheduling kann dieses Verfahren eingesetzt werden, indem die beiden oben genannten Anforderungen als eine Menge von Nebenbedingungen formuliert werden. Mithilfe der Zielfunktion kann dann zusätzlich und optional bestimmt werden, welche Lösungen bevorzugt werden sollen.

Dadurch wäre auch eine Simulation der Worst-Fit und Best-Fit Algorithmen im Falle eines neu zu startenden Dienstes möglich. Die Wahl beliebiger Zielfunktionen macht diesen Ansatz äußerst flexibel. Bei korrekter Definition der Nebenbedingungen wird garantiert, dass eine Lösung gefunden wird, falls es eine gibt, im Gegensatz zu den anderen vorgestellten Algorithmen. Diese Lösung ist dann hinsichtlich der Zielfunktion sogar optimal. Auf der anderen Seite ergeben sich durch die NP-Schwierigkeit von ganzzahligen Optimierungsproblemen bei steigender Komplexität der gegebenen Host-Situation schnell sehr viel größere Laufzeiten als bei den Worst-Fit und Best-Fit Algorithmen.

4.7.2. Real-Time-Calculus

(Vasco Fachin) In den letzten Jahren war der Entwicklungsprozess von eingebetteten Systemen durch hohen "time-to-market" Druck geprägt. Von Anfang des Designprozesses an benötigt man daher die Unterstützung von Methoden zur Performanzanalyse [7]. Dabei werden zwei Ansätze häufig eingesetzt, um solche Systeme zu analysieren: *Simulation* und *formale Analyse* (siehe Abbildung 10).

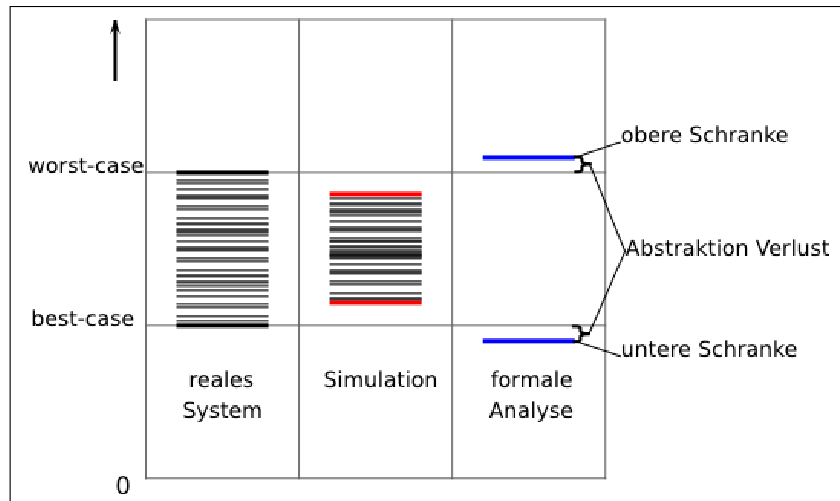


Abbildung 10: Implementierung, Simulation und Formale Analyse. Die Abbildung zeigt die unterschiedlichen WCET - BCET Berechnungen durch die unterschiedlichen Evaluationsmethoden. (Selbstbearbeitung nach [10])

Simulation und Formale Performanzanalyse Bei der Simulation werden Tools wie System C, VHDL oder Matlab/Simulink eingesetzt, um ein System zu modellieren. Durch diese Werkzeuge kann ein System mit unterschiedlichen Abstraktions- und Modularitätsebenen repräsentiert werden, um eine Performanzanalyse durchzuführen. Außerdem werden diese Hilfsmittel immer mehr in der Industrie eingesetzt, wodurch zusätzliche Bibliotheken entwickelt werden und somit viele wiederverwendbare Komponenten zur Verfügung stehen [7].

Problematisch ist die Tatsache, dass der Simulationsprozess möglicherweise eine hohe Berechnungskomplexität besitzt. Zusätzlich ist die Simulation eine Trace-basierende Methode. Das bedeutet, dass zur Bestimmung der Worst Case Execution Time (kurz: WCET) bzw. Best Case Execution Time (kurz: BCET) die Eingabewerte benötigt werden, die das Worst-, bzw. Best-Case-Szenario repräsentieren. Diese kennt man i.d.R. nicht [8].

Eine formale Analyse verwendet stattdessen ein mathematisches Werkzeug, d.h. eine formale Methode, die eine höhere Abstraktionsebene als eine Simulation darstellt. Der Vorteil ist, dass man so harte oder möglicherweise sogar exakte Betriebsschranken bestimmen kann [9]. Des Weiteren kann die höhere Abstraktionsebene die Systemdarstellung vereinfachen, was jedoch auch zu einer gewissen Ungenauigkeit führt:

Die durch das Modell berechneten Schranken liegen eventuell nicht nahe genug an den tatsächlichen (realen) Schranken. Dieses Phänomen wird auch als *abstraction loss* (Abstraktionsverlust) bezeichnet und ist in Abbildung 10 zu erkennen. Ein weiterer großer Nachteil ist, dass die Systemmodellierung nicht automatisiert werden kann. Das System muss von Grund auf modelliert und implementiert werden, was ein enormes Spektrum an Fehlern zulässt [10].

Real-Time Calculus RTC ist eine formale Methode, die im Zusammenhang mit *Modular Performance Analysis* zur formalen Performanzanalyse verwendet wird. Er basiert auf dem *Network Calculus* [8] und wurde von Chakraborty et al. in [11] definiert. RTC ermöglicht beispielsweise die Berechnung von unteren und oberen Schranken (wie beispielsweise BCET und WCET) von Ausführungszeiten, end-to-end Delays oder Puffergrößen.

Grundlage des RTC ist die Modellierung von Strömen; diese Ströme repräsentieren dabei potenziell eintretende Ereignisse und verfügbare Leistung. Ein Ereignisstrom wird durch die Funktion $R(t)$ repräsentiert und ist definiert als die Gesamtanzahl von Ereignissen im Zeitintervall $[0, t)$. Der Leistungsstrom wird auf ähnliche Art und Weise definiert: die Funktion $C(t)$ repräsentiert die verfügbare Leistung im Zeitintervall $[0, t)$.

Die Funktionen $R(t)$ und $C(t)$ beschreiben konkrete Ströme, aber zur Analyse verwendet man eine höhere Abstraktionsebene. Aus diesem Grund betrachtet man sogenannte *Arrival-* und *Service-Kurven* anstelle der oben definierten Funktionen.

Das Tupel $\alpha(\Delta) = [\alpha^u(\Delta), \alpha^l(\Delta)]$ bezeichnet jeden möglichen Ereignisstrom im Zeitintervall Δ, mit oberer Schranke $\alpha^u(\Delta)$ und unterer Schranke $\alpha^l(\Delta)$. Analog bezeichnet $\beta(\Delta) = [\beta^u(\Delta), \beta^l(\Delta)]$ die verfügbare Leistung im Zeitintervall Δ.

Legt man diese Abstraktion zu Grunde, so gilt:

- für $\alpha(\Delta)$ gibt es maximal α^u und mindestens α^l eingehenden Ereignisse;
- für $\beta(\Delta)$ gibt es maximal β^u und mindestens β^l verfügbare Leistung.

Durch diese Repräsentation kann nun eine Komponente analysiert und die verbleibende Leistung beziehungsweise der ausgehende Ereignisstrom berechnet werden. Bei der analysierten Komponente kann es sich dabei beispielsweise um eine CPU

handeln (siehe Abbildung 11).

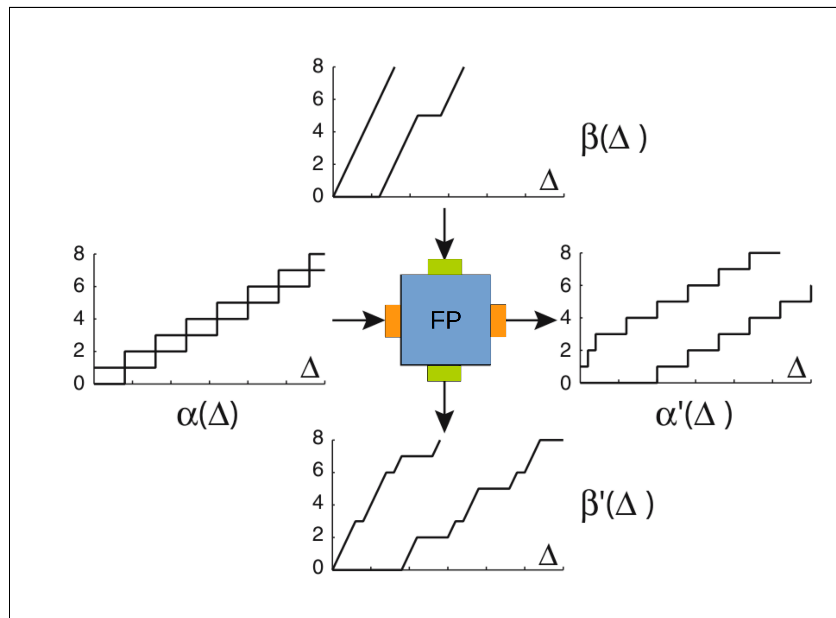


Abbildung 11: Darstellung einer Komponente nach [7]

Die ausgehenden Kurven hängen von der internen Semantik der Komponente ab. Das bedeutet, dass eine Funktion das konkrete Verhalten beschreiben kann. Sei $\alpha'(\Delta)$ der ausgehende Ereignisstrom und $\beta'(\Delta)$ die verbleibende Kapazität der Ressource, dann gilt:

- $\alpha' = f_\alpha(\alpha, \beta)$;
- $\beta' = f_\beta(\alpha, \beta)$.

Analyse Durch die Berechnung der ausgehenden Kurven kann man nun das Verhalten des Systems beschreiben und analysieren. Um komplexere Systeme zu modellieren, muss man lediglich die korrekte Verarbeitungsreihenfolge der Komponenten bestimmen und die ausgehenden Ströme, der jeweils vorangegangenen Komponente als eingehenden Strom der nachfolgenden Komponente modellieren.

Verzögerungen und Puffergrößen können ebenfalls durch die Verwendung des RTC berechnet werden. Die Berechnungsfunktion hängt auch hier von der eigentlichen Semantik ab. Abbildung 12 zeigt eine graphische Darstellung dieser Anwendung.

Mit RTC kann die in der Projektgruppe verwendete Architektur der Virtualisie-

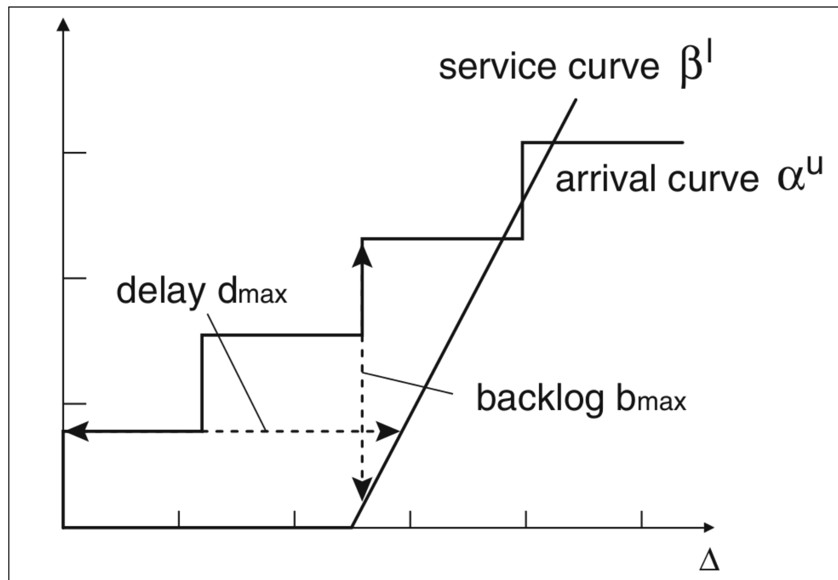


Abbildung 12: Delay und Puffergröße (*backlog*). Darstellung nach [7]

rungsplattform modelliert werden. Die Komponenten repräsentieren dabei die einzelnen VMs, die auf einem Rechner ausgeführt werden. Die Kurven repräsentieren einerseits die Tasks, die berechnet werden müssen sowie andererseits die Leistung, die für die einzelnen Rechner zur Verfügung steht. Durch die Wahl der Scheduling-Strategie (z.B. *fixed-priority* oder *earliest deadline first*) kann man nun das end-to-end delay bestimmen und dann die Echtzeit-Garantien des Systems verifizieren (siehe Abbildung 13).

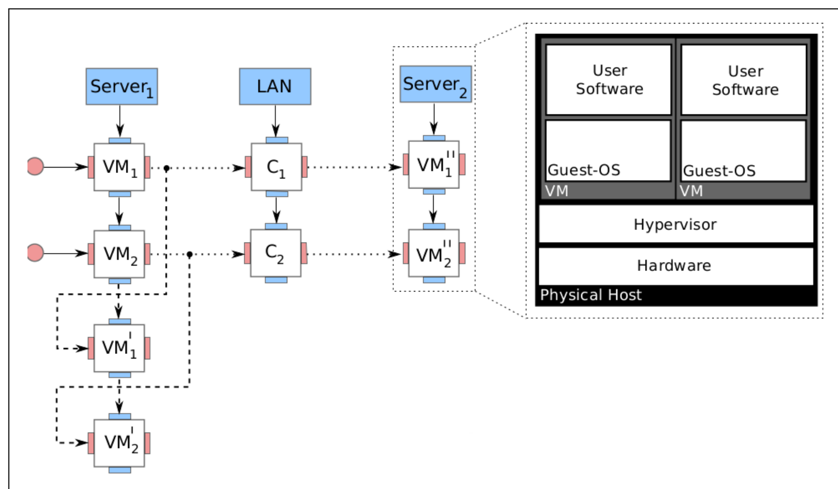


Abbildung 13: Verteiltes System in RTC nach [12]

4.8. Implementierungs-Sprachen

(Parinas Nassiri) Für die Implementierung des Systems musste eine geeignete Programmiersprache gefunden werden. Die Anforderungen an die Programmiersprache waren unter anderem, dass sie leicht erlernbar und gut verständlich sein sollte. Zudem sind Strukturiertheit und umfangreiche Bibliotheken erwünscht. Daher kamen gängige Sprachen wie Python, Java, C und C++ in die engere Auswahl, welche aufgrund von Bekanntheit der Sprachen, deren großem Funktionsumfang und persönlicher Präferenzen getroffen wurde.

Die Sprache C ist eine komplexe und imperative Programmiersprache, die sehr systemnah ist. Sie erlaubt zwar die Erzeugung von schnellen Code, bedarf aber einer Kompilierung vor jeder Ausführung. Zudem verfügt sie über keine automatische Speicherverwaltung.
C++ basiert auf der Sprache C und erweitert diese um Konzepte der Objektorientierung wie Klassen, Vererbung und Polymorphie. Die Programmiersprache bietet im wesentlichen die gleichen Vor- und Nachteile wie C und ermöglicht durch smart pointer die automatische Speicherverwaltung.
Java ist ebenfalls eine objektorientierte Sprache, die kompiliert werden muss. Da bei der Kompilierung nur Bytecode erzeugt wird, der anschließend in einer VM ausgeführt oder just-in-time in Maschinencode übersetzt wird, sind Java Programme langsamer als Programme, die in den zuvor genannten Sprachen geschrieben wurden.
Python ist von den oben genannten Sprachen die einzige Skript-Sprache, die keine Kompilierung erfordert. Sie bietet dynamische Typisierung und Objektorientierung an. Python wird häufig für Rapid-Prototyping eingesetzt. Sowohl Python als auch Java verfügen über einen Garbage Collector, welcher die Speicherverwaltung stark vereinfacht.

Wir haben uns aus mehreren Gründen für Python entschieden. Da die Teilnehmer der PG unterschiedliche Programmierkenntnisse hatten, war es für uns vorteilhaft, dass Python von den oben genannten Sprachen am leichtesten zu erlernen ist. Ein weiterer großer Vorteil ist die Möglichkeit der schnellen Erstellung und einfachen

Wartung von Programmen. Insbesondere ist durch den Garbage Collector kein expliziter Code für die Speicherverwaltung notwendig.

Die Einfachheit des Codes zeigt sich auch darin, dass in der Regel in Python geschriebene Programme kürzer und übersichtlicher sind, als äquivalente Programme, die in C, C++ oder Java geschrieben wurden. [43]. Somit ist gewährleistet, dass das Augenmerk auf Ziele und Funktionen der geschriebenen Skripte gelegt wird als auf Syntax und Umgang mit der Sprache.

> Dies hat mehrere Konsequenzen: Die Programmierung in einer Skriptsprache ist damit in Normalfall schneller erledigt. Auch die Wartung eines Programms in einer Skriptsprache ist in Normalfall effizienter.

[30]

Ein weiterer wichtiger Punkt für die Entscheidung für Python war, dass ein signifikanter Anteil von XEN und REMUS in Python implementiert wurde. Dadurch ist eine homogene Plattform gegeben und für die weitere Entwicklung des Systems sind keine Kenntnisse einer weiteren Sprache notwendig.
Auch die Anforderung einer umfangreichen Bibliothek ist bei Python erfüllt. Zugunsten eines stabilen und korrekt arbeitenden Systems wurde die geringe Ausführungsgeschwindigkeit in Kauf genommen.

Auf die im Frontend verwendeten Implementierungssprachen wird in Kapitel 6.2 näher eingegangen.

5. Systementwurf

5.1. Überblick

(Dominic Wirkner) Auf Basis der Anforderungsanalyse und der Bewertung vorhandener Technologien entwickelte die Projektgruppe ein Konzept, welches bereits sehr genau den logischen und physikalischen Aufbau des endgültigen Systems umfasst. In einigen Punkten wurde die Entscheidung bezüglich der einzusetzenden Technologien bewusst offen gelassen. Es ist jedoch zu erkennen, dass die Verwendung von XEN als Virtualisierungstechnologie bereits berücksichtigt wurde.

Im Folgenden werden nun die Einzelheiten des Konzeptes erörtert.

5.1.1. Logischer Aufbau

Der logische Aufbau beschreibt die Komponenten des Systems und deren Kommunikation untereinander (siehe Abbildung 14).

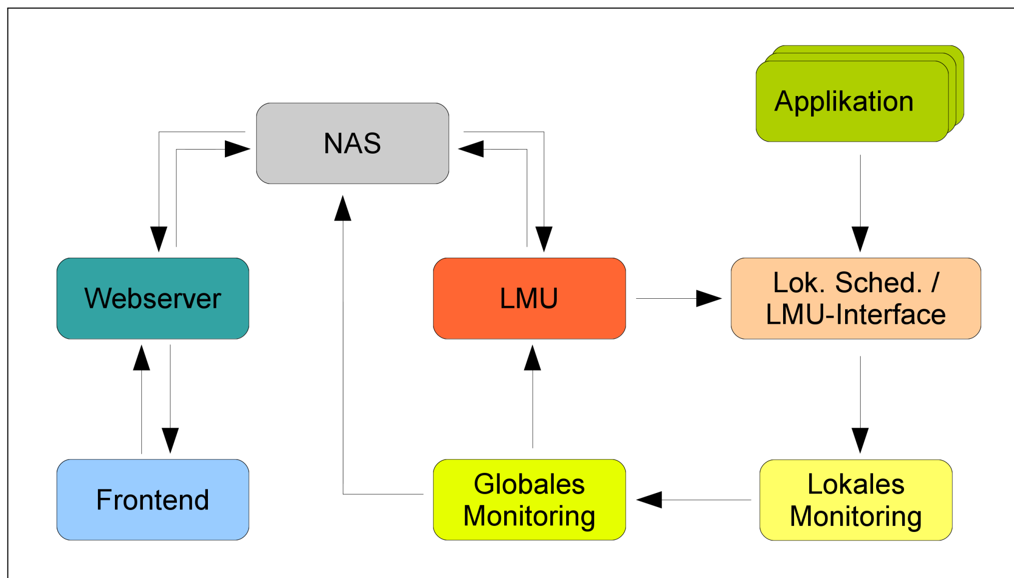


Abbildung 14: Logischer Aufbau

Netzwerkspeicher (NAS) Bereits in einer frühen Phase der Konzepterstellung wurde der Vorteil eines zentralen Speichers für Dateien deutlich: Zum einen kann auf diesem der Programm-Code für die einzelnen Komponenten zentral hinterlegt werden. Dies umgeht beispielsweise die Synchronisierungsproblematik bei einer lokalen Speicherung auf den physikalischen Rechnern.
Zum anderen kann der Speicher ebenfalls dazu benutzt werden, Informationen zwischen den Komponenten auszutauschen. Dies betrifft neben Informationen über den Zustand des Systems auch die Steuerung einiger Komponenten. Auch die zu virtualisierenden Applikationen können dort zentral hinterlegt werden.
Das Konzept eines zentralen Speichers hat jedoch zum Nachteil, dass bei dessen Ausfall das übrige System ebenfalls von einem Ausfall bedroht ist. In der späteren Phase der Realisierung wurde dieser daher redundant im Netzwerk ausgelegt.

Frontend und Webserver Aufgrund vorhandener Kenntnisse im Bereich der Webentwicklung und einiger anderer Vorteile (siehe Abschnitt 4.5), entschied sich die Projektgruppe das Benutzer-Interface in Form einer Webseite selbst zu realisieren.
Dem Nutzer muss es möglich sein, über die Oberfläche neue Applikationen zu starten bzw. vorhandene Applikationen zu stoppen, gegebenenfalls zu ändern oder zu löschen. Des Weiteren soll der Nutzer zu Wartungszwecken einzelne Hosts herunterfahren können.
Wie im Umfeld von Webseiten üblich, steuert der Benutzer das System über Klicks auf entsprechende Icons bzw. trägt Informationen in Formulare ein. Diese Informationen werden an den Webserver gesendet, welcher im Anschluss passende XML-Dateien generiert und diese auf dem NAS speichert.
Gleichzeitig werden im Frontend auch Informationen über die Auslastung der einzelnen Hosts und den Zustand der Applikationen angezeigt. Diese Informationen werden aus Log-Dateien im CSV-Format gelesen, welche sich ebenfalls auf dem NAS befinden.

Load-Management-Unit (LMU) Die Load-Management-Unit (LMU) ist die zentrale Steuerungskomponente des Systems, denn ihre Aufgabe ist es, die Verteilung der Applikationen auf die verschiedenen Hosts zu dirigieren. Dies gilt natürlich sowohl für den initialen Start einer Applikationen, als auch für die Wiederherstellung der gewünschten Redundanz nach dem Ausfall eines Hosts. Weitere Aufgabe der LMU ist es, im Fall einer Wartung oder beim Start einer Applikation, die vorhandenen Applikationen gegebenenfalls umzuverteilen.
Etwaige Steuerungsbefehle findet die LMU auf dem Netzwerkspeicher in Form von XML-Dateien. Sie liest diese ein und generiert daraufhin eine Antwort, welche ebenfalls als XML-Datei auf dem NAS hinterlegt wird.

Lokaler Scheduler / LMU-Interface Der lokale Scheduler, als Teil des Hypervisors, und das LMU-Interface sind bezüglich des gesamten Systems lokale Komponenten, d.h. jeder Host führt eine Instanz dieser Komponenten aus.
Als Schnittstelle zur LMU besteht die Aufgabe des LMU-Interfaces darin, die Befehle der LMU entgegen zu nehmen und auf dem lokalen Host auszuführen. Diese Befehle werden durch eine Nachricht im XML-Format mittels eines Netzwerk-Sockets aus-

getauscht.
Der lokale Scheduler teilt den Applikationen die angegebene Prozessorzeit zu und überwacht deren Einhaltung. Zwecks Überwachung werden diese Informationen an das lokale Monitoring weitergereicht.

Lokales Monitoring Das lokale Monitoring ist ebenfalls eine lokale Komponente und wird daher auf jedem Host ausgeführt. Wie bereits erwähnt, wird ihm der aktuelle Zustand der Applikationen übermittelt. Um dem Nutzer auch ein Bild über den Zustand des Hosts an sich mitzuteilen, ermittelt das lokale Monitoring Informationen über die Auslastung von Prozessor, Arbeitsspeicher und Netzwerk auf Basis von Werkzeugen der Kommandozeile.
Die gesammelten Informationen werden zyklisch an das globale Monitoring in Form von XML-Nachrichten über einen Netzwerk-Socket weitergegeben. Dies geschieht in etwa alle fünf Sekunden, um die Netzwerklast gering zu halten.

Globales Monitoring Dem globalen Monitoring kommen zwei wesentliche Aufgaben zu. Erstens werden die gesammelten Informationen aller Hosts konsolidiert und auf dem Netzwerkspeicher als CSV-Dateien hinterlegt (Logging). Zweitens muss bei etwaigen Problemen mit einer Applikation oder einem Host die LMU informiert werden. Dies betrifft z.B. das Überschreiten von Ausführungszeiten oder den Ausfall einer Applikation bzw. des gesamten Hosts. Da sich LMU und globales Monitoring auf dem selben System befinden, ist diese Benachrichtigung mittels einer lokalen Systemschnittstelle realisiert.

5.1.2. Physikalischer Aufbau

Der physikalische Aufbau gibt einen Überblick darüber, wo die logischen Komponenten im realen System implementiert sind (siehe Abbildung **??**).

Die Grafik zeigt den Aufbau exemplarisch für drei Hosts. Ein Host besitzt jeweils zwei Netzwerk-Schnittstellen: eine zum Sensornetz (für Applikationen) und eine zum Netzwerk des Systems. An letzteres ist neben dem Rechner des Benutzers, welcher das Frontend mittels eines Browsers aufruft, auch der Netzwerkspeicher angeschlossen.

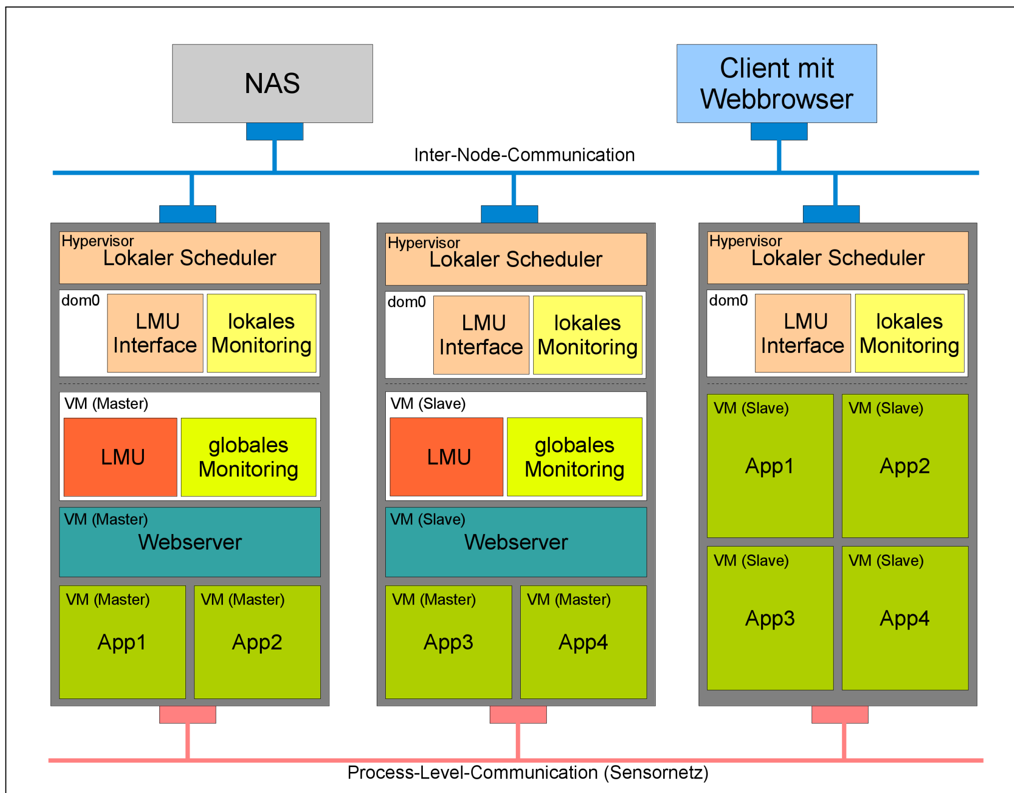


Abbildung 15: Physikalischer Aufbau

Wie bereits oben beschrieben, läuft auf jedem Host, neben dem Hypervisor, eine Instanz des lokalen Monitorings und des LMU-Interfaces.
Die LMU bildet zusammen mit dem globalen Monitoring eine virtuelle Maschine. Die zweite, system-eigene virtuelle Maschine ist der Webserver. Beide werden redundant ausgeführt.
Das Konzept system-eigener virtueller Maschinen bietet den Vorteil, dass diese in gleicher Weise von der LMU behandelt werden können, wie die virtuellen Maschinen der Benutzer-Applikationen. Um die Konfiguration der Kommunikation zu erleichtern, haben die system-eigenen virtuellen Maschinen, das NAS und alle physikalischen Hosts feste IP-Adressen im Systemnetz.
Des weiteren zeigt die Abbildung vier Applikationen, welche jeweils redundant über mehrere Hosts verteilt ausgeführt werden.

Im Folgenden werden nun die wichtigsten Konzepte des Entwurfs genauer erläutert.

5.2. Hochverfügbarkeitskonzept

(Tim Harde) Wie in Kapitel 2.2.2 bereits erläutert wurde, existieren unterschiedliche Ansätze, um bei virtualisierten Systeme Hochverfügbarkeit beziehungsweise Fehlertoleranz zu realisieren. In den folgenden Unterkapiteln soll das innerhalb der Projektgruppe verwendete dreistufige Hochverfügbarkeitskonzept vorgestellt und näher erläutert werden. Bei den einzelnen Stufen handelt es sich dabei um *Single Master*, *Master+Backup* sowie *Master+Backup+Node*.

5.2.1. Single Master

Abbildung 16 zeigt das Konzept des *Single Master*. Hierbei existiert lediglich eine einzige virtuelle Maschine, im Falle eines Hostausfalls oder des Absturzes der VM (Abbildung 16, Fall 1) wird diese auf einem anderen Host mit ausreichend Ressourcen neu gestartet.

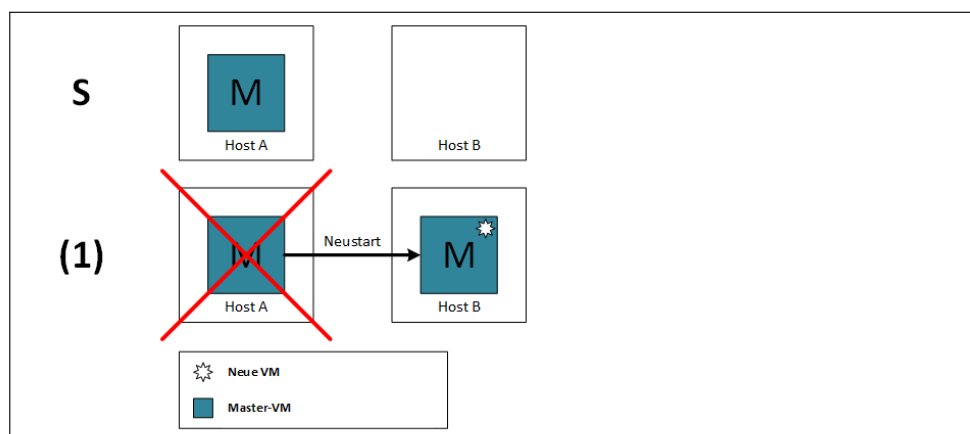


Abbildung 16: HA-Konzept (Master).

Insgesamt bietet diese Lösung eine lediglich sehr eingeschränkte Form der Hochverfügbarkeit ohne Erhaltung des Systemzustandes und somit keinerlei Fehlertoleranz. Aus diesem Grund eignet es sich nur für Applikationen oder Dienste, die über keinerlei Echtzeitanforderungen verfügen.

5.2.2. Master + Backup

Abbildung 17 zeigt das Hochverfügbarkeitskonzept *Master+Backup*. Dieser Ansatz basiert dabei auf der eigentlich VM (Master) sowie einer Replikation (Backup), die mit Hilfe von Remus synchronisiert wird.

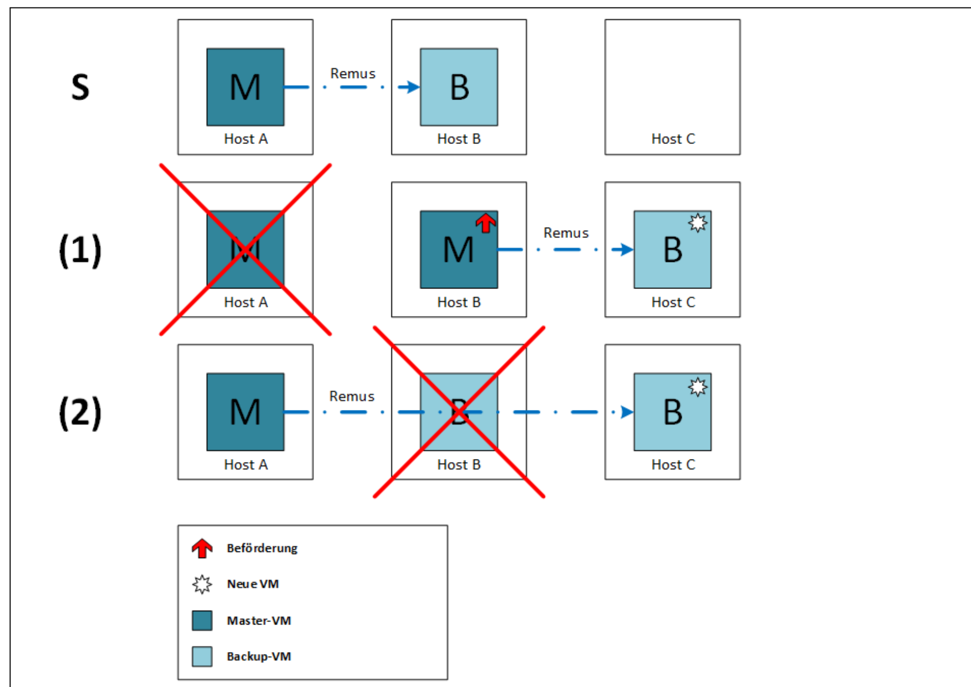


Abbildung 17: HA-Konzept (Master+Backup).

Im Falle des Ausfalls der Master-VM (oder des Hosts, auf dem die Master-VM ausgeführt wird) übernimmt sofort die Backup-VM die Aufgaben der nicht mehr verfügbaren virtuellen Maschine (Abbildung 17, Fall 1). Des Weiteren wird eine neue Backup-VM durch eine neue Remus-Replikation auf einem weiteren Host erstellt.

Im Falle des Ausfalls der Backup-VM (oder des Hosts, auf dem die Backup-VM ausgeführt wird) wird der ursprüngliche Systemzustand durch eine neue Remus-Replikation wiederhergestellt (Abbildung 17, Fall 2). Dieser Ansatz ermöglicht also sowohl Hochverfügbarkeit von Diensten als auch Fehlertoleranz.

5.2.3. Master + Backup + Node

Das dritte und komplexeste Backup-Konzept ist das Konzept unter Verwendung von drei Komponenten: Master, Backup und Node. Im Prinzip unterscheidet sich dieser Ansatz vom vorherigen lediglich dadurch, dass bei der initialen Erstellung der virtuellen Maschine zusätzlich Ressourcen auf einem dritten Virtualisierungsserver (Node) reserviert werden. Dies führt dazu, dass im Falle eines Ausfalls der Master- oder Backup-VM (oder des korrespondierenden Hosts) keine Entscheidung getroffen werden muss, auf welchem Server die neue Remus-Replikation erstellt wird sowie auf jeden Fall ausreichend Ressourcen zur Verfügung stehen, um die fehlende Redundanz wiederherzustellen.

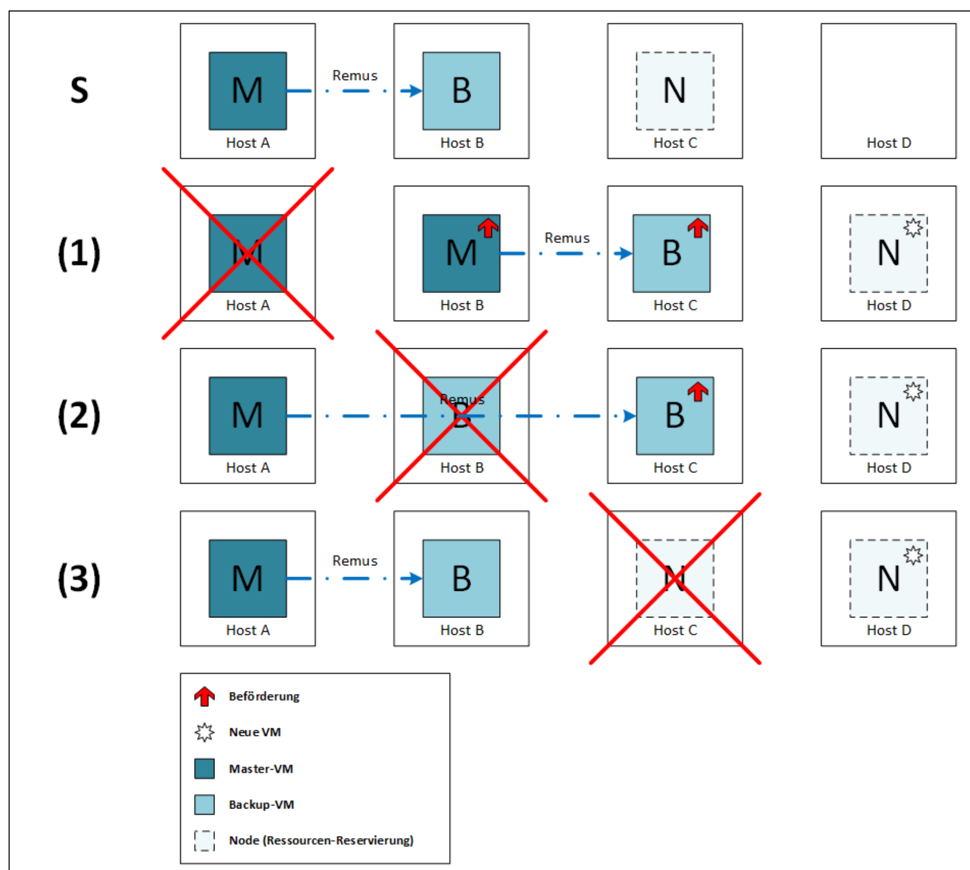


Abbildung 18: HA-Konzept (Master+Backup+Node).

Abbildung 18 zeigt dieses Konzept im Detail. Im Falle des Ausfalls der Master- oder Backup-VM (Fall 1 bzw. 2) kann die neue Remus-Replikation sofort in Richtung

der *Node* gestartet werden. Dieser Ansatz ermöglicht also die Wiederherstellung des ursprünglichen Systemzustandes in einer minimalen Wiederherstellungszeit. Bei Nichtverfügbarkeit des Node-Servers (Fall 3) muss - wie in den beiden vorangegangenen Fällen auch - eine neue Ressourcenreservierung auf einem weiteren Host erfolgen. Diese Reservierung kann dabei aber zu einem späteren Zeitpunkt erfolgen und verzögert somit nicht die möglicherweise notwendige Erstellung der Backup-VM durch eine neue Remus-Replikation.

Die dritte Lösung ermöglicht also sowohl Hochverfügbarkeit von Diensten als auch Fehlertoleranz bei gleichzeitig minimaler Wiederherstellungszeit des ursprünglichen Systemzustandes.

5.3. Frontend

(Hülya Kaplan) Das Konzept der GUI hat sich im Laufe der PG immer weiterentwickelt. Doch schon zu Beginn war schnell entschieden, dass sie auf einem eigenen Webserver laufen und von dort aus sämtliche Informationen von allen Hosts verwalten soll (siehe 4.5.2). In unserem Gesamtsystem knüpft die GUI am NAS an. Unter anderem hat der Nutzer hier einen Einblick in die Fehlermeldungen, Statistiken zu einzelnen Diensten und durchgeführten Aktionen. Um diese Möglichkeiten schlank und übersichtlich zu halten, wurde die GUI in verschiedene Bereiche aufgeteilt und die zusammengehörenden Segmente auch in der Bedienung visuell abgegrenzt. Der Überwachungsbereich der die Daten anzeigt und die Steuerung der laufenden Dienste anbietet, der Bereich zum Starten von Diensten, der Bereich Einstellungen um die Hosteinstellungen einzusehen und der Hilfebereich (siehe Abbildung 22).

Überwachung Geht man in den Überwachungsbereich, so landet man direkt in der Dienst-Überwachung. Hier sieht man in einzelnen Segmenten, welche Dienste laufen, pausiert sind, noch ausstehen, abgelehnt wurden oder abgestürzt sind. Zu jedem Dienst wird sein Name, seine IP mit seiner MAC und seine Redundanz angezeigt. Außerdem wird aufgelistet, auf welchem Host die jeweilige Redundanz des Dienstes läuft. Desweiteren werden die Periode und WCET so wie die Laufzeit und die Logs zu den Diensten mit ausgegeben. An dieser Stelle hat der Nutzer auch die Möglichkeit über die Aktionen-Spalte einzelne Dienste zu pausieren, fortzuführen

oder zu entfernen.
Neben der Dienst-Überwachung gibt es auch die Hosts-Überwachung. Hier wird jeder Host in einem eigenen Tab mit Informationen zu seinem Status, seinem letzten Update, seiner IP etc. aufgeführt. In diesem Bereich kann man ablesen, welche Dienste auf dem jeweiligen Host laufen. Außerdem werden die aktuellsten statistischen Informationen zum Host, wie zB. CPU- oder Netzwerkauslastung grafisch dargestellt. Dieser Bereich bietet auch die Host-Wartung an, worüber man den jeweiligen Host herunterfahren kann, und zeigt die Log-Daten zu diesem Host.
Unter Ereignisse, ebenfalls im Überwachungsbereich, werden regelmäßig sämtliche abgerufene Logs angezeigt. Neben Quelle und Zeitpunkt, erfährt man die eigentlichen Ereignisse und Daten.

Dienst Starten Von diesem Bereich aus startet man einen Dienst. Hierfür muss der Nutzer die vorgegebenen Felder ausfüllen und ein Betriebssystemabbild als Image der Applikation, die er starten will, hochladen. Neben den Angaben zum Dienstnamen, IP und MAC muss auch die Redundanz (M = Master, M+B = Master-Backup, M+B+N = Master-Backup-Node) mit ausgewählt werden. Ferner gibt der Nutzer auch die Periode und die WCET des Dienstes an, wodurch das System ausrechnen kann, wieviel Auslastung jener Dienst verursacht. So wird vorab überprüft, ob der Dienst überhaupt gestartet werden kann oder gegebenenfalls abgelehnt werden muss (siehe 6.3.3). Nachdem der Nutzer seine Daten hochgeladen hat, erscheint sein Dienst im Überwachungsbereich.

Einstellungen Die verfügbaren Hosts werden unter Einstellungen angezeigt. Hier stehen neben dem Namen auch die IP. Will man einen neuen Host hinzufügen, geschieht dies über diesen Bereich.

Hilfe An dieser Stelle erhält der Nutzer Hilfe zur Bedienung der GUI.

5.4. LMU

(Tim Harde) Wie im Kapitel 4.7.1 bereits ausführlich erläutert wurde, ist das globale Scheduling von zentraler Bedeutung, wenn es um Garantien zur Einhaltung von

harten Deadlines geht. Die Load Management Unit (LMU) ist die zentrale Instanz für die Verteilung der entsprechenden virtuellen Maschinen auf die zur Verfügung stehenden Virtualisierungsserver.

5.4.1. Kriterien zur Verteilung virtueller Maschinen

Es existieren unterschiedliche Kriterien, die bei der Verteilung der VMs zu beachten sind. Die wesentlichen Kriterien (Einhaltung der Echtzeit-Garantien und das auf den Virtualisierungsservern verwendete Scheduling-Verfahren) sollen im Folgenden näher betrachtet werden.

Echtzeit-Garantien In der Domäne der Cyber-Physikalischen Systeme sind harte Deadlines allgegenwärtig und die Garantie von Echtzeitverhalten eine große Herausforderung. Um die Einhaltung solcher Garantien gewährleisten zu können, ist es extrem wichtig, die genaue Auslastung der verwendeten Systemkomponenten zu kennen. Die Auslastung des gesamten Systems ist dabei von unterschiedlichen Faktoren abhängig.

Zum einen ist die Anzahl der vorhandenen Virtualisierungsserver sowie die jeweilige Leistungsfähigkeit (beispielsweise in einer inhomogenen Umgebung mit unterschiedlicher Serverhardware) ein wichtiger Parameter, um eine gültige Verteilung der virtuellen Maschinen auf die einzelnen Serversystemen zu berechnen.

Des Weiteren ist es essenziell wichtig, eine möglichst gleichmäßige Auslastung auf den unterschiedlichen Systemen zu erreichen, damit im Falle eines Ausfalls eines Virtualisierungsservers eine minimale Anzahl an virtuellen Maschinen betroffen ist. Hierdurch kann eine Minimierung der Wiederherstellungszeit erreicht werden. Die Verteilung erfolgt auf Basis eines Lastwertes (siehe 4.7.1), der wiederum auf der WCET und Periode der eigentlichen Anwendung basiert.

Problematisch ist hierbei die Tatsache, dass dieser Lastwert zunächst noch keinen Puffer für die jeweiligen Remus-Replikationen sowie für möglicherweise notwendig werdende Live-Migrationen (z.B. im Falle eines Wartungseingriffes) berücksichtigt. Der hierzu benötigte Puffer ist dabei abhängig von der Rate, in der die in der virtuellen Maschine vorhandenen Speicherseiten modifiziert werden (Dirty Rate), was bei einer hohen Rate möglicherweise eine hohe Bandbreite zur Synchronisation

der Speicherseiten innerhalb der Remus-Replikation benötigt.

Diese Parameter dürfen bei der Verteilung der virtuellen Maschinen auf die unterschiedlichen Server keinesfalls außer Acht gelassen werden, da dies zu einer Überlastung einzelner Systeme führen könnte. In einer solchen Lastsituation besteht dabei die Gefahr, dass die obligatorischen Echtzeit-Garantien nicht mehr eingehalten werden können.

Scheduling-Verfahren Um eine gültige Verteilung (d.h. eine garantierte Einhaltung für die jeweils vorgegebenen Deadlines) der CPS-Gastsysteme zu ermöglichen, ist es insbesondere wichtig, dass der auf den Virtualisierungsservern verwendete Scheduling-Algorithmus bekannt ist. Dieser Algorithmus wird vom Hypervisor dazu verwendet, die Ausführungsreihenfolge der einzelnen VMs festzulegen (siehe 4.6) und muss dementsprechend bei der Verteilung der VMs berücksichtigt werden.

5.4.2. Dynamische Anpassung des Systemzustandes

Es können unterschiedliche Situationen entstehen, in denen die LMU die momentane Verteilung der virtuellen Maschinen anpassen muss. Diese Situationen sind beispielsweise die Erkennung eines neuen Hostsystems, auf dem weitere virtuelle Maschinen ausgeführt werden können sowie der Ausfalls eines derzeit vorhandenen Virtualisierungsservers. Des Weiteren können vom Benutzer neue Applikationen gestartet werden, die von der LMU auf die Virtualisierungsserver unter Beachtung der oben genannten Kriterien verteilt werden müssen. Das System muss also in der Lage sein, dynamisch auf Änderungen der Systemkonfiguration oder -topologie zu reagieren. Diese einzelnen Fälle werden wir im Folgenden näher betrachten.

Neuer Host Bei der Erkennung eines neuen Hosts benötigt die LMU Informationen über die Leistungsmerkmale des neuen Systems - dies können beispielsweise die jeweils vorhandene Rechenkapazität oder der vorhandenen Hauptspeicher sein.

Diese Informationen müssen von der LMU in einer internen Datenstruktur verwaltet werden, um jederzeit die maximale Auslastung eines jeden Virtualisierungsservers bestimmen zu können.

Hostausfall Im Falle eines Hostausfalls wird die LMU vom globalen Monitoring benachrichtigt. In Abhängigkeit davon, welche virtuelle Maschinen betroffen sind und welches Redundanzkonzept (siehe 5.2) verwendet wurde, muss die LMU unterschiedliche Aktionen durchführen, um den globalen Systemzustand wiederherzustellen.

Die folgende Fallunterscheidung erläutert die Vorgehensweise bei Verwendung der unterschiedlichen Redundanzkonzepte:

- **Single Master** (M): Die betroffene VM benötigt keine Backup-VM und verfügt somit über keinerlei Hochverfügbarkeit. Die VM muss von der LMU auf einem anderen Server neu gestartet werden.
- **Master + Backup** (M+B): Sollte der Host, auf dem die Master-VM ausgeführt wurde, ausgefallen sein, so hat die Backup-VM dies erkannt und die Ausführung der Applikation bereits übernommen (Failover). Sollte es sich bei dem ausgefallen Host um das Hostsystem der Backup-VM handeln, so verfügt die Master-VM über keine Redundanz mehr. In beiden Fällen muss eine neue Backup-VM (d.h. eine neue Remus-Replikation) durch die LMU initiiert werden.

  Aufgrund der Tatsache, dass im Redundanzkonzept M+B lediglich eine einzige Backup-VM existiert, verfügt die Applikation bis zur Erstellung der neuen Remus-Replikation über keinerlei Ausfallsicherheit mehr. Es schützt deshalb nur gegen einzelne Fehler innerhalb eines gegebenen Zeitraums.
- **Master, Backup + Node** (M+B+N): Sollte der Host, auf dem die Master-VM ausgeführt wurde, ausgefallen sein, so hat die Backup-VM dies erkannt und die Ausführung der Applikation bereits übernommen (Failover). Sollte es sich bei dem ausgefallen Host um das Hostsystem der Backup-VM handeln, so verfügt die Master-VM über keine Redundanz mehr.

  In beiden Fällen wird sofort eine neue Remus-Replikation gestartet (Ziel: System mit zuvor reservierten Ressourcen (Node)). Anschließend muss eine neue Reservierung der Ressourcen auf einem anderen Zielsystem durch die LMU initiiert werden.

Neue Applikation Wenn eine neue Applikation über die GUI gestartet wurde, so werden alle notwendigen Instanzen der VMs (abhängig vom verwendeten Redun-

danzkonzept) gescheduled.

Nach erfolgreichem Scheduling erfolgt eine Kommunikation zwischen der LMU und dem LMU-Interface auf den einzelnen Virtualisierungsservern. Dies führt dazu, dass der „Master-Server“ die entsprechende Konfiguration und das Applikationsimage vom NAS kopiert und die neue virtuelle Maschine startet. Eine eventuell vorhandene Backup-VM wird durch die Initiierung einer Remus-Replikation auf dem „Backup-Server“ erzeugt. Die Reservierung der Ressourcen auf dem „Node-Server“ erfolgt lediglich in der internen Datenstruktur der LMU.

Nachdem die neue Applikation gestartet wurde, müssen noch das lokale und globale Monitoring angepasst werden, damit die Einhaltung der Deadlines überwacht werden kann.

5.4.3. Konfigurations- und Erweiterungsmöglichkeiten

Insgesamt existieren unterschiedliche Ansätze, um die LMU zu konfigurieren und so das globale Scheduling möglichst flexibel zu machen. Diese Anpassungsmöglichkeiten sind notwendig, um es an unterschiedlichen Anforderungen anzupassen zu können.

Das initiale Konzept des globalen Scheduling sieht vor, dass SEDF als lokaler Scheduling-Algorithmus auf den einzelnen Virtualisierungsservern verwendet wird, da dies die Verwendung eines relativ einfachen globalen Scheduling-Algorithmus ermöglicht. Die unterschiedlichen Verfahren zur Lastverteilung wurden in Kapitel 4.7.1 bereits eingehend beschrieben.

Die jeweils relevanten Parameter (beispielsweise der Schwellwert der maximalen Auslastung eines einzelnen Cores bei SEDF) können dabei frei konfiguriert werden. Eine maximale Core-Auslastung von beispielsweise 70% (im Worst Case, d.h. alle auf diesem Core ausgeführten Applikationen benötigen ihre WCET für die Berechnungen einer Periode) sorgt dafür, dass genügend Puffer für gegebenenfalls anfallenden Overhead (Live-Migration bzw. Remus-Replikation, siehe oben) vorhanden ist.

5.5. Monitoring

(Parinas Nassiri) Unter dem Kapitel 4.4 wurde aufgeführt, warum das Monitoring-System

eine zentrale Rolle in der CyPhy-Control-Architektur spielt. Das Monitoring soll eine effiziente Erhebung und Weitergabe von Informationen und Fehlermeldungen gewährleisten. Um so wichtiger ist dabei die korrekte Funktionsweise und ein logischer und effizienter Ablauf, denn die Informationen müssen so durchgereicht werden, dass das weitere System möglichst zeitnah die relevanten Informationen erhält und verarbeiten kann.

Die Fehlererkennung muss das System schnellstmöglich alarmieren und kritische Werte anzeigen beziehungsweise weitergeben. So soll bei auftretenden Problemen eine möglichst schnelle Stabilisierung des Systems unterstützt werden.

Das Monitoring-System kann in zwei Teile aufgeteilt werden (siehe Abb. 19), das

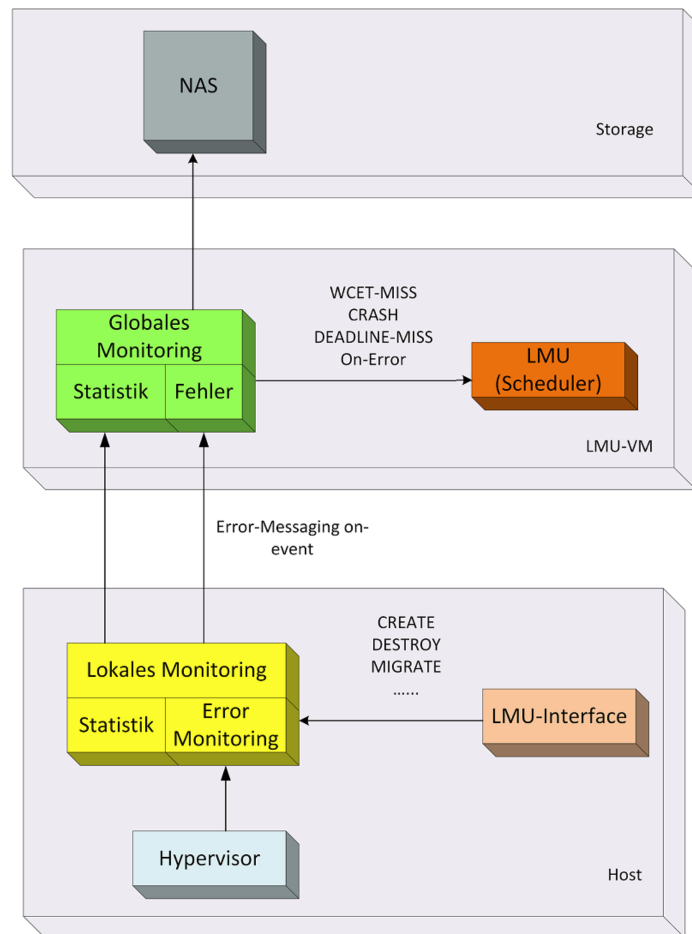


Abbildung 19: Übersicht des Monitorings

lokale und das globale Monitoring. Das lokale Monitoring befindet sich auf jedem Host in der Dom0, wohingegen das globale Monitoring sich nur in der LMU befindet.

Zunächst erhält ausschließlich das lokale Monitoring Informationen beziehungsweise Meldungen vom LMU-Interface, wie zum Beispiel, dass eine VM migriert, erstellt oder beendet werden soll. Zudem wird der Status der VMs, wie zum Beispiel ob diese pausiert sind, ermittelt und weitere Statusänderungen erfasst. Darüber hinaus wird überprüft, ob tatsächlich alle VMs laufen, welche laufen sollten. Sollten VMs abgestürzt sein, wird dies erkannt und dem globalen Monitoring mitgeteilt. Gleichermaßen geht das lokale Monitoring bei den Remus-Replikationen vor.
Weiterhin ermittelt das lokale Monitoring Daten über Host-Events und App-Events. Diese Daten lassen sich in statische, zyklische und Event-basierte Daten für Hosts, Applikationen und Fehlermeldungen gruppieren.
Zudem gibt es eine Schnittstelle vom Hypervisor zum lokalen Monitoring. Dabei wartet das lokale Monitoring passiv auf eintretende Meldungen über verpasste Deadlines und WCET Überschreitungen. Sämtliche erfasste Statistiken und Ereignisse, wie beispielsweise verpasste VM Deadlines, WCET Überschreitungen von VMs, Remus Failover und abgestürzte Hosts, werden an das globale Monitoring weitergegeben.
Das globale Monitoring ist sozusagen die Sammelstelle und die zentrale Instanz des Monitorings und fungiert als logische Einheit. Die vom lokalen Monitoring ermittelten Daten werden an das globale Monitoring übermittelt, woraufhin das globale Monitoring diese Daten in entsprechende Log-Dateien speichert oder vorhandene Log-Dateien ergänzt. Zudem gibt es noch Error-Logs für die jeweiligen Hosts, worin zum Beispiel abgestürzte Hosts aufgeführt sind. Die Log-Dateien werden im CSV Format zentral für alle weiteren Module des Systems im NAS abgespeichert.
Von hier aus liest das Frontend die Log-Dateien und stellt einige Daten und gegebenenfalls Fehler auf der Webseite dar. Durch die grafische Oberfläche sollen die wichtigsten Werte schnell und einfach für den Nutzer sichtbar sein.
Abgesehen von der Erstellung von Log-Dateien, hat das globale Monitoring die wesentliche Funktion, die LMU über Fehlerfälle, wie verpasste Deadlines, überschrittene WCETs oder abgestürzte VMs zu informieren. So kann die LMU korrekt mit den erhaltenen Meldungen weiterarbeiten und das Frontend die Daten vom NAS ermitteln und diese dann anzeigen.
Die Listen aller ermittelten Daten, Fehlerfälle und Logs befinden sich im Anhang A

und A. Die technische Erläuterung des Monitorings wird in Kapitel 6.4 ausführlich erläutert.

5.6. Lokales Scheduling

(Daniel Stoller, Vasco Fachin) Das Scheduling der virtuellen Maschinen spielt eine zentrale Rolle, um Echtzeitfähigkeit und Hochverfügbarkeit als zentrale Anforderungen an das System zu gewährleisten. Besonders die Laufzeiten der virtuellen Maschinen soll gesetzt und beachtet werden. Da wir auf XEN als Virtualisierungslösung aufsetzen, ergeben sich dadurch verschiedene Möglichkeiten für die Auswahl des lokalen Schedulers.

Der Schedulingalgorithmus „Simple Earliest Deadline First“ (kurz SEDF) ist in XEN bereits implementiert, sodass dieser ohne großen Aufwand direkt verwendet werden kann und kein anderer Schedulingalgorithmus manuell und zeitintensiv implementiert werden muss. Das SEDF-Verfahren garantiert eine lokale Echtzeit-Fähigkeit, da immer die VM ausgeführt wird, deren Deadline zeitlich am nächsten liegt. Allerdings ist das Verhalten des Systems im Fall einer Systemüberlastung nicht vorhersehbar [38].

Auch der bereits vorimplementierte Credit-2-Scheduler ist für den Einsatz in unserem System ungeeignet, da er keine Möglichkeit bietet, den virtuellen Maschinen maximale Laufzeiten vorzugeben.

Weiterhin wurde „RT-XEN“ als ein echtzeitfähgier Xen-Scheduler (siehe auch 4.6.2), der in den letzten Jahren von einigen Universitäten entwickelt wurde [12], in Betracht gezogen. Allerdings unterstützt dieser Scheduler (in der Version 0.3) keine Migration und wurde deswegen nicht weiter getestet. Während der PG-Zeit wurde noch die Version 1.0 im Mai 2013 veröffentlicht, welche anschließend ebenfalls getestet, aber schon ab drei virtuellen Maschinen nicht stabil war [13].

Aufgrund der Unzulänglichkeiten der oben genannten Scheduling-Verfahrens fiel die Entscheidung für die Implementierung eines Fixed-Priority Schedulers, auf den in

[12] Washington University in St. Louis und University of Pennsylvania (siehe [3])

[13] Seit Ende November 2013 (siehe [39]) existiert auch eine v2.0 Version von RT-XEN. Die v2.0 Version erschien aber, als einen eigenen VM-Scheduler innerhalb der PG entwarf und im Aufbau war.

Kapitel 6.5 eingangen wird.

5.7. Gastsystem

(Gregor Kotainy)

Die Rolle des Gastsystems Durch den stetig wachsenden Anteil an Software in der technischen Evolution und den damit verbundenen Vorteilen, wie zum Beispiel der Wiederverwendung von Ressourcen - durch den Einsatz generischer Hardware - oder der Erweiterung und Verbesserung der Funktionalität, liegen die Ziele der Projektgruppe offen auf der Hand. Um jene Ziele in Cyber-physikalischen Systemen erfüllen zu können, wird die Hardware langfristig durch Software im entsprechenden Gastsystem ersetzt. Die strikte Trennung der einzelnen Cyber-physikalischen Komponenten in Gastsysteme - und nicht etwa in Programme - hat den Vorteil, tatsächliche Ressourcen durch den geschickten Einsatz virtueller Abstraktionen und eines intelligenten Schedulers im Hypervisor umzuverteilen.

Die Rolle innerhalb der PG Die in unserem System als virtuelle Maschinen zum Einsatz kommenden Betriebssysteme müssen für den Dauerbetrieb geeignet sein und eine geringe Fehleranfälligkeit haben. Ein Betriebssystem, das wenig Funktionalität bereitstellt und sehr einfach gestrickt ist, verhält sich fehlertoleranter als komplex aufgebaute Allzweck-Betriebssysteme. Des Weiteren kann es schneller gebootet und migriert werden.

Ein kleines Betriebssystem verringert nicht nur den aufwändigen Migrationsaufwand von einer Maschine zur anderen, sondern hält die Möglichkeit offen, das System auf diversen Maschinen redundant zu starten und das Ausgabeverhalten zu vergleichen. Mithilfe dieses Vergleichs lassen sich Anomalien aufspüren und fehlerhafte Systeme mit geringem Aufwand neu starten. Dieses Vorgehen wird von der Projektgruppe allerdings nicht angestrebt und soll hier demnach auch nicht weiterverfolgt werden.

Typisches Applikationsszenario: zyklische Echtzeit-Sensorapplikation Ein beispielhaftes System würde über einen einzigen Task verfügen, der zyklisch Sensordaten empfängt und verarbeitet. Im Notfall würde dieser in einen Fehlerzustand

übergehen und eine Schutzfunktion auslösen. In Kapitel 7 wird eine von der Projektgruppe implementierte, zyklische Distanzschutz-Applikation beschrieben. Diese Applikation erhält in jeder Periode Sensordaten, welche verarbeitet werden müssen. Es gilt demzufolge *Deadline = Periode*. Eine schnelle Kommunikation mit dem Hypervisor ist sehr entscheidend, um einen abgeschlossenen Berechnungszyklus rechtzeitig zu signalisieren. Dies ist notwendig, da es sich um virtuelle Maschinen handelt und die Gastsysteme abhängig von der Last geschedulet werden. Die betroffene *VM* würde dann für die aktuelle Periode nicht mehr geschedulet werden, wobei die Ressourcen den anderen *VMs* zugute kommen.

Ein Gastsystem kann über drei verschiedene Methoden mit dem lokalen Scheduler kommunizieren. Da die ersten zwei der folgenden Kommunikationsmethoden einen erheblichen Nachteil gegenüber der dritten haben, wurden sie vernachlässigt und nicht implementiert.

1. Methode **IO-Wait**:
   Nachdem die Berechnung für die Periode fertig ist, geht der Task in einen *IO-Wait* Zustand über. Die *VCPU* ist danach *IDLE*, wonach die *VM* verdrängt werden kann. Bei eingehenden Sensorpaketen wird das entsprechende Betriebssystem und der Task geweckt, wodurch er einen weiteren Berechnungszyklus starten kann. Die Anzahl der Netzwerkpakete des Sensors führt zu einem Tradeoff zwischen dem Wake/Suspend Overhead - den jedes eintreffende Paket verursacht - und dem Nutzen der Entlastung des Prozessors.

   Die Notwendigkeit der Ausführung eines Gastsystems - für welches Netzwerkpakete anliegen - ist unklar. Sollte der Prozess seine fertige Abarbeitung nur insofern signalisieren können, indem er auf neue Daten wartet, existiert ein Problem: Falls Daten mit hoher Frequenz eintreffen und ein Puffer praktisch nicht geleert werden kann, wird es nicht zu einem *IO-Wait* kommen. Der Scheduler kann objektiv nicht beurteilen, ob der Prozess für die Periode genügend Daten empfangen und verarbeitet hat. Er muss aktiv eingreifen, um anderen Systemen noch Rechenzeit zukommen zu lassen. Eine *preemptive Verdrängung* kann zu einem weiteren Problem führen, indem die Puffer irgendwann überlaufen und Daten für aktuelle Perioden verloren gehen.

   Um diesen Ansatz sinnvoll zu verfolgen, sollte der Scheduler darüber infor-

miert werden, wie viele Pakete eine VM empfangen muss, um eine Berechnung abzuschließen.

2. Methode **Versand von Netzwerkpaketen**:
   Zur Kommunikation mit dem Hypervisor werden Netzwerkpakete versandt. Da der Virtualisierungsserver über zwei Netzwerkkarten verfügt, ist es durchaus denkbar, die Sensordaten über die eine Netzwerkkarte entgegenzunehmen und die Signalisierung der fertigen Berechnung über die zweite Netzwerkkarte durchzuführen.

   Einen Nachteil, den dieses Methode hat, sind Latenzen. Die Netzwerklatenzen führen dazu, dass ein dem Hypervisor signalisiertes Ende der Berechnung verzögert bei ihm eintrifft.

   Wenn ein Paket noch vor der Deadline eintrifft, kann entschieden werden dass die Berechnung rechtzeitig beendet und die Deadline nicht verpasst wurde. Ist die Berechnung hingegen genau zum Ende der Periode fertig, trifft das Paket erst nach der Deadline beim Scheduler ein. Je länger ein Paket verzögert wird, desto später kann erst darauf reagiert werden, wenn die Deadline verpasst wurde.

3. Methode **Hypercall**:
   Um die Vorteile der zweiten Methode behalten zu können und Latenzen zu vermeiden muss das Betriebssystem direkt mit dem Hypervisor kommunizieren, was über *Hypercalls* realisiert werden kann.

   Das Gastsystem muss für den Einsatz dieser Methode angepasst werden und eine spezielle, vom Hypervisor zur Verfügung gestellte, API implementieren, um damit die Schedulingfunktion des Schedulers zu triggern. Die notwendige Anpassung des Hypervisors findet sich in Kapitel 6.5.4, die der Gastsysteme in Kapitel 6.6.

   Der Implementierungsaufwand für *Hypercalls* - der deutlich größer als ein *IO-Wait* oder ein verschicktes Netzwerkpaket war - hat sich im Endeffekt dennoch gelohnt, da die langsameren Methoden nicht akzeptabel waren.

5.8. Systemstart

(Dominic Wirkner) Das System wirkt im laufenden Zustand stabil und selbst-regulierend. Jedoch muss das System zunächst in diesen Zustand gebracht werden. Der Systemstart, sprich das Starten eines oder mehrerer physikalischer Hosts (z.B. nach einem Stromausfall), ist problematisch. Die Ursache dafür ist unter anderem die zentrale Steuerungskomponente LMU. Wie die Abbildung **??** zeigt, wird diese in einem Master-Slave-Konzept redundant ausgeführt.
Es muss jedoch zunächst eindeutig sein, welcher Host die LMU initial startet oder gestartet hat. Aus diesem Zweck wurde der Discovery-Prozess entwickelt. Jeder Host überprüft während des Bootvorgangs, ob eine Instanz der LMU bereits läuft. Ist dies nicht der Fall, müssen sich die Hosts über einen gewissen Zeitraum darauf einigen, welcher die LMU ausführt.

Das Konzept In einer ersten Version tauschten die Hosts ihre IP-Adressen untereinander aus, bis innerhalb eines Zeitfensters keine Änderungen mehr zu verzeichnen waren. Im Anschluss startete dann der Host mit der kleinsten IP-Adresse die LMU. Aufgrund der Tatsache, dass alle Hosts feste IP-Adressen haben, kann der Discovery-Prozess folgendermaßen vereinfacht werden: Jeder Host lädt eine Konfigurationsdatei vom NAS, welche die IP-Adressen aller am System beteiligten Hosts enthält. Im Anschluss überprüft jeder Host, ob ein anderer Host mit kleinerer IP-Adresse erreichbar ist. Ist dies nicht der Fall, so kann der jeweilige Host die LMU starten. Ein aufwändiger Austausch von IP-Adressen entfällt in diesem Fall. Dafür muss die Konfiguration des Netzwerkes, sprich die Zuordnung von IP-Adressen, mit der Konfiguration auf dem NAS übereinstimmen.
Ist der Discovery-Prozess abgeschlossen und die LMU gestartet, muss jeder Host zusätzlich die lokalen Komponenten starten. Dazu gehören das LMU-Interface, um Steuerungsbefehle entgegen zu nehmen, und das lokale Monitoring. Letzteres ist nicht nur für das Sammeln der Laufzeitinformationen verantwortlich, sondern teilt der LMU auch mit, dass der jeweilige Host erfolgreich gestartet und bereit ist, Befehle entgegen zu nehmen.

6. Endprodukt

Anhand des im vorherigen Kapitel 5 vorgestellten Entwurfs, wurden die einzelnen Komponenten implementiert. Welche technischen Werkzeuge und Verfahren dabei zum Einsatz kamen, wird im Folgenden genauer beschrieben.

6.1. Host-Konfiguration

(Vasco Fachin)

6.1.1. Xen als Hypervisor

Derzeit besteht die Virtualisierungsplattform aus vier Rechnern, auf denen jeweils Ubuntu 12.10 mit Xen 4.1.4 installiert wurde. Die Entscheidung für Xen als Hypervisor wurde getroffen, weil KVM keinen Fehlertoleranzmechanismus implementiert.

Innerhalb der Projektgruppe wurden zwei verschiedene Xen-Versionen evaluiert: 4.1.4 und 4.2.1, die jeweils verschiedene Toolstacks zu der Verwaltung verwenden (`xend/xm` bzw. `xl`). Die unterschiedlichen Funktionalitäten wurden am Anfang der Projektgruppe getestet und die Entscheidung zu Gunsten der Version 4.1.4 gefällt. Nachfolgend finden sich noch einige Anmerkungen zu den evaluierten Xen-Version.

Xen 4.2.1 Die Xen-Version 4.2.1 enthält standardmäßig den neuen xl-Toolstack. In den letzten Jahren wurden neue Technologien zur Virtualisierung entwickelt, was zur Einführung von zusätzlichen Bibliotheken, Diensten und Werkzeugen führte. Beispielsweise implementieren *libvirt*, eine Linux-Bibliothek zur Verwaltung unterschiedlicher Virtualisierungslösungen, XAPI, der *Citrix XenServer* Toolstack sowie der *xend*-Daemon ähnliche *low-level* Operationen (wie beispielsweise *Speicherverwaltung* und *Hypercalls*). Dies führte zu Code-Duplikation, Ineffizienzen und diversen Bugs, weshalb der xl-Toolstack von Grund auf neu entwickelt wurde [17].

Allerdings ist Remus in dieser neuer Version nicht benutzbar. Vor allem wurde das Netzwerkpuffer-Managementssystem, das von Remus benutzt wird, um die VM zu replizieren, nicht richtig implementiert und führt zu *Segmentation Fault*-Fehlern.

Name	Fujitsu Esprimo P710
CPU	Intel® Core™ i5-3570-Prozessor, 6 MB, 3,40 GHz
Chipsatz	Intel® Q75
RAM	2x 4GB DIMM DDR3 Synchronous 1600 MHz
FestPlatte	2x 500.1 GB ATA ST500DM002-1BD14
Netzwerkkarte	82579V Gigabit Network Connection Broadcom Corporation NetXtreme BCM5761 Gigabit Ethernet PCIe

Tabelle 1: Details zur verwendeten Hardware der Virtualisierungsserver.

Xen 4.1.4 Aufgrund der Remus-Problematik in 4.2.1 wurde Xen auch in der Version 4.1.4 getestet, die den `xend`-Daemon zur Verwaltung verwendet. Diese Version scheint insbesondere im Bezug auf Remus stabiler zu sein und wird deshalb innerhalb der Projektgruppe als Hypervisor eingesetzt.

6.1.2. Systemarchitektur

Wie oben bereits erwähnt wurde besteht die Virtualisierungsplattform aus vier baugleichen Rechnern (siehe Tabelle 1). Wichtig ist dabei insbesondere, dass die Rechner das gleiche CPU-Modell verwenden , da sowohl die Live-Migration als auch Remus die gleiche *Instruction Set Architecture* (kurz: ISA) benötigen.

Es wurde einige Tests zur Migration zwischen Intel und AMD CPUs durchgeführt (Intel Core i5-3570 und AMD Phenom X4).

Die Migration einer virtuellen Maschine von der AMD- zu der Intel-Plattform funktionierte problemlos, von der Intel- zur AMD-Plattform wird die VM in mehreren Fällen nicht fortgesetzt. Diese Problem wurde in unserem Fall dadurch verursacht, weil verwendete Chipsatz (AMD 785G) keine Vollvirtualisierung unterstützt [27].

Als Dom0 wird Ubuntu in der Version 12.10 eingesetzt. Die Installation erfolgte mit einer netboot-Distribution, da nur ein Minimalsystem die Basis für die nachfolgenden Xen 4.1.4-Installation bildete. Die Xen-Installation erfolgte ohne Probleme.

Zusätzlich wird ein NFS-Verzeichnis gemountet, damit die unterschiedlichen VM-Images ohne weitere Synchronisationsmechanismen (wie beispielsweise DRBD) zwischen den einzelnen Rechnern zur Verfügung stehen. Außerdem wird dieses Verzeich-

nis für die Kommunikation zwischen der LMU, dem globalen Monitoring und dem Webserver verwendet.

Um das Testen in einer realitätsnahen Umgebung zu ermöglichen, wurden insgesamt vier Rechner als Virtualisierungsserver zur Verfügung gestellt. Insbesondere waren mehr als zwei Hosts notwendig, um Tests ohne triviale Verteilung der virtuellen Maschinen (alle VMs auf Host A. alle Remus-Replikationen auf Host B) zu gewährleisten. Im Falle eines Ausfalls könnte in diesem Fall der verbleibende Rechner die Ausführung der VMs und somit die Bereitstellung der Dienste übernehmen.

6.1.3. Fehlertoleranz mit Remus

Der Xen-Hypervysor wurde insbesondere wegen Remus als Fehlertoleranz-Mechanismus ausgewählt. Dieser Mechanismus soll in diesem Unterkapitel kurz näher erläutert werden.

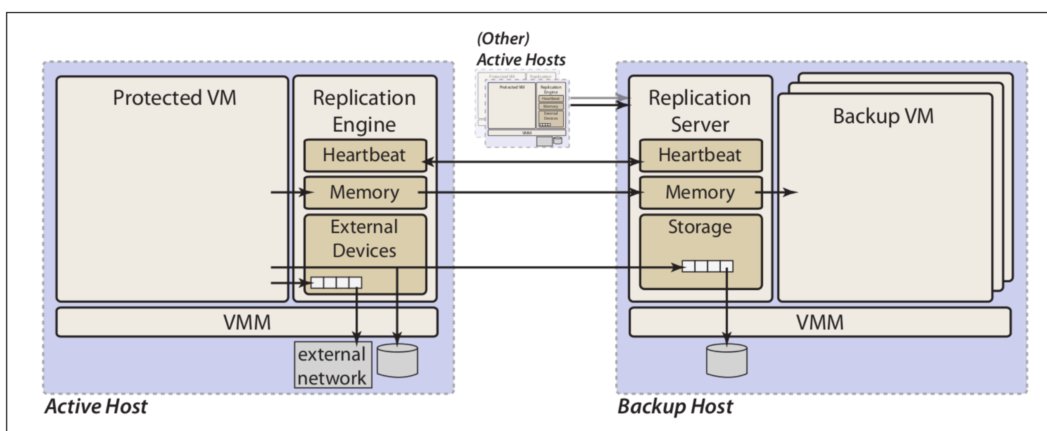


Abbildung 20: Remus Architektur [21]

Remus ist ein Softwaresystem, das Hochverfügbarkeit auf Hardwareebene ermöglicht. Der Zustand einer VM wird in einem konfigurierten Zeitintervall (bis zu allen 25 Millisekunden) auf einen Backup-Host repliziert (siehe Abbildung 20). Diese Methode wird durch ein sogenanntes Checkpointing-Verfahren implementiert. Jede Iteration des Verfahrens besteht dabei aus den folgenden vier Schritten (siehe Abbildung 21):

- Wie bei Live-Migration wird die VM kurz pausiert (*Checkpoint*). Jede Zustandsänderung der primären virtuellen Maschine wurde in einen Puffer (*State Buffer*) kopiert.

- Der Pufferszustand wird zum Backup-Host übertragen (*Transmit*) und die primäre VM fortgesetzt (*Speculative Execution*).
- Die vollständige Zustandsmenge wird vom *Backup Host* empfangen (*Sync*). Die Backup-VM verfügt nun über den identischen Systemzustand wie die primäre VM zum Zeitpunkt des Checkpoints.
- Der Puffer wird freigegeben (*Release*); weitere Zustandsänderungen der primären VM werden übertragen.

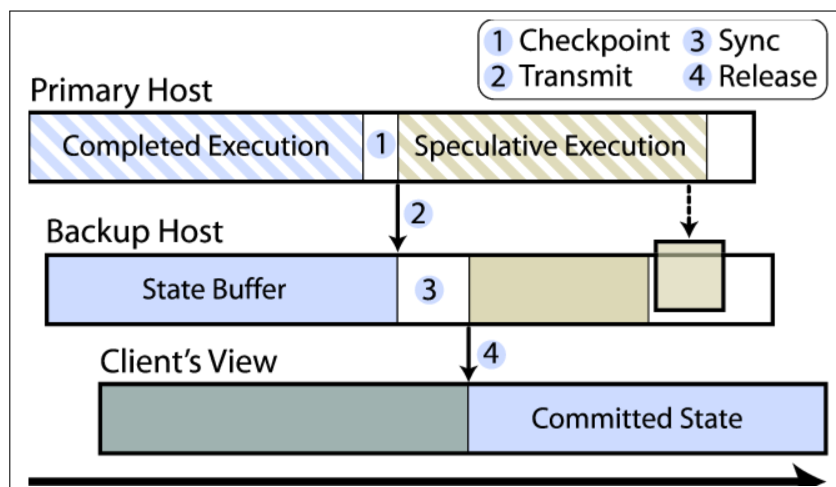


Abbildung 21: Remus Architektur [21]

Die eigentliche Fehlererkennung erfolgt über *Heartbeat*-Nachrichten. Wenn innerhalb eines vordefinierten Intervalls keine neue Nachrichten eintreffen, nimmt das Backup-System an, dass ein Fehlerzustand (z.B. Hostausfall oder Absturz der VM) auf dem primären Host eingetreten ist: die auf dem Backup-Host replizierte VM wird nun ausgeführt (*Failover*).

6.2. Frontend

(Dominic Wirkner, Parinas Nassiri) Die folgenden Unterkapitel beschreiben die Implementierung der Benutzeroberfläche, wie sie im Entwurf (siehe Abschnitt 5.3) gefordert wurde.

6.2.1. Genutzte Technologien

Die Umsetzung einer Benutzeroberfläche als dynamische Webseite erfordert zunächst eine Entscheidung bezüglich der serverseitigen Programmiersprache. Neben der sehr weit verbreiteten Skriptsprache PHP sind auch Implementierungen auf Basis von Java (Server-Pages, Server Faces, Servlets) oder Microsofts Active Server Pages (ASP.NET) denkbar. Letzteres erfordert zumeist sowohl für das Server-Betriebssystem als auch die Entwicklungsumgebung kostenpflichtige Lizenzen, da es sich um eine proprietäre Technologie handelt, und wird daher von der Projektgruppe als ungeeignet beurteilt. PHP und die Java-Technologien unterscheiden sich in den Funktionsmöglichkeiten kaum. Für beide ist zudem quell-offene und kostenfreie Server-Software verfügbar. PHP hat jedoch als Skriptsprache einige Nachteile, wie z.B. fehlende Typ-Sicherheit. Dadurch werden Fehler im Quellcode häufiger erst zur Laufzeit sichtbar. Aufgrund der Tatsache, dass bereits einige Mitglieder Erfahrungen mit der Entwicklung von Webseiten auf Basis von PHP gemacht haben und sich dadurch eine Zeitersparnis erhoffen lies, entschied sich die Projektgruppe dennoch für die Nutzung von PHP.

Wenn auch nicht vollständig befolgt, so diente das Model-View-Controller-Pattern als Basis für den Entwurf der Programmstruktur. Das Entwurfsmuster sieht vor, den Quellcode der Darstellung (View) und der Manipulation der Daten (Model) voneinander zu trennen. Als Bindeschicht existiert kontrollierender Code (Controller), welcher passende Daten abruft/manipuliert und sie der Darstellung übergibt. Zur Unterstützung dieses Konzeptes wurde das auf PHP basierende Smarty [14] verwendet.

Da der Entwurf vorsah, den Webserver als virtuelle Maschine zu realisieren, griff die Projektgruppe auf das Linux-basierte Betriebssystem TinyCore zurück, vorgestellt in Abschnitt 4.3.1, und erweiterte dies entsprechend um einen Apache-Webserver und PHP.

In der heutigen Zeit macht die serverseitige Programmiersprache jedoch nur die Hälfte der Technologien aus, welche für die Erstellung von dynamischen Webseiten genutzt werden. Es ist de facto Standard die Inhalte der Webseite mittels JavaScript zu manipulieren und lästige Wartezeiten beim Seitenwechsel durch Asynchronous JavaScript and XML (kurz Ajax) zu umgehen. Für beide Anwendungsbereiche bie-

[14] Smarty Template-Engine, *www.smarty.net*

tet das weit verbreitete Framework jQuery [15] eine einfach anzuwendende Lösung. Der Nutzen lässt sich zudem durch zahlreiche verfügbare Erweiterungen vergrößern. Einige davon sind auch zur Implementierung der Benutzeroberfläche verwendet worden; das Plugin DataTables [16] zur dynamischen Darstellung von Tabellen, ein Plugin zur clientseitigen Formularvalidierung [17] und jQuery UI [18], welches die Erstellung von Elementen für Nutzerinteraktionen, wie Tabs oder Dialogfenster, vereinfacht. Zur Darstellung von Diagrammen wurde HighCharts [19] verwendet.
Ein Teil des Entwurfs verlangt es, dass der Benutzer eine Rückmeldung zu seinen Steuerungsbefehlen bekommt. Weil diese jedoch von der LMU asynchron verarbeitet werden, müsste der Benutzer auf die Antwort warten oder es wäre auch denkbar, solch eine Funktionalität durch zyklischen Abrufen mittels JavaScript zu realisieren. Mit Erscheinen der Version 5 des HTML-Standards wurde das Konzept von WebSockets in viele Browser implementiert. Diese ermöglichen es dem Webserver eine Verbindung zum Client zu initiieren, was im diesem Umfeld normalerweise nicht möglich ist; dadurch entfällt lästiges Polling oder Warten. Die verwendete Implementierung im Projekt nennt sich phpws [20].

6.2.2. Struktur und Konfiguration

Wie bereits im vorherigen Kapitel erwähnt, lehnt sich die Implementierung an das MVC-Pattern an.
Dementsprechend wurden zunächst die Log- und XML-Dateien als die Modelle des Frontends identifiziert. Letztere dienen zur Steuerung des Systems und werden entsprechend aus den Eingaben des Benutzers erstellt. Um eine saubere Schnittstelle zur LMU zu gewährleisten, werden die XML-Dateien beim lesen und schreiben gegen eine XSD-Datei validiert.
Die Verarbeitung von Log-Dateien unterliegt ebenfalls einer Besonderheit. Bei einer längeren Ausführung des Systems fällt eine große Menge an zyklischen Informationen an und die entsprechenden Log-Dateien werden zu groß, um sie im Webserver

[15] jQuery Java-Script Library, *jquery.com*
[16] DataTables jQuery Plugin, *datatables.net*
[17] Validate.js jQuery Plugin, *jqueryui.com*
[18] jQuery UI user interface, *github.com/engageinteractive/validate*
[19] Highcharts JS, *www.highcharts.com*
[20] phpws - Websockets Server and Client Library, *code.google.com/p/phpws*

komplett zu verarbeiten. Häufig ist es jedoch nur nötig bestimmte Informationen und von diesen auch nur die neusten Einträge auszulesen. Das Modell der Log-Dateien wurde aus diesem Grund dahingehend angepasst, dass die Datei rückwärts und Zeile für Zeile gelesen wird, bis der gewünschte Eintrag gefunden wurde.

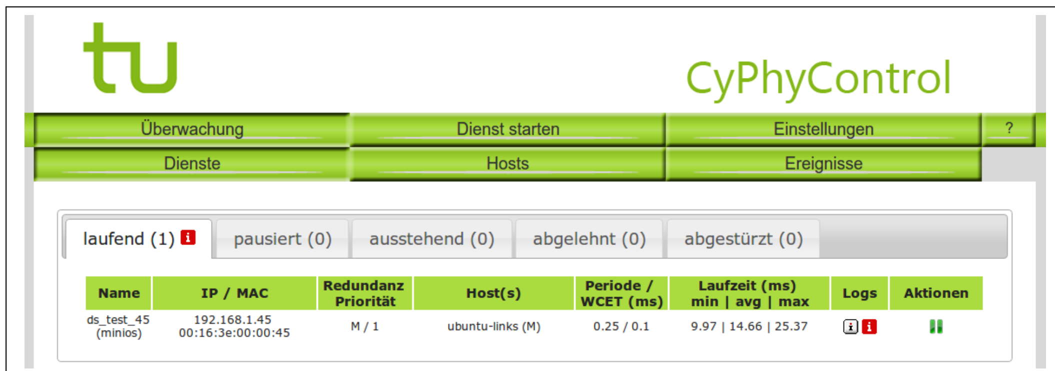


Abbildung 22: Task-Übersicht im Frontend

Eine Datenverarbeitung findet im Frontend auf fünf Unterseiten statt (siehe Abbildung 22): Dienst starten, Dienst-Überwachung, Host-Überwachung, Ereignisse und Einstellungen. Zudem gibt es eine Start- und Hilfe-Seite. Zu jeder dieser Seite wurde ein entsprechender Controller und ein View (Template) angelegt. Des weiteren gibt es noch einige Hilfstemplates, da Smarty die Verschachtelung von Templates erlaubt, und einige Hilfscontroller, welche der ausschließlich Verarbeitung von Anfragen per Ajax dienen. Ein grober Aufbau der Struktur ist in Abbildung 23 zu sehen.

Das Frontend wurde zudem konfigurierbar gemacht, um eine minimale Portierbarkeit zu garantieren. Entsprechende Verzeichnisse und die Adresse, unter welcher das Frontend aufgerufen werden kann, können über eine zentrale Datei geändert werden. Dies ermöglicht z.B. auch das externe hosten des Frontends außerhalb der entwickelten Architektur. Es muss lediglich ein Zugang zum NAS gewährleistet werden.
Eine Beschreibung der Verzeichnisstruktur befindet sich im Anhang unter Abschnitt E. Dort ist auch ein Beispiel der Konfiguration zu finden.

6.3. Load Management Unit

(Sebastian Struwe, Hülya Kaplan)

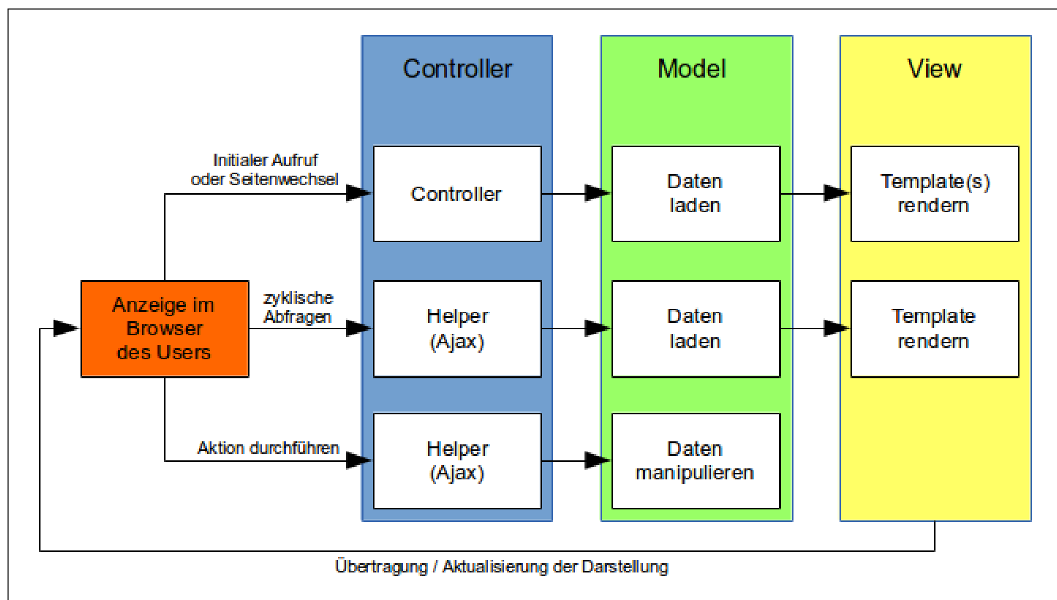


Abbildung 23: Strukturentwurf des Frontends

Die Load Management Unit (kurz LMU) hat die Aufgabe ein Scheduling zu finden, so dass die Echtzeitbedingungen von Diensten gewährleistet werden. Dazu nimmt sie Befehle von dem User Interface entgegen und leitet diese an die entsprechenden Hosts weiter. Des Weiteren reagiert sie auf Dienst- oder Hostausfälle.
Dieses Kapitel ist in drei Unterkapitel geteilt. In dem ersten Kapitel werden die Schnittstellen der LMU behandelt, wobei zwischen eingehenden und ausgehenden Schnittstellen unterschieden wird. Das zweite Kapitel beschreibt die Komponenten der LMU, hierbei werden deren Aufgaben in drei Teile unterteilt: dem Einlesen der Daten, dem Verarbeiten und der Ausgabe von Daten oder Befehlen. Das dritte Kapitel handelt von der Funktionsbeschreibung der LMU, hier werden verschiedene Funktionen vorgestellt.

6.3.1. Schnittstellenbeschreibung

In diesem Kapitel werden die verschiedenen Schnittstellen der LMU vorgestellt. Die Schnittstellen werden in eingehende und ausgehende Kommunikation unterteilt.

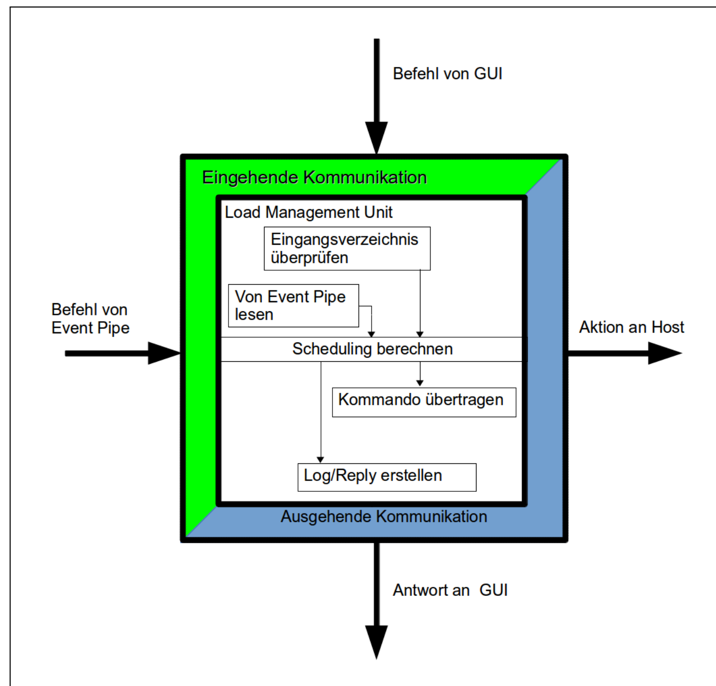


Abbildung 24: Übersicht der LMU

Eingehende Schnittstellen Die erste eingehende Schnittstelle liest die vom Webserver erzeugten XML Dateien ein. Damit die Kommunikation mit dem Webserver korrekt funktioniert, muss eine bestimmte Ordnerstruktur eingehalten werden (siehe Abbildung 25).

Um neue Anfragen zu signalisieren wird vom Webserver eine *action.xml-Datei* in die LMU_INBOX gelegt. Eine Antwort wird in dem Ordner LMU_OUTBOX erwartet. Für die Anfragen und Antworten wird ein speziell definiertes Format nach dem XML Schema verwendet. Der Verzeichnisscanner überprüft regelmäßig die LMU_INBOX auf neue *action.xml-Dateien*. Sie beinhaltet vom User initialisierte Befehle, die nun vom Verzeichnisscanner weiter verarbeitet werden. Für den Fall, dass ein neuer Dienst gestartet werden soll, wird neben der *action.xml-Datei* noch eine *task.xml-Datei* erwartet, die Informationen über den Task enthält. Um eine Vielzahl von *task.xml-Dateien* organisieren zu können, wird auch hier eine spezielle Ordnerstruktur verwendet: task.xml Dateien, die auf ihren Start warten, werden in dem Ordner pending abgelegt. Wenn der Dienst erfolgreich gestartet wurde, wird die *task.xml-Datei* nach running verschoben. Für den Fall, dass das Starten nicht erfolgreich war, wird die *task.xml-Datei* nach refused verschoben. Nachdem die Daten eingele-

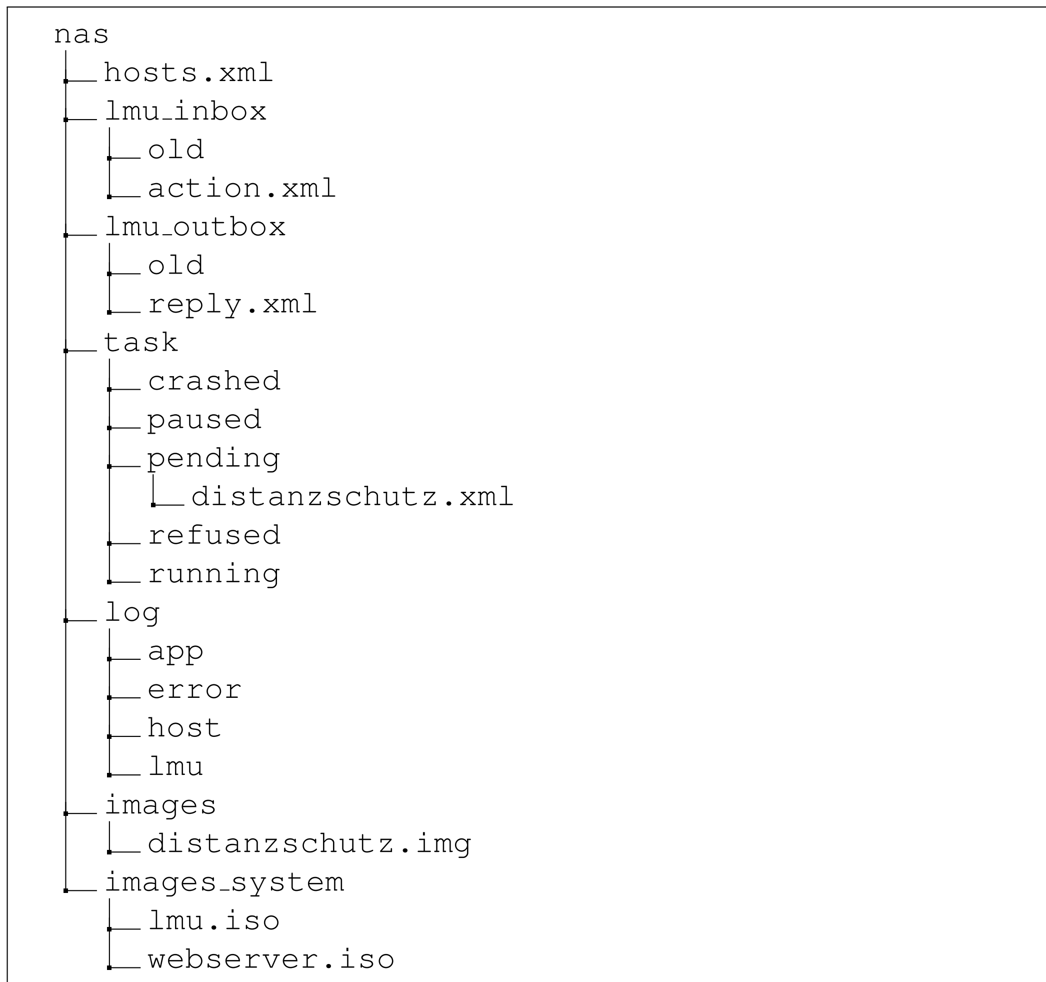


Abbildung 25: Ordnerstruktur des nas

sen wurden, werden Sie an den *scanadapter* übergeben.

Die zweite eingehende Schnittstelle besteht aus einer *Event-Pipe*. Eine Pipe ist ein Datenstrom zwischen zwei Prozessen. Aus dieser Pipe werden Informationen über fehlerhaftes Verhalten von VMs oder Hosts sowie über andere Ereignisse, wie z.B. das Hinzufügen eines Hosts gelesen. Diese Informationen werden vom globalen und lokalen Monitoring zur Verfügung gestellt. Die *Event-Pipe*enthält zeitkritischere Informationen als die Anfragen des Webservers. Daher werden auch zuerst alle Informationen aus der Pipe gelesen, bevor Informationen vom Webserver verarbeitet werden.

Ausgehende Schnittstellen Die erste ausgehende Schnittstelle beschreibt die Kommunikation zwischen LMU und Webserver. Wie im ersten Teil beschrieben, wird vom Benutzer eine Aktion initiiert. Auf diese Aktion folgt, nachdem sie ausgeführt wurde, eine Antwort. Diese wird in Form einer *reply.xml-Datei* generiert. Typischerweise sind hierbei zwei verschiedene Antworten möglich. Die erste Antwort meldet die erfolgreiche Ausführung. Falls jedoch der Befehl nicht erfolgreich ausgeführt wurde, wird eine umfangreiche Nachricht zurückgegeben, um einen möglichst detaillierten Ablauf des Fehlers zu beschreiben. Eine positive Nachricht wird auch bei Zwischenfehlern gegeben, d.h. wenn z.B. ein Host nicht erreichbar ist, der Dienst jedoch trotzdem erfolgreich gestartet werden konnte. Für Aktionen, die über die Pipe initialisiert wurden, werden keine Antworten an den Webserver generiert. Es wird nur auf Befehle geantwortet, die vom User initialisiert wurden.
Eine weitere Möglichkeit, mehr über die Abarbeitung eines Befehls zu erfahren, bietet das Logging. Neben dem Logeintrag selber wird ein Abbild der aktuellen Daten der LMU erzeugt. Damit lassen sich getroffene Entscheidungen nachvollziehen und verifizieren.

Die zweite ausgehende Schnittstelle sendet Befehle an die Hosts. Hierbei wird eine Pipe beim Starten der LMU initialisiert und über diese werden die Befehle übertragen. Nachdem eine Befehl gesendet wurde, wird ein Rückgabewert erwartet. Dieser enthält Informationen darüber, ob ein Fehler aufgetreten ist, und wie die Fehlermeldung ist.

6.3.2. Komponentenbeschreibung

In diesem Kapitel werden die einzelnen Komponenten der LMU in drei Bereiche unterteilt und beschrieben.

Daten einlesen und vorbereiten Die erste Komponente liest die Daten und Befehle ein und bereitet die Daten für den Scheduler vor. Dies übernimmt in der Implementierung der Verzeichnisscanner, der in einer Endlosschleife läuft und die LMU_INBOX auf neue *action.xml-Dateien* überprüft. Eine beispielhafte *action.xml-Datei* ist in Listing 1 zu sehen. Die *action.xml-Datei* ist immer nach demselben Sche-

ma aufgebaut: zwischen den command Tags steht der Befehl, der ausgeführt werden soll. In dem Beispiel soll ein Dienst gestartet werden. Der Name des Dienstes steht zwischen den target_name Tags.

Listing 1: action.xml-Datei

```
<?xml version="1.0"?>
<action>
  <command>start_task</command>
  <target_name>Distanzschutz</target_name>
</action>
```

Um einen Dienst zu starten, werden jedoch noch weitere Informationen benötigt. Diese befinden sich in der *task.xml-Datei*. Listing 2 zeigt zu der *action.xml-Datei* die dazugehörige *task.xml-Datei*

Listing 2: task.xml-Datei

```
<?xml version="1.0"?>
<task>
  <name>Distanzschutz</name>
  <os>ciao</os>
  <redundancy>1</redundancy>
  <period>200</period>
  <wcet>100</wcet>
  <ip>192.168.1.91</ip>
  <mac>00:16:3E:00:00:91</mac>
  <imagPath>/home/nas/images/distanzschutz.img</imagPath>
  <createdOn>20140225152555</createdOn>
</task>
```

Die *task.xml-Datei* enthält alle Informationen, die nötig sind um den Dienst zu starten. Nachdem der Verzeichnisscanner die Daten eingelesen hat, übergibt die „Rohdaten" an den *scanadapter*. Dieser bereitet die Rohdaten auf, indem er aus den einzelnen Strings ein VM Objekt erstellt und dieses dem Scheduler übergibt.

Neben der LMU_INBOX wird die *Event-Pipe* regelmäßig auf neue eintretende Events überprüft. Falls ein neues Event vorhanden ist, werden ähnlich wie bei den xml Dateien, die Daten vom *scanadapter* für den Scheduler vorbereitet. Dazu wird, falls erforderlich, das passende VM Objekt aus der internen Datenstruktur gesucht.
Durch die Trennung von Verzeichnisscanner und Scheduler war eine parallele Entwicklung und Implementierung möglich.

Globaler Scheduler Die zweite Komponente ist der Scheduler der LMU. Dieser bekommt die Daten vom *scanadapter* übergeben und führt die angeforderte Aktion aus. So sucht er z.B. einen Platz für eine neue VM und gibt den Befehl zum Starten an die dritte Komponente weiter. Dabei überprüft der Scheduler, ob genügend Ressourcen auf dem Rechner vorhanden sind. Ein weiteres Kriterium, das überprüft wird, ist die Redundanz. Falls ein Dienst als Master und Backup gestartet werden soll, wird die Backup VM nicht auf demselben Host gestartet, auf dem die Master VM läuft. Außerdem reagiert der Scheduler auf plötzlich auftretendes Fehlverhalten, wenn z.B. beim Starten einer VM auf einem Host nicht alle Skripte laufen, oder das Starten an sich nicht erfolgreich verläuft. Dann reagiert der Scheduler und versucht die VM auf einem anderen Host zu starten. Falls das Starten eines Dienstes nur teilweise erfolgreich war, z.B. nur eine Master VM erzeugt wurde, da das Starten einer Backup VM nicht erfolgreich verlief, wird auch die Master VM wieder beendet.

Transmittcommand Die dritte Komponente besteht aus der *Transmittcommand-Bibliothek*, der Logfunktion und dem Erstellen von replys. Die *Transmittcommand-Bibliothek* ist für die Kommunikation mit den Hosts verantwortlich. Das bedeutet, dass der Scheduler eine Funktion in der *Transmittcommand-Bibliothek* mit bestimmten Parametern aufruft. Diese überträgt den Befehl an die Hosts, die den Befehl ausführen. Nachdem die Aktion abgeschlossen ist, gibt die *Transmittcommand-Bibliothek* eine Nachricht und eine Bool an den Scheduler zurück, der beschreibt, ob die Ausführung korrekt war. Falls das nicht der Fall ist, wird in der Nachricht die Ursache für die nicht korrekte Ausführung des Befehls formuliert. Um die Aktionen der LMU besser verstehen und nachvollziehen zu können gibt es die Möglichkeit, einen Log erstellen zu lassen. Listing 3 zeigt den Aufbau der Log-Datei. Der erste Eintrag

ist ein Zeitstempel, der notwendig ist, da zu jedem Eintrag im Log ein Abbild der internen Datenstruktur (Heap, siehe 6.3.3) erstellt wird. Mithilfe des Abbilds und des Logs, können die Entscheidungen der LMU nachvollzogen und optimiert werden. In dem Beispiel wurde überprüft, ob ein Host erreichbar ist. Da er erreichbar ist, wird er nun der internen Datenstruktur hinzugefügt.

Listing 3: Beispiel eines Log Eintrags

```
1393343188.9226234: Der Host ist erreichbar. Er wird dem
    Heap hinzugefuegt.
```

Das Listing 4 zeigt die Antwort der LMU auf das Starten eines Tasks. Dabei kann jede *reply.xml-Datei* einer *task.xml-Datei* zugeordnet werden, da der Timestamp eindeutig ist und sich auf den Erstellungszeitpunkt der *task.xml-Datei* bezieht. Zwischen den Tags <result >wird die vom Scheduler erzeugte Nachricht angezeigt.

Listing 4: reply.xml-Datei

```
<reply>
<action_TS>20140225152555</action_TS>
<result>Der Dienst wurde erfolgreich gestartet.</result>
</reply>
```

6.3.3. Funktionsbeschreibung

In diesem Kapitel werden die wichtigsten Funktionen der LMU vorgestellt. Dazu muss zunächst die interne Datenstruktur, der Heap sowie Host und VM Objekte, erklärt werden.

Heap Der Heap ist die zentrale Datenstruktur der LMU. Er ist eine Liste, die aus Tupeln von dem am geringsten ausgelasteten Kern eines Hosts und dem Host selber besteht.

Listing 5: Aufbau des Heaps

```
1 ((0.31, ubuntu-links), (0.828, ubuntu-rechts))
```

Ein Host Objekt beinhaltet alle Informationen, die den Host betreffen. In der Implementierung sind dies der Name des Hosts, die IP Adresse und die Anzahl der Kerne. Außerdem besitzt jeder Host eine Liste mit VM Objekten. Auch diese Liste besteht aus Tupeln, die den Kern beschreiben, auf dem die VM läuft sowie das eigentliche VM Objekt selber. Ein VM Objekt besteht aus Name, WCET, Periode, Status und Redundanz. Der interne Status beschreibt, ob das VM Objekt eine Master, Backup oder Node auf dem Host repräsentiert. Hierbei nimmt die Node noch einen besonderen Platz ein, denn sie dient der LMU nur als Platzhalter und wird nicht als VM auf den Rechner gestartet (siehe 5.2).
Der Vorteil des Heaps gegenüber einer normalen Liste besteht neben dem Geschwindgkeitsvorteil auch darin, dass durch das Auswählen des ersten Objekts im Heap automatisch das Worst-Fit Verfahren angewendet wird.

Initialer Start der LMU Der erste Start der LMU läuft folgendermaßen ab: Nach dem Hochfahren, startet das Discovery-Script eine LMU. Sie initialisiert ein Socket und wartet, dass sie vom Discovery die IP von dem Rechner mitgeteilt bekommt, auf dem sie sich aktuell befindet. Wenn sie diese Daten empfangen hat, trägt die LMU sich selber in den Heap ein. Nach und nach empfängt die LMU über die *Event-Pipe* Benachrichtigungen welche Hosts zur Verfügung stehen. Diese werden dann, falls sie außerdem in der *hosts.xml-Datei* stehen, dem Heap hinzugefügt. Nachdem nun die erste VM der LMU gestartet wurde, wird nun versucht ein Backup zu erzeugen. Dies setzt voraus, dass ein zweiter Host verfügbar ist. Falls das möglich ist, wurde die LMU korrekt gestartet. Falls es nicht möglich ist ein Backup zu starten, läuft die LMU so lange als Master, bis ein weiterer Host dem Heap hinzugefügt wird. Danach wird, wenn möglich, ein Backup der LMU auf dem Host gestartet. Nachdem die LMU gestartet wurde, startet sie den Webserver, der auch wie jeder andere normale Dienst behandelt wird.

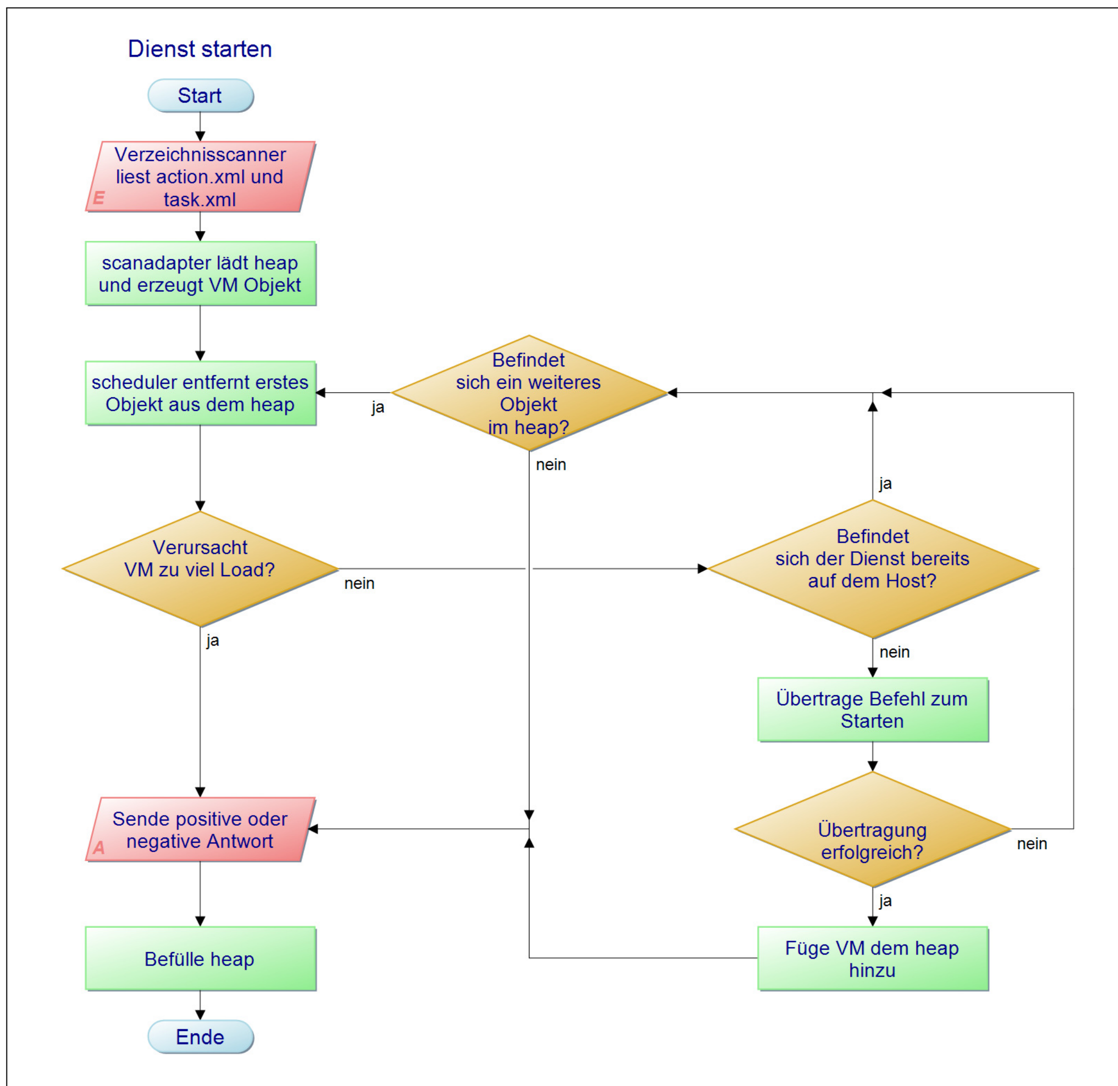


Abbildung 26: Starten eines Dienstes

Starten eines Dienstes Der Programmablaufplan in Abbildung 26 zeigt das Starten eines Dienstes. Dabei wurde davon ausgegangen, dass der Dienst als Master gestartet werden sollte, also ohne Redundanzen. Beim Starten eines Dienstes liest der Verzeichnisscanner die *action.xml-Datei* und *task.xml-Datei* ein und übergibt die Informationen dem *scanadapter*. Dieser bereitet die Daten vor und übergibt dem Scheduler ein VM Objekt. Dieser nimmt nun das erste Host Objekt aus dem Heap und überprüft, ob die VM zu viel Last verursacht, also die Auslastung eines Kerns größer als 0,69 ist (siehe 6.5.1). Falls das der Fall ist, kann der Dienst direkt abgelehnt werden, da es keinen Host gibt, der mehr freie Last zur Verfügung hat. Falls die VM nicht zu viel Last verursacht, wird überprüft, ob sich bereits eine VM dieses Dienstes auf dem Host befindet. Falls das nicht der Fall ist, wird der Befehl zum Starten einer VM über die *Transmittcommand-Bibliothek* übertragen. Kommt es dabei zu Problemen, die Antwort also negativ ist, wird ein weiterer Host aus dem Heap gesucht. Nachdem das Kommando erfolgreich übertragen wurde, wird die VM dem Heap hinzugefügt. Als Abschluss wird noch eine Antwort für den Webserver generiert und der Heap wieder mit den Hostobjekten und der aktualisierten minimalen Auslastung neu befüllt.

Stoppen eines Dienstes Das Stoppen eines Dienstes wird durch den Programmablaufplan in Abbildung 27 dargestellt. Hierbei wurde als Beispiel eine VM ausgesucht, die als Master, Backup und Node gestartet wurde. Zuerst werden die Informationen vom Verzeichnisscanner eingelesen und vom *scanadapter* weiter verarbeitet. Dieser gibt ein VM Objekt an den Scheduler weiter. Der Scheduler sucht nun zu dem VM Objekt den Host, auf dem die Master VM läuft. Wenn er den Host gefunden hat, sendet er den Befehl zum Beenden einer VM an diesen. Nachdem der Befehl gesendet wurde, wird die VM aus der Datenstruktur entfernt. Nun wird der Host gesucht, auf dem sich die Backup VM befindet. Wenn der Host gefunden wurde, wird der Befehl wie bei der Master VM übertragen. Anschließend wird die Backup VM aus der Datenstruktur gelöscht. Es ist wichtig, dass das Beenden in dieser bestimmten Reihenfolge abläuft, da sonst vom Monitoring ein Fehler, genauer ein VM_DOWN gemeldet wird (siehe 6.4). Als Letztes wird die Node gesucht. Da diese nur in der Datenstruktur vorhanden ist, reicht es die Node aus dieser zu entfernen. Nachdem alle VMs beendet wurden, kann der Dienst als beendet angesehen werden und es wird eine Nachricht für den Webserver generiert. Abschließend wird der Heap mit

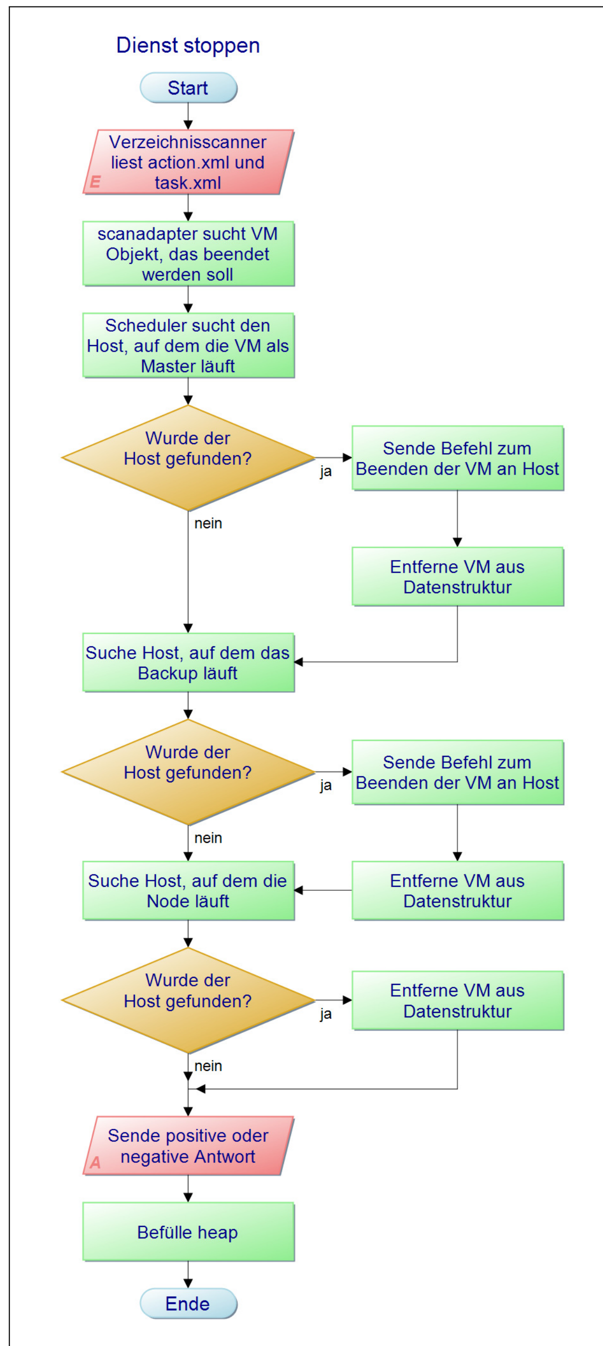


Abbildung 27: Stoppen eines Dienstes

aktualisierten Auslastungen neu befüllt.

Host herunterfahren Um einen Host herunterzufahren, wird vom *scanadapter* das Hostobjekt aus dem Heap gesucht und dem Scheduler übergeben. Dieser überprüft nun für jede VM, die sich auf dem Host befindet, mit welcher Redundanz sie gestartet wurde. Falls sie nur als Master gestartet wurde, wird die VM auf einen anderen Host migriert, natürlich unter der Voraussetzung, dass die VM den Host nicht überlastet. Falls es sich bei dem Dienst um einen Master und Backup Dienst handelt, kann keine Migration erfolgen. Daher wird hier noch mal eine Unterscheidung vorgenommen. Für den Fall, dass es sich bei der zu verschiebenden VM um eine Master VM handelt, wird diese beendet. In diesem Moment übernimmt das Backup. Dieses wird nun in der Datenstruktur gesucht und der Status von Backup auf Master geändert. Als letzter Schritt wird ein neues Backup von dem Master auf einem anderen Host erzeugt. Falls dies nicht möglich ist, wird kein Backup erzeugt und die Master VM läuft weiter. Für den Fall, dass die VM, die sich auf dem zu herunterfahrenden Host befindet, eine Backup VM ist, wird diese beendet und auf einem anderen Host neu gestartet. Falls der Dienst als Master, Backup und Node gestartet wurde, wird als erster Schritt die Node gelöscht. Nun wird für den Dienst, der verschoben werden soll, ein neuer Platz gesucht. Dieser muss nicht zwangsläufig der sein, den vorher die Node hatte. Ab diesem Zeitpunkt ist das Verhalten analog zu dem vorher beschriebenen Fall. Als letzter Schritt wird wieder ein Platz für die Node reserviert.

Host hinzufügen Um einen Host während des Betriebs hinzuzufügen, muss die host.xml aktualisiert werden. Wenn das über die Website geschieht, wird eine *action.xml-Datei* erzeugt, die das erneute Einlesen der host.xml triggert. Hierbei werden alle Hosts zum Heap hinzugefügt, die erreichbar sind. Dies wird durch eine Funktion vom LMU Interface realisiert. Nachdem die Hosts zum Heap hinzugefügt wurden, wird überprüft, ob alle erforderlichen Redundanzen vorhanden sind. Dazu wird der Heap durchlaufen und überprüft, ob sich alle VMs eines Dienstes, der z.B. als Master, Backup und Node gestartet werden sollte, auch so im Heap wiederfinden. Ist dies nicht der Fall, werden die fehlenden Redundanzen hinzugefügt.

VM Down Das Event VM Down wird generiert, falls eine VM beendet bzw. abgestürtzt ist. Das Event wird über die *Event-Pipe* dem Verzeichnisscanner mitgeteilt, der die Informationen enthält, welchen Namen die VM hatte und auf welchem Rechner die VM lief. Mit Hilfe dieser Informationen sucht der *scanadapter* das korrekte VM Objekt und übergibt es dem Scheduler, der nun überprüft , um welche Art von Dienst es sich handelt. Falls es sich um einen Dienst handelt, der als Master, Backup und Node gestartet wurde, wird zuerst die Node gelöscht. Wenn die Master VM abgestürzt ist, wird die Backup VM gesucht. Von dieser wird der Status auf Master gesetzt, anschließend wird für das neue Backup ein Platz gesucht und die Backup VM gestartet. Als Letztes wird die Node der Datenstruktur hinzugefügt.

Für den Fall, dass die Backup VM abgestürzt ist, wird ebenfalls zuerst die Node beendet. In einem nächsten Schritt wird ein Platz für die neue Backup VM gesucht. Wenn dieser gefunden ist, wird die Node der Datenstruktur hinzugefügt.

Für den Fall, dass die VM nur als Master gestartet wurde, wird für sie ein neuer Platz gesucht, und die VM wird dort gestartet (siehe 5.2).

mkr ist eine Funktion (make room) , die bei jedem Dienststart ausgeführt wird. Sie überprüft, ob die VMs des neuen Dienstes ohne Weiteres auf einen der verfügbaren Hosts passen. Ist das der Fall, so wird diese Funktion beendet und der Dienst gestartet. Sollten aber alle Hosts bereits so befüllt sein, dass sie die neuen VMs nicht mehr aufnehmen können, dann versucht das mkr durch Verschieben bereits laufender VMs genug Platz zu finden.
Hierfür holt sich die Funktion die Informationen über aktuell laufende Hosts und VMs aus dem Heap. Für jeden Kern wird ein Tupel aus Host, Kern, Kernauslastung und VM-Liste angelegt. Die VM-Liste besteht aus allen auf diesem Kern laufenden VMs und ist aufsteigend nach der Auslastung sortiert, die die jeweilige VM auf dem Kern verursacht. Anschließend werden auch die Tupel aufsteigend nach Kernauslastung sortiert und zu einer verketteten Liste zusammengelegt. mkr beginnt nun mit dem ersten Element der Liste, dem am wenigsten ausgelasteten Kern, und überprüft, ob die neue VM hier drauf passt. Falls nicht werden der Reihe nach die VMs dieses Kerns auf andere Ziel-Kerne migriert. Die Suche nach einem Ziel-Kern beginnt vom

letzten Listenelment aus, also dem Kern, der am meisten ausgelastet ist. Passt die zu migrierende VM nicht auf diesen Ziel-Kern, so geht die Funktion ein Element weiter und arbeitet sich durch die gesamte Liste. Sobald die VM migriert werden kann, wird überprüft, ob die neue VM nun auf den Anfangs-Kern passt. Wenn noch immer nicht genügend Platz vorhanden ist, werden weitere VMs verschoben, bis die neue VM passt oder keine VMs mehr migrierbar sind. Diesen Vorgang wiederholt die Funktion bei jedem einzelnen Element der Liste und fertigt dabei einen Ablaufplan der Migrationen an. Erst wenn alle VMs des neuen Dienstes Platz auf den Kernen finden, wird nach Plan migriert und der Dienst gestartet. Wurde allerdings trotz Migrationen nicht genug Platz gefunden, so wird der neue Dienst abgelehnt.

reschedule dient der Ausbalancierung der Auslastungen auf den Kernen. Diese Funktion soll gestartet werden, nachdem ein neuer Host hinzugefügt oder die Funktion mkr aufgerufen wurde. Denn in beiden Fällen kann es vorkommen, dass einige Kerne völlig ausgelastet sind, während andere vielleicht leer bleiben. Der Beginn dieser Funktion ähnelt dem der mkr Funktion. Es wird erneut der Heap aufgerufen, um daraus Tupel aus Host, Kern, Kernauslastung und VM-Liste zu bilden und diese nach ihren Kernauslastungen aufsteigend zu sortieren. Mit dieser entstehenden verketteten Liste arbeitet das reschedule. Es beginnt am Ende der Liste und sucht für die VMs, des am meisten ausgelasteten Kerns, Platz auf einem anderen Ziel-Kern. Den Ziel-Kern sucht sich die Funktion am Anfang der Liste und geht bei Misserfolg immer ein Element weiter. Sämtliche Migrationen werden auch hier erst ein Mal in einem Ablaufplan gesammelt und am Ende hintereinander abgearbeitet. Dabei gilt eine VM nur dann migrierbar, wenn der Ziel-Kern nach der Migration weniger ausgelastet ist als der vorherige Kern vor der Migration. Natürlich werden dabei auch die Redundanzen beachtet, damit diese nicht auf demselben Host laufen. Leider wurde diese Funktion noch nicht ausgiebig getestet und ist daher noch im System inaktiv.

6.4. Monitoring

(Tim Harde) Im folgenden Kapitel wird die Implementierung des Monitoring-Systems beschrieben. Das Kapitel gliedert sich in zwei Unterkapitel: Schnittstellenbeschreibung

und Funktionsbeschreibung. In der der Schnittstellenbeschreibung wird zunächst die Implementierung der einzelnen Systemschnittstellen und die konkrete Interaktion mit anderen Modulen des Gesamtsystems beschrieben; die Funktionsbeschreibung bildet den Abschluss dieses Kapitels - hier wird auf die Implementierung der wesentlichen Aufgaben und Abläufe eingegangen.

6.4.1. Schnittstellenbeschreibung

Insgesamt verfügt das Monitoring-System über fünf Schnittstellen, die im folgenden Kapitel näher beschrieben werden.

LMU-Interface - Lokales Monitoring Die Schnittstelle zwischen dem LMU-Interface und dem lokalen Monitoring dient dazu, den Zustandsautomaten des lokalen Monitorings bei Veränderung des Systemzustands zu aktualisieren, damit dieses die einzelnen Applikationen überwachen und korrekt auf Zustandsänderungen reagieren kann. Die konkreten Zustände und Zustandsübergänge werden in Kapitel 6.4.2 näher erläutert.

Hypervisior - Lokales Monitoring Die Schnittstelle zwischen Hypervisor und lokalem Monitoring erfüllt zwei Aufgaben: Überwachung der Applikationslaufzeiten und Erkennung von WCET- und Deadline-Überschreitungen.

Die relevanten Informationen werden vom Scheduler direkt in den *Xend Message Buffer* geschrieben. Damit kritische Meldungen in diesem Puffer nicht durch Polling oder Spinning ermittelt werden müssen, wird bei solchen Events (wie beispielsweise WCET- oder Deadline-Überschreitungen) ein virtueller Interrupt (VIRQ) ausgelöst. Dieser virtuelle Interrupt wird anschließend vom Monitoring-Sytem abgefangen und weiter behandelt. Auf diese Art und Weise ist eine effiziente Verarbeitung der bereitgestellten Informationen möglich. Die konkreten Abläufen werden in Kapitel 6.4.2 näher beschrieben.

Lokales Monitoring - Globales Monitoring Die Kommunikation zwischen lokalem und globalem Monitoring erfolgt über TCP-Sockets. Hierbei werden generell zwei Aspekte unterschieden: zyklische Zustandsinformationen und Fehlernachrichten.

Die zyklischen Zustandsinformationen (siehe Anhang X) werden vom lokalen Monitoring-Skript erhoben. Die eigentliche Erhebung der Daten erfolgt in einer Endlosschleife, nach einem konfigurierbarem Intervall werden alle aggregierten Zustandsinformationen (Standardintervall: fünf Sekunden) an das globale Monitoring versendet. Durch das Ausbleiben der zyklischen Zustandsinformationen kann zusätzlich der Ausfall eines Virtualisierungsservers oder der Absturz der lokalen Monitoring-Komponente erkannt werden.

Fehlernachrichten (beispielsweise Absturz einer VM oder WCET- bzw. Deadline-Überschreitung) werden unabhängig von diesem Intervall an das globale Monitoring verschickt, damit die Verzögerung zwischen Auftreten des Ereignisses und Fehlerbehandlung minimiert wird. Auf die Interaktion zwischen lokalem und globalem Monitoring wird in Kapitel 6.4.2 noch näher eingegangen.

Globales Monitoring - LMU Beide Komponenten (globales Monitoring und LMU) werden in der gleichen virtuellen Maschine ausgeführt. Die Kommunikation erfolgt über die sogenannte *Event Pipe*. Diese Pipe ist dabei nicht blockierend, damit die LMU beim Zugriff auf diese nicht fälschlicherweise blockiert und so weiterhin mit dem Benutzer interagieren kann. Über die Pipe werden für die LMU relevante Informationen propagiert.

Im Fehlerfall werden die vom lokalen Monitoring generierten Fehlernachrichten vom globalen Monitoring verarbeitet und die LMU über die entsprechende Situation informiert. Die möglichen Kategorien sind in der nachfolgenden Tabelle aufgelistet:

- **VM_CRASH**: Absturz einer virtuellen Maschine (*STANDALONE oder REMUS_SOURCE*)
- **VM_REMUS_FAILOVER**: Remus-Replikation wurde unterbrochen (z.B. durch Host-Ausfall, Probleme mit der Netzwerkverbindung, Absturz der Master-VM, etc.), die Backup-VM hat die Aufgabe der Master-VM übernommen
- **VM_DEADLINE_MISS**: Deadline-Überschreitung einer virtuellen Maschine (*STANDALONE oder REMUS_SOURCE*)
- **VM_WCET_MISS**: WCET-Überschreitung einer virtuellen Maschine (*STANDALONE oder REMUS_SOURCE*)

- **HOST_UP**: Erkennung eines neuen Virtualisierungsservers
- **HOST_CRASH**: Ausfall eines Virtualisierungsservers

Globales Monitoring - NAS Diese Schnittstelle dient lediglich dazu, um sämtliche Zustandsinformationen und Fehlernachrichten persistent auf dem NAS abzulegen, damit diese vom Webfrontend aufbereitet und für den Benutzer angezeigt werden können.

6.4.2. Funktionsbeschreibung

Das Monitoring-System erfüllt drei wichtige Funktionen: die Überwachung und Aktualisierung der Zustände sowie die Überwachung der einzelnen Virtualisierungsserver und Applikationslaufzeiten. Die genaue Funktionsweise wird innerhalb dieses Kapitels näher beschrieben.

Aktualisierung und Überwachung der Zustände Die Motivation zur Überwachung der Zustände wurde in der Einführung zu diesem Kapitel bereits eingehend erläutert.

Abbildung 28 zeigt den Zustandsautomaten des lokalen Monitorings. In diesem Abschnitt wird dabei zunächst nur auf die vom Benutzer intendierten Zustandsänderungen eingegangen, wie diese beispielsweise beim Erstellen/Zerstören von virtuellen Maschinen, bei der Migration oder bei der Erstellung einer Remus-Replikation erfolgen. Eine neu gestartete virtuelle Maschine (Event: CREATE) befindet sich zunächst im Zustand *STANDALONE*.

Eine virtuelle Maschine kann durch eine Live-Migration auf einen anderen Host migriert werden. Hierzu werden die Montoring-Syteme der beiden Hosts vor Beginn der Migration über die anstehende Migration informiert (Event: OUTGOING_MIGRATION auf dem Quellhost bzw. INCOMING_MIGRATION auf dem Zielhost). Die virtuellen Maschinen verbleiben so lange in diesem Zustand, bis die Migration erfolgreich abgeschlossen ist. Nach abgeschlossener Migration (Event: MIGRATION_COMPLETE) wird die virtuelle Maschine auf dem Quellhost nicht weiter überwacht, auf dem Zielhost geht sie in den Zustand *STANDALONE* über.

Bei der Erstellung einer neuen Remus-Replikation (Event: OUTGOING_MIGRA-

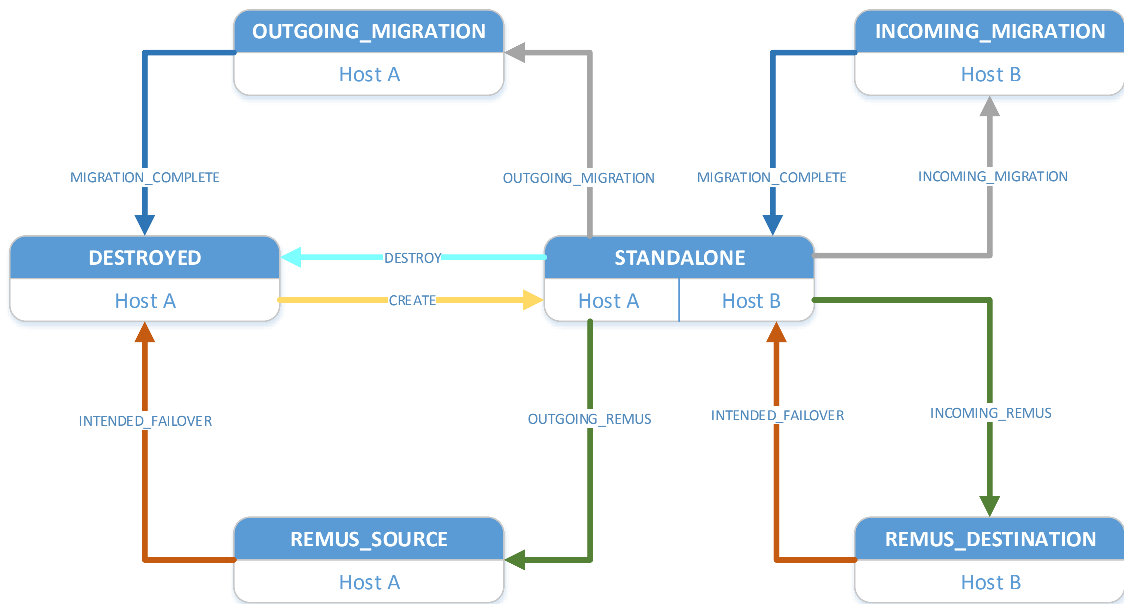


Abbildung 28: Zustandsautomat des lokalen Monitorings (Intendierte Zustandsänderung).

TION auf dem Quellhost bzw. INCOMING_MIGRATION auf dem Zielhost) geht die virtuelle Maschine in den Zustand *REMUS_SOURCE* bzw. *REMUS_DESTINATION* über; die VMs verbleiben in diesem Zustand für die gesamte Dauer der Replikation. Soll die Quell-VM zerstört werden (wenn der Server beispielsweise bei durchzuführenden Wartungszwecken heruntergefahren werden muss), so werden die beiden Monitoring-Systeme über den anstehenden Zustandswechsel informiert (Event: INTENDED_FAILOVER) - der anschließende Failover wird nicht als Fehlermeldung an das globale Monitoring beziehungsweise die LMU propagiert. Die Ziel-VM geht nach erfolgtem Failover in den Zustand *STANDALONE* über. Wird eine *STANDALONE*-VM zerstört (Event: DESTROY), so wird sie aus der Datenstruktur des lokalen Monitorings entfernt und nicht länger überwacht.

Abbildung 29 zeigt den Ablauf bei Einordnung in das Gesamtsystem. Soll von der LMU eine neue Aktion durchgeführt werden (Starten oder Migration einer VM oder Erstellung einer Remus-Replikation), so wird zunächst ein *Sanity Check* auf den betreffenden Hosts durchgeführt, um sicherzustellen, dass der Host ordnungsgemäß funktionert. Nach derzeitigem Stand wird überprüft, ob alle benötigten System-Skripte ausgeführt werden und ob die Verbindung mit Internode- und Sensornetz

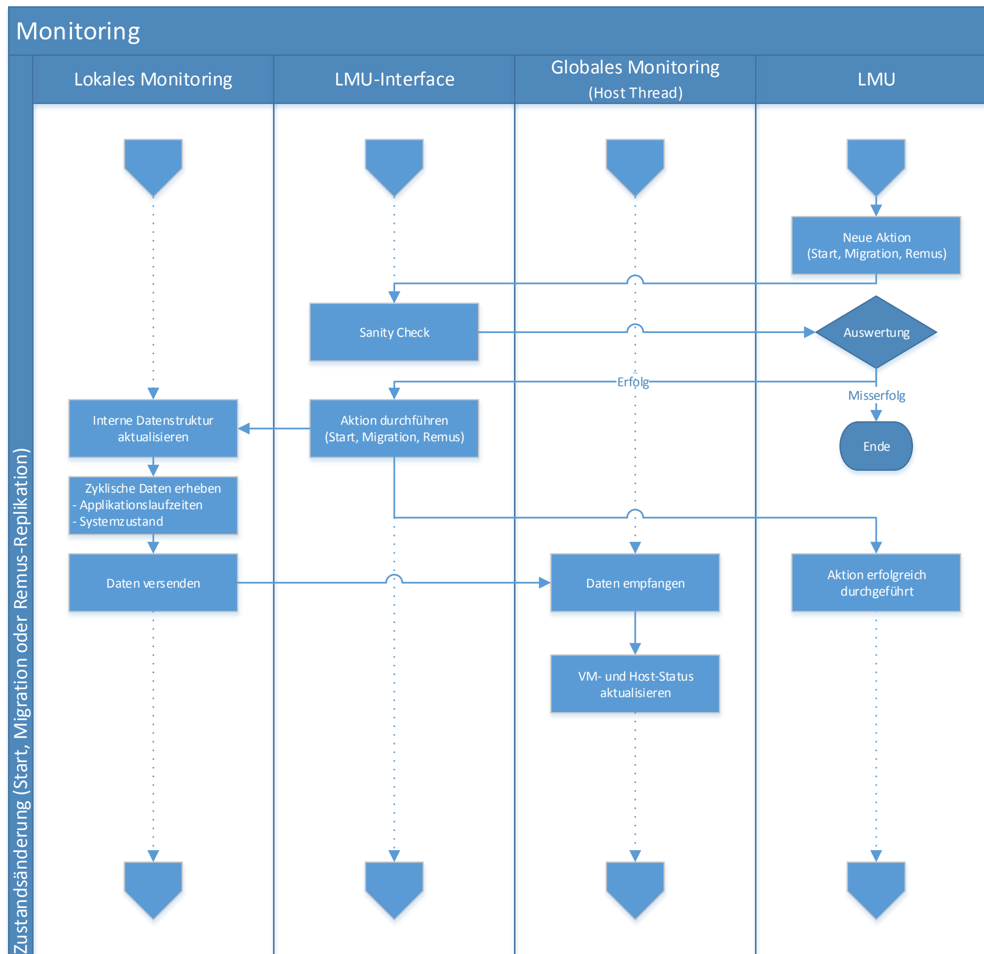


Abbildung 29: Ablauf einer intendierten Zustandsänderung.

hergestellt ist. Ist dieser erfolgreich, so wird das Kommando zur Durchführung der Aktion an das LMU-Interface weitergegeben. Das LMU-Interface informiert das lokale Monitoring über die bevorstehende Zustandsänderung und führt anschließend die Aktion durch; nach der Aktualisierung der internen Datenstruktur wird der neue Zustand an das globale Monitoring weitergeleitet. Das globale Monitoring sorgt dafür, dass die entsprechenden Log-Einträge erzeugt werden, damit der aktuelle Systemzustand in der GUI korrekt angezeigt werden kann.

Neben den intendierten Zustandsänderungen (siehe 6.4.2) kann es zu ereignisbasierten Zustandsänderungen kommen. Abbildung 30 zeigt den Zustandsautomaten, der die Zustandsübergänge zu den zwei wesentlichen Ereignisse beschreibt.

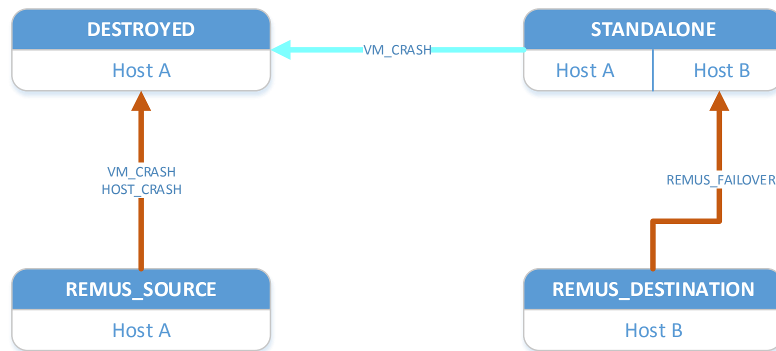


Abbildung 30: Zustandsautomat des lokalen Monitorings (Ereignisbasierte Zustandsänderung).

Wenn sich eine virtuelle Maschine im Zustand *STANDALONE* befindet, kann sie im Falle eines Absturzes in den Zustand *DESTROYED* übergehen. Wird diese Zustandsänderung vom lokalen Monitoring erkannt, so generiert dieses eine Fehlernachricht, die anschließend an das globale Monitoring übertragen wird.

Ein ähnliches Szenario ergibt sich für den Fall, dass entweder die Quell-VM einer Remus-Replikation oder der Host, auf dem sich diese befindet, abstürzt (VM_CRASH bzw. HOST_CRASH). In diesem Fall wird die Ziel-VM der Remus-Replikation aktiv; das lokale Monitoring dieses Hosts erkennt diese Zustandsänderung und informiert das globale Monitoring über dieses Ereignis (REMUS_FAILOVER). Auch an dieser Stelle soll die Einordnung in das Gesamtsystem anhand einer Abbildung (siehe Abbildung 31) näher erläutert werden.

Sobald das lokale Monitoring eine Differenz zwischen Soll- und Ist-Zustand erkennt, wird die lokale Datenstruktur aktualisiert und eine Fehlernachricht an das globale Monitoring versendet. Das globale Monitoring informiert die LMU über das entsprechende Ereignis und erzeugt die entsprechenden Log-Einträge, damit der aktuelle Systemzustand in der GUI korrekt angezeigt werden kann.

Überwachung der Hosts In diesem Kapitel soll die Überwachung der einzelnen Virtualisierungsserver näher beschrieben werden. Der Start des Monitoring-Systems (also des lokalen und globalen Monitorings) ist in Abbildung 32 zu sehen.

Zunächst werden die einzelnen Monitoring-Komponenten gestartet und initialisiert. Das lokale Monitoring sammelt daraufhin einige statische Daten (Hostname, IP-

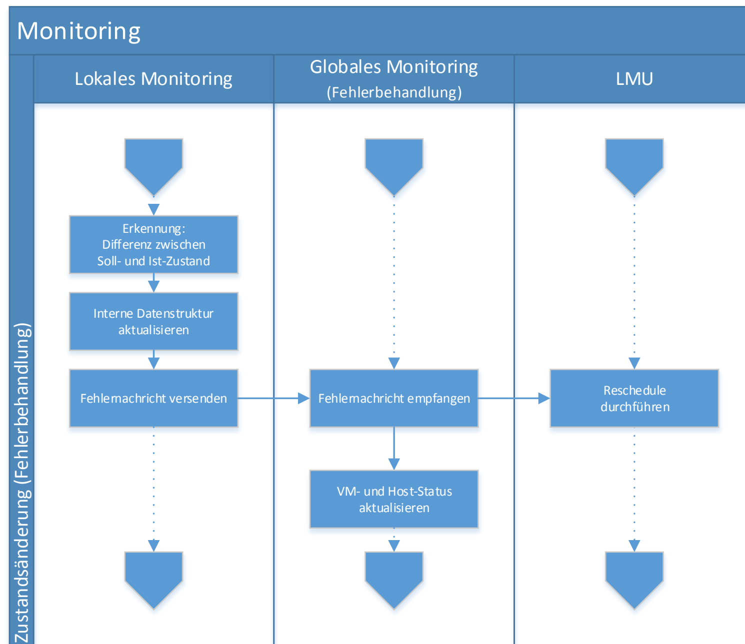


Abbildung 31: Ablauf einer ereignisbasierten Zustandsänderung.

Adressen, verfügbare Ressourcen, etc.) und schickt diese an das globale Monitoring. Nachdem diese Daten empfangen wurden, wird für den neuen Host ein eigener Thread erzeugt, in dem fortan alle zyklischen Zustandsinformationen empfangen, überprüft und verarbeitet werden.

Nach dem Empfangen der ersten zyklischen Zustandsinformation führt der erzeugte Monitoring-Thread über das LMU-Interface einen *Sanity Check* durch, um sicherzustellen, dass der neue Virtualisierungsserver voll funktionsfähig ist. Wird dieser Check erfolgreich abgeschlossen, so wird die LMU über die *Event Pipe* über einen neuen Host (Event: HOST_UP) informiert. Der Virtualisierungsserver kann nun von der LMU dazu verwendet werden, um neue Applikationen zu starten, fehlende Redundanzen (wieder-)herzustellen oder die Last des Systems zu verteilen.

Abbildung 33 zeigt das Verhalten der Monitoring-Komponenten während des regulären Betriebs. Das lokale Monitoring sammelt zyklisch Zustandsinformationen, aggregiert sie und verschickt diese daraufhin an den korrespondierenden Thread des globalen Monitorings. Dieser Thread verarbeitet die Zustandsinformationen und archiviert diese in den entsprechenden Applikations- bzw. Host-Logdateien auf dem NAS, damit jederzeit aktuelle Informationen im Webfrontend dargestellt werden

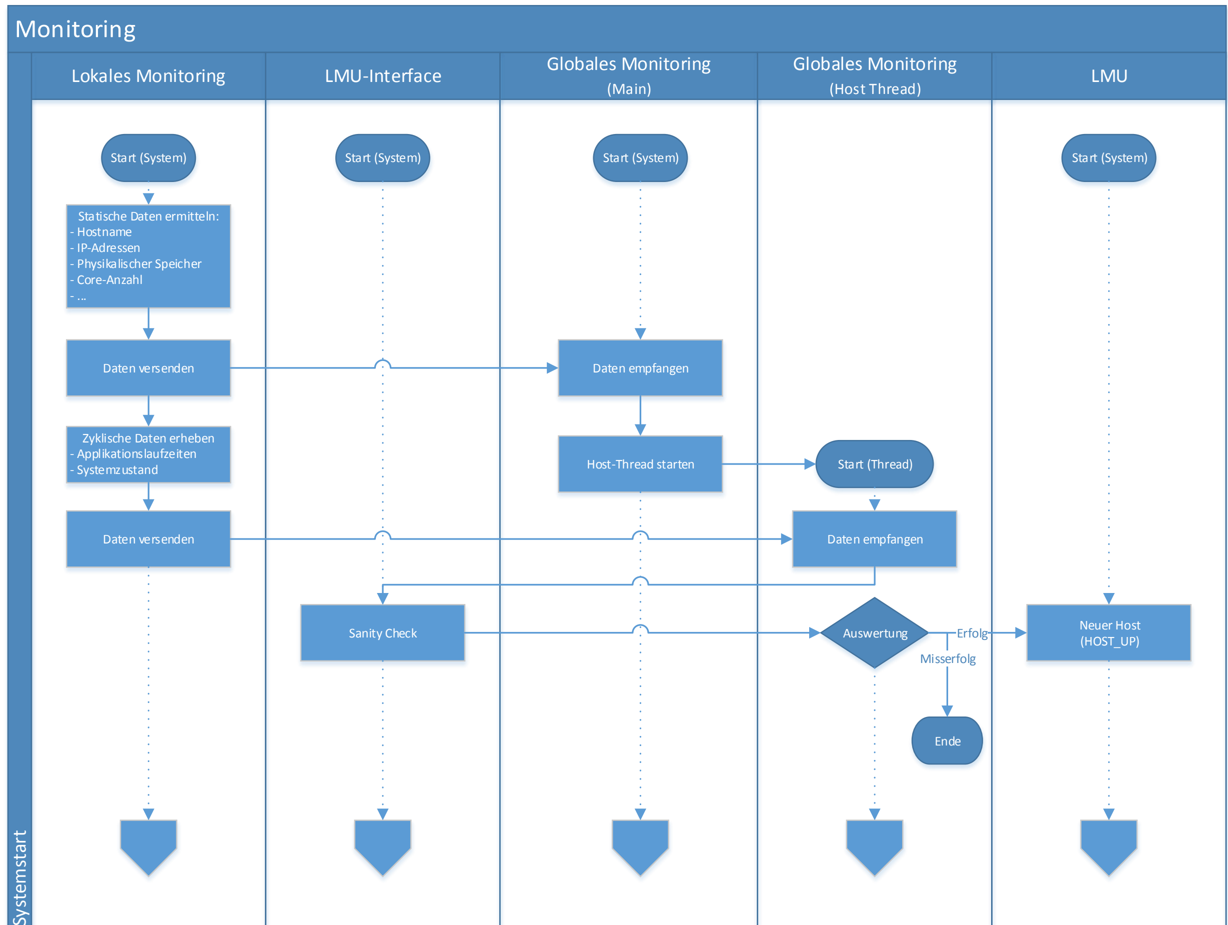


Abbildung 32: Starten des Monitoring-Systems.

können.

Des weiteren wird im Thread des globalen Monitorings mit Empfang einer neuen Nachricht ein Timer zurückgesetzt, der zur Erkennung eines Host-Ausfalls verwendet wird. Läuft dieser Timer ab, ohne dass neue Zustandsinformationen vom entsprechenden Virtualisierungsserver eingetroffen sind, so liegt möglicherweise ein Hardwaredefekt oder eine Fehlfunktion in der lokalen Monitoring-Komponente vor.

Abbildung 34 zeigt die Erkennung eines Host-Ausfalls durch das Monitoring-System. Zunächst befindet sich das Monitoring-System im regulären Betrieb. Die Zustandsinformationen werden lokal gesammelt, aggregiert und an das globale Monitoring versendet. Der Timer im Thread des globalen Monitorings wird nach dem Empfangen der Informationen zurückgesetzt und wartet daraufhin wieder auf neue Infor-

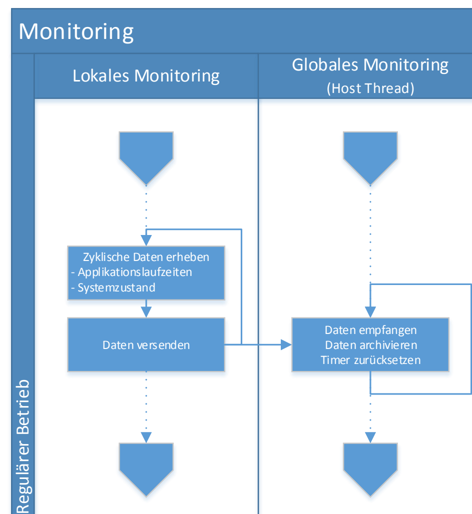


Abbildung 33: Regulärer Betrieb des Monitoring-Systems.

mationen. Kommt es nun zu einem Host-Absturz, so sorgt der Timer im Thread des globalen Monitorings dafür, dass diese kritische Situation erkannt und ein entsprechender Logeintrag erzeugt wird.

Das globale Monitoring führt daraufhin einen *Sanity Check* bei dem entsprechenden Host durch. Ist dieser erfolgreich, so geht das System wieder in den Normalzustand über. Schlägt dieser hingegen fehl, so wird die LMU über einen abgestürzten Host (Event: HOST_CRASH) informiert und kann das Rescheduling durchführen, um den Systemzustand wiederherzustellen.

Überwachung der Applikationslaufzeiten

Die Applikationslaufzeiten werden vom Scheduler zur Laufzeit erfasst. Solange die in der GUI angegebene WCET in der angegebenen Periode nicht überschritten wird, wird lediglich in regelmäßigen Abständen (Standardintervall: fünf Sekunden) die minimale, durchschnittliche und maximale Laufzeit jeder Applikation in diesem Intervall in den *Xend Message Buffer* geschrieben. Diese Informationen werden vom lokalen Monitoring in regelmäßigen Abständen (Standardintervall: fünf Sekunden) aus dem Puffer extrahiert und an das globale Monitoring weitergeleitet.

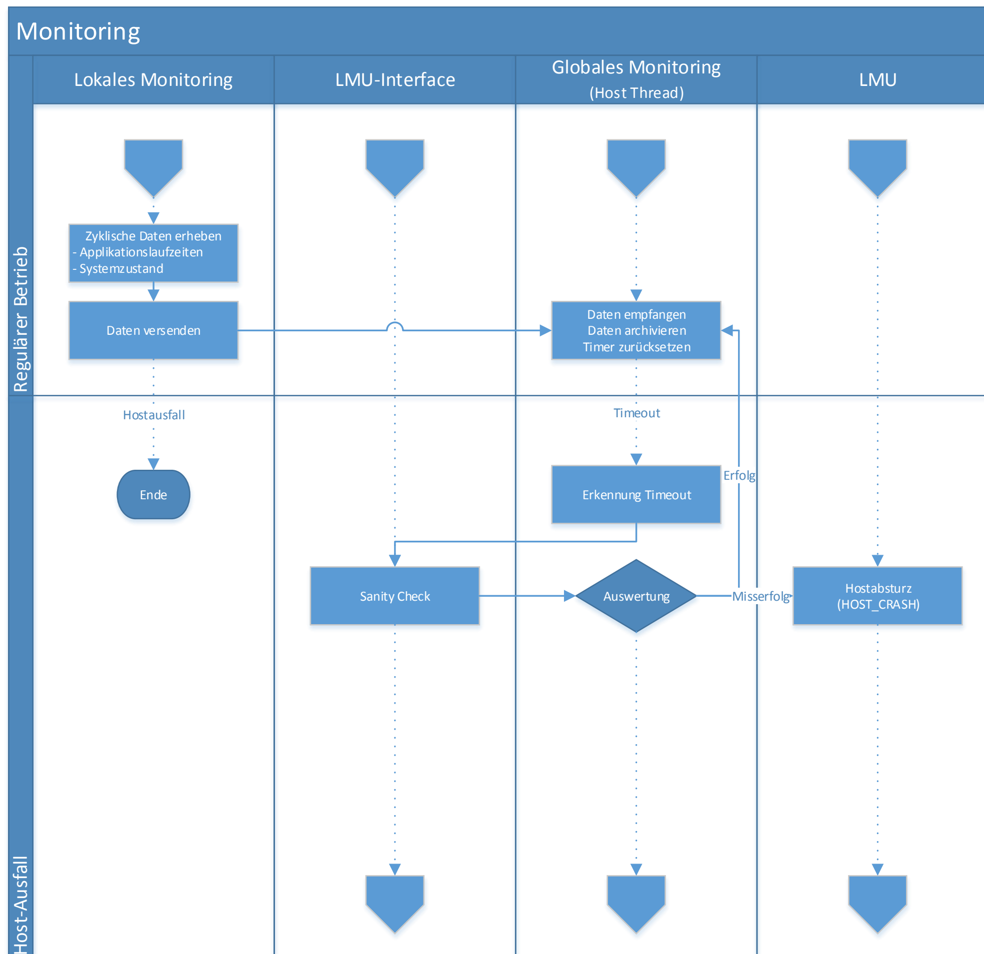


Abbildung 34: Erkennung eines Host-Ausfalls durch das Monitoring-System.

Neben der Bestimmung der Applikationslaufzeiten im regulären Betrieb werden WCET- oder Deadline-Überschreitungen vom Scheduler erkannt und behandelt. Abbildung 35 zeigt den Ablauf bei der Erkennung einer WCET- oder Deadline-Überschreitung.

Der Scheduler erkennt eine WCET- oder Deadline-Überschreitungen einer virtuellen Maschine und löst daraufhin einen virtueller Interrupt (VIRQ) aus. Dieser Interrupt wird über den sogenannten *Event Channel* an die dom0 weitergeleitet, wo er vom *grabvirq*-Prozess abgefangen wird. Dieser Prozess extrahiert die entsprechende Fehlermeldung aus dem *Xend Message Buffer* und leitet diese dann an das lokale Monitoring weiter (siehe 35). Das lokale Monitoring erzeugt daraufhin eine Fehlernachricht, die an das globale Monitoring weitergeleitet wird. Das globale Monitoring

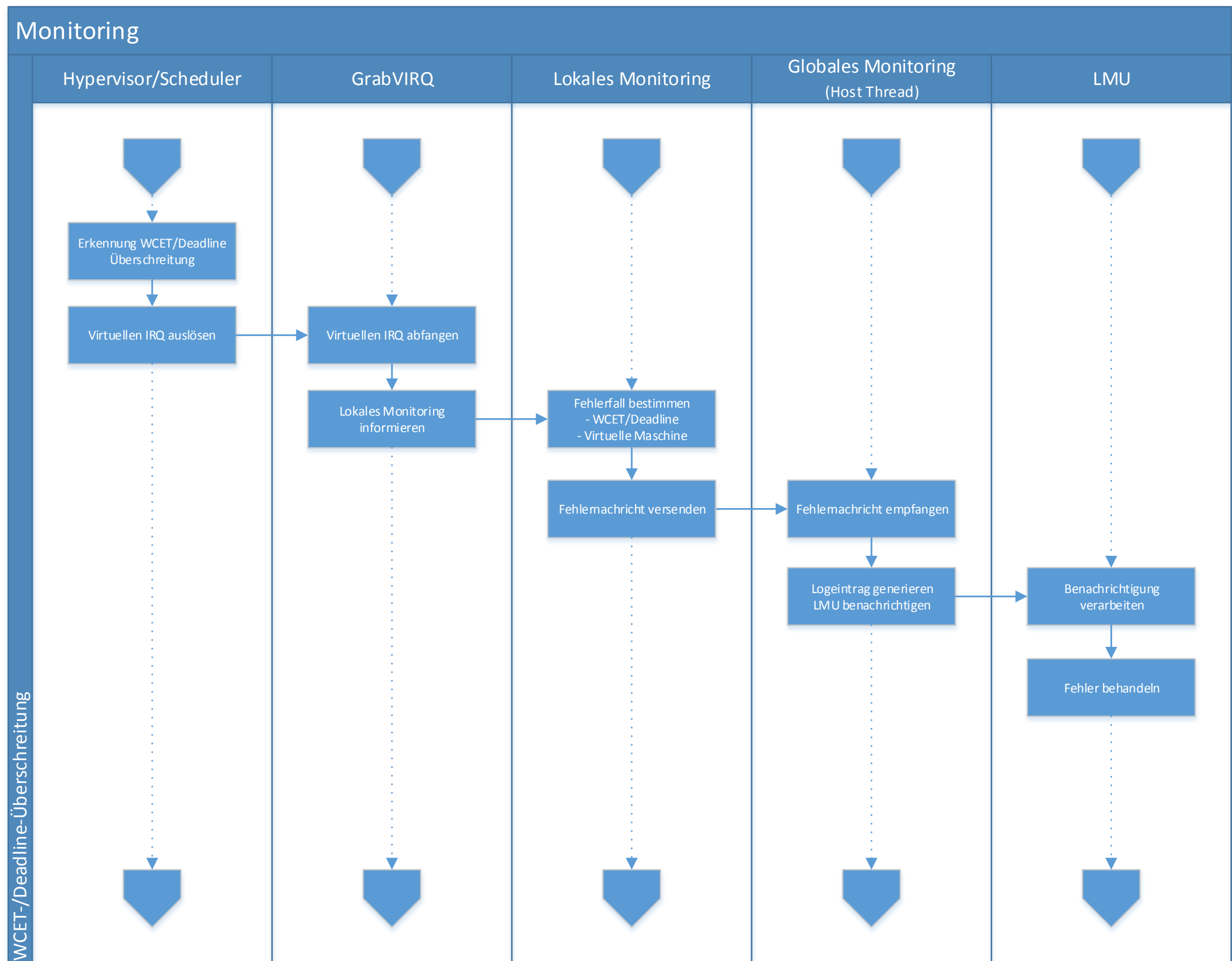


Abbildung 35: Fehlerbehandlung bei WCET- und Deadlineüberschreitungen.

erzeugt den entsprechenden Logeintrag im Fehlerlog der entsprechenden Applikation und informiert die LMU über die Fehlersituation.

6.5. Lokales Scheduling

(Vasco Fachin) Wie in Kapitel 5.6 erläutert wurde, fiel die Wahl des lokalen Scheduling-Verfahrens auf das *Fixed-Priority Scheduling*, auf das im Folgenden näher eingegangen wird.

6.5.1. Fixed-Priority Scheduling

Das *Fixed-Priority Scheduling* (kurz FP) weist jedem Task eine statische Priorität zu. Es wird somit garantiert, dass die CPU immer den Task mit der höchsten Priorität ausführt.

Eine Erweiterung von FP ist das sogenannte *Rate-Monotonic Scheduling* (kurz RM). Die Tasks in einem RM-Scheduler erhalten die zusätzlichen Parameter *Slice* und *Periode*: dabei gibt Slice die Rechnerzeit C des Tasks für jede Periode T an. In diesem Fall wird die Priorität des Tasks durch die Periode zugeordnet: der Task mit der kleinsten Periode bekommt immer die höchste Priorität.

Der *Schedulability Test* wurde von Liu und Layland in [40] bewiesen und definiert die folgende Eigenschaft:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} < n(2^{\frac{1}{n}} - 1), \tag{1}$$

Für $n \to +\infty$ konvergiert der Grenzwert gegen $0,69$; d.h.: eine Menge von Tasks mit insgesamt weniger als 69% CPU-Auslastung lässt sich mit diesem Verfahren immer schedulen [41].

6.5.2. Xen Scheduler Architektur

Der Xen Scheduler entscheidet, welche VCPU als nächste für eine bestimmte CPU auszuführen ist. Einer VM können in Xen mehrere VCPUs zugeordnet sein, allerdings werden von dem im Rahmen der PG eingesetzten FP-Scheduler nur virtuelle Maschinen mit jeweils einer einzigen VCPU unterstützt. Einerseits ist ein solches Verfahren wegen der Analogie von VCPU, VM und Task einfacher zu implementieren, andererseits ist ein Single-Core Modell leichter modellierbar mit formalen Methoden wie Real-Time Calculus (siehe 4.7.2).

Die Xen Scheduler Architektur ist modular organisiert und lässt sich leicht verändern, um einen neuen Scheduler zu implementieren.

Eine Schnittstelle ist in der Datei `/xen/include/xen/sched-if.c` definiert und beschreibt die Struktur eines Schedulers und die Parameter und Funktionen,

die ein Scheduler implementieren soll.

Diese Struktur wird durch die Methoden der Datei `/xen/common/schedule.c` verwendet, um die entsprechenden Funktionen des jeweiligen Schedulers auszuführen.

Die eigentliche Implementierung des Schedulers kann in einer einzelnen Datei erfolgen; zusätzlich muss die bestehende Struktur (siehe Listing 6) um den neuen Scheduler erweitert werden. Der FP-Scheduler wurde als `sched_fp` benannt.

```
70 extern const struct scheduler sched_fp_def;     /* FP Scheduler */
71 extern const struct scheduler sched_sedf_def;
72 extern const struct scheduler sched_credit_def;
73 extern const struct scheduler sched_credit2_def;
74 extern const struct scheduler sched_arinc653_def;
75 static const struct scheduler *schedulers[] = {
76     &sched_fp_def,                              /* FP Scheduler */
77     &sched_sedf_def,
78     &sched_credit_def,
79     &sched_credit2_def,
80     &sched_arinc653_def,
81     NULL
82 };
```

Listing 6: Scheduler Deklarationen in schedule.c

6.5.3. Rate-Monotonic Scheduler - Architektur

Der Rate-Monotonic Scheduler für Xen basiert auf dem FP-Scheduler, welcher am Lehrstuhl XII für Informatik der TU Dortmund implementiert wurde.

Die FP-Scheduler Architektur besteht aus drei unterschiedlichen Strukturen, die Informationen über die VCPU, CPU und Domain enthalten.

Die `struct fp_vcpu` spielt eine zentrale Rolle, da sie die entscheidenden Informationen für das Scheduling speichert (siehe Listing 7).

```
110 struct fp_vcpu {
111   struct vcpu *vcpu;
112   struct list_head queue_elem;
113 
114   /*Parameters for FP*/
115   int        priority;   /* priority*/
116   s_time_t   period;     /*=relative deadline*/
117   s_time_t   slice;      /*=worst case execution time*/
118 
119   bool_t     awake;
```

```
120    bool_t     init;
121    bool_t     is_booted;
122
123    /* Time Accounting */
124    s_time_t   cpu_time;
125    s_time_t   period_time;
126    s_time_t   new_start;
127 };
```

Listing 7: Virtuelle CPU Struktur in sched_fp.c

Am wichtigsten sind die Variablen `priority` (die Priorität der VCPU), `period` (die Periode, also der Abstand, nach dem sich ein Berechnungszyklus wiederholt) und `slice` (die maximale Berechnungszeit der VCPU (WCET) bezogen auf eine einzelne Periode).

Die Variablen `cpu_time`, `period_time` und `new_start` stehen als Überwachungsparameter zur Verfügung: `cpu_time` ist die bisher in der laufenden Periode verwendete Berechnungszeit der VCPU, `period_time` die abgelaufene Zeit des aktuellen Zyklus und `new_start` enthält der neuen Zyklusstart.

Die booleschen Parameter werden zur Zustandsüberprüfung verwendet: ob die VM schläft und somit nicht schedulebar ist, ob die Überwachungs-parameter bereits initialisiert wurden `init` und ob die VM potenziell gescheduled werden kann(`awake`).

Außerdem wird auch die `struct fp_dom` definiert: diese ist in unserem Falls nicht voll ausgeschöpft, da jede Domain nur eine VCPU hat. In einem Multi-Core-CPU-Modell werden die VCPUs direkt durch die Parameter der Domain initialisiert und somit nur einmalig instanziiert (siehe Listing 8 und Listing 9).

```
144 struct fp_dom {
145    struct domain   *domain;
146    s_time_t   period;      /*=(relative deadline)*/
147    s_time_t   slice;       /*=worst case execution time*/
148    int priority;
149 };
```

Listing 8: Domain Struktur in sched_fp.c

```
585 /* Priority is set as a Rate Monotonic Scheduling
586 * The highest priority is taken by the dom0
587 * the VM priority is inversely proportional
588 * to the period w.r.t. dom0
589 */
590 fp_dom->priority = (VM_DOM0_PRIO - op->u.fp.priority);
```

```
591 for_each_vcpu ( d, v ) {
592   FPSCHED_VCPU(v)->slice = op->u.fp.slice;
593   FPSCHED_VCPU(v)->period = op->u.fp.period;
594   FPSCHED_VCPU(v)->priority = (VM_DOM0_PRIO - op->u.fp.priority);
595   FPSCHED_VCPU(v)->cpu_time = 0;
596    fp_reinsertsort_vcpu(v);
597 }
```

Listing 9: Setzen der Priorität für jede VCPU der Domain in sched_fp.c

Schließlich definiert die `fp_cpu` die VCPU-Queues: die *Running Queue* (kurz RunQ) enthält die aktuellen schedulebaren VCPUs. Diese sind nach der entsprechenden Priorität sortiert. In die Deferred Queue (kurz DefQ) werden die VMs hinzugefügt, die ihre Berechnung in dieser Periode bereits abgeschlossen haben; sie sind dabei nach dem neuen Startpunkt des nächsten Zyklus geordnet (siehe Listing 10).

```
102 struct fp_cpu {
103   struct list_head runq;
104   struct list_head deferredq;
105 };
```

Listing 10: Cpu Struktur in sched_fp.c

Das Modell des Schedulers wurde wie folgt konzipiert:

- Die `do_schedule` Funktion nimmt die erste VCPU in der RunQ und liefert sie an den Hypervisor zurück;
- Wenn die VM ihre Berechnung durchgeführt hat, schickt sie einen Hypercall und wird in die DefQ verschoben (und nach dem neuen Startpunkt sortiert eingefügt);
- Sobald für die VM eine neue Periode beginnt, wird sie wieder in die RunQ verschoben;

Zusätzlich integriert RM Sicherheit- und Überwachungsmethoden im Fall einer WCET bzw. Deadline Überschreitung, um eine VM zu verdrängen: wenn die VCPU ihre Berechnungszeit überschreitet, wird sie in die DefQ einsortiert und für diese Periode nicht mehr gescheduled. Bei einer Deadline-Überschreitung werden die Überwachungsparameter zurückgesetzt und anschließend ein neuer Zyklus begonnen (siehe Abbildung 36).

Die Zuordnung der VCPUs zu den Queues ist nach diesem Modell der entscheiden-

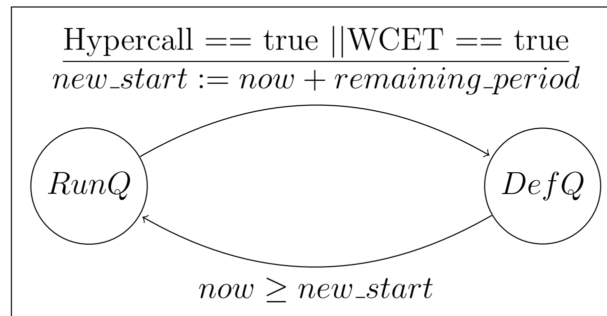


Abbildung 36: Queue-Zustandsautomat

de Punkt, um die FP-Strategie zu implementieren. Diese wird durch die Methode `__runq_insert` durchgeführt: die übergebene VCPU wird mit den schon enthaltenen Objekten der RunQ verglichen und nach absteigender Sortierung hinzugefügt (siehe Listing 11).

```
264 static inline void __runq_insert(unsigned int cpu, struct fp_vcpu *fpv)
265 {
266   struct list_head * const runq = RUNQ(cpu);
267   struct list_head *iter;
268 
269   list_for_each( iter, runq ) {
270       const struct fp_vcpu * const iter_fpv = __runq_elem(iter);
271       /* Where the magic happens*/
272       if ( fpv->priority > iter_fpv->priority )
273           break;
274   }
275   list_add_tail(&fpv->queue_elem, iter);
276 }
```

Listing 11: RunQ in sched_fp.c

Die `runq_insert` Funktion wird in drei Fälle aufgerufen:

1. bei der Erzeugung einer VCPU;
2. beim *wakeup* (d.h. nach *I/O Wait*) einer VCPU;
3. wenn die neue Periode beginnt und die VCPU wieder schedulebar ist;

In gleicher Weise werden die VMs in die DefQ verschoben: wenn ein *Berechnungsende*-Hypercall gesendet wird oder die VCPU ihre WCET überschreitet, wird für die VCPU der zukünftige Startpunkt der neuen Periode berechnet und sie danach entsprechend in die DefQ eingefügt (siehe Listing 12).

```
293 static inline void __defq_insert(unsigned int cpu, struct fp_vcpu *fpv)
294 {
295   struct list_head * const defq = DEFERREDQ(cpu);
296   struct list_head *iter;
297
298   list_for_each( iter, defq ) {
299     const struct fp_vcpu * const iter_fpv = __defq_elem(iter);
300     if ( fpv->new_start < iter_fpv->new_start ) {
301       break;
302     }
303   }
304   list_add_tail(&fpv->queue_elem, iter);
305 }
```

Listing 12: DefQ in sched_fp.c

Wie bereits erwähnt wurde, ist der Kern des Schedulers die `do_schedule` Funktion. Ziel dieses Algorithmus ist es, die jeweils nächste ausführbare VCPU auszuwählen. Da diese Funktion oft aufgerufen wird (in unserem Fall mindestens alle 10 Mikrosekunden), sollte die Implementierung so effizient wie möglich sein.

```
 1 /* FP_Scheduler Algorithm */
 2 INPUT: currentVCPU, RunQ, DefQ, now
 3 OUTPUT: nextVCPU
 4
 5 if head of DefQ is schedulable
 6   then do insert head in RunQ
 7
 8 for each vcpu in RunQ
 9   do update vcpu.time_accounting
10   if vcpu.cpu_time > vcpu.wcet
11     do insert vcpu in DefQ
12     do notify Hypervisor
13   if vcpu.period_time > vcpu.period
14     do reset vcpu.time_accounting
15     do notify Hypervisor
16
17 do remove head of RunQ
18 if head = NULL
19   then nextVCPU = idleVCPU
20 else nextVCPU = head
21
22 return nextVCPU
```

Listing 13: Scheduling Algorithmus - PseudoCode

Der Algorithmus bekommt als Eingabe die aktuelle laufende VCPU, die beiden Queues (RunQ und DefQ) und den jetzigen Zeitpunkt.

Zuerst wird überprüft, ob das erste Element der DefQ schedulebar ist, d.h. für die VCPU hat die neue Periode begonnen. In diesem Fall wird sie in die RunQ einsortiert.

Zeile 8 und 9 des Listings 13 sorgen dafür, die Überwachungsparameter der VCPUs zu aktualisieren. Wenn eine VM die currentVCPU enthält, d.h. sie zuletzt gescheduled wurde, wird die Berechnungszeit der VCPU aktualisiert. Andernfalls wird für alle VCPUs der neue Zeitpunkt berechnet.

Danach, falls eine VM ihrer WCET überschritten hat, wird sie in die DefQ hinzugefügt und der Hypervisor (und anschließend die Dom0) benachrichtigt. Somit hat die VM keinen Einfluss auf die anderen VCPUs, da sie nicht mehr als ihre angegebene Berechnungszeit bekommen wird.

In gleicher Weise wird der Hypervisor benachrichtigt, wenn eine VM ihre Deadline verpasst: in diesem Fall muss nur mit dem neuen Zyklus angefangen werden, da die VCPU ihre WCET nicht überschritten und deshalb die Auslastung des System nicht negativ beeinflusst hat.

An dieser Stelle (Zeile 17 des Listings 13) sind die beiden Queues richtig sortiert: der Algorithmus nimmt dann die erste VCPU der RunQ und liefert sie an den Hypervisor zurück.

Der Schwachpunkt des Algorithmus ist das Update der Queues und der Zeitparameter. Es ist aber nicht möglich, diese an einer anderen Stelle zu aktualisieren. Die `do_schedule` Funktion entscheidet, wie lange die VCPU berechnen darf; ein Zeitquantum wird im Zusammenhang mit der nextVCPU zurückgeliefert. Das bedeutet aber nicht, dass die VM so lange berechnen wird: sie kann auch selbst die CPU abgeben (I/O Wait oder Berechnungsende) oder `do_schedule` wird nach einer Unterbrechung wieder aufgerufen [15]. Es ist in diesem Fall nicht klar, wie lange die currentVCPU tatsächlich berechnet hat. Aus diesem Grund müssen die Überwachungsparameter der VCPUs an dieser Stelle aktualisiert werden [21].

[21] In gleicher Weise hat der Xen SEDF Scheduler auch eine `update_queues()` Funktion innerhalb des Scheduling Algorithmus

6.5.4. Hypervisor Änderungen

Der Scheduler wird durch die Xen-Schnittstelle in der `const struct scheduler sched_fp_def` Struktur definiert (siehe Listing 14).

```
761 const struct scheduler sched_fp_def = {
762   .name       = "Fixed Priority Scheduler",
763   .opt_name = "fp",
764   .sched_id  = XEN_SCHEDULER_FP,
765 
766   .init_domain     = fp_init_domain,
767   .destroy_domain = fp_destroy_domain,
768 
769   .insert_vcpu     = fp_insert_vcpu,
770   .remove_vcpu     = fp_vcpu_remove,
771 
772   .alloc_vdata     = fp_alloc_vdata,
773   .free_vdata      = fp_free_vdata,
774   .alloc_pdata     = fp_alloc_pdata,
775   .free_pdata      = fp_free_pdata,
776   .alloc_domdata   = fp_alloc_domdata,
777   .free_domdata    = fp_free_domdata,
778 
779   .do_schedule     = fp_do_schedule,
780   .pick_cpu        = fp_pick_cpu,
781 
782   .sleep           = fp_sleep,
783   .wake            = fp_vcpu_wake,
784   .adjust          = fp_adjust,
785 
786   .reset_period    = fp_reset_period,
787   .is_booted       = fp_is_booted,
788 };
```

Listing 14: FP-Scheduler Struktur in sched_fp.c

Die Attribute haben die folgende Bedeutung:

Beschreibung `.name` benennt den Scheduler; `.opt_name` ist der Paramenter, der beim Booten gelesen wird und somit den entsprechenden Scheduler ausgewählt; und `.sched_id` definiert den entsprechenden Scheduler-Operation Hypercall;

Domänen Verwaltung `.init_domain` initialisiert eine Domain mit den zugehörigen Parametern und `.destroy_domain` deallokiert den reservierten Speicher;

VCPU Verwaltung Beim Erzeugen einer Domain fügt `.insert_vcpu` die VCPU in die RunQ hinzu und `.remove_vcpu` entfernt sie von der RunQ beim Zerstören der Domain; `.sleep` und `.wake` besorgen dafür, eine VCPU schla-

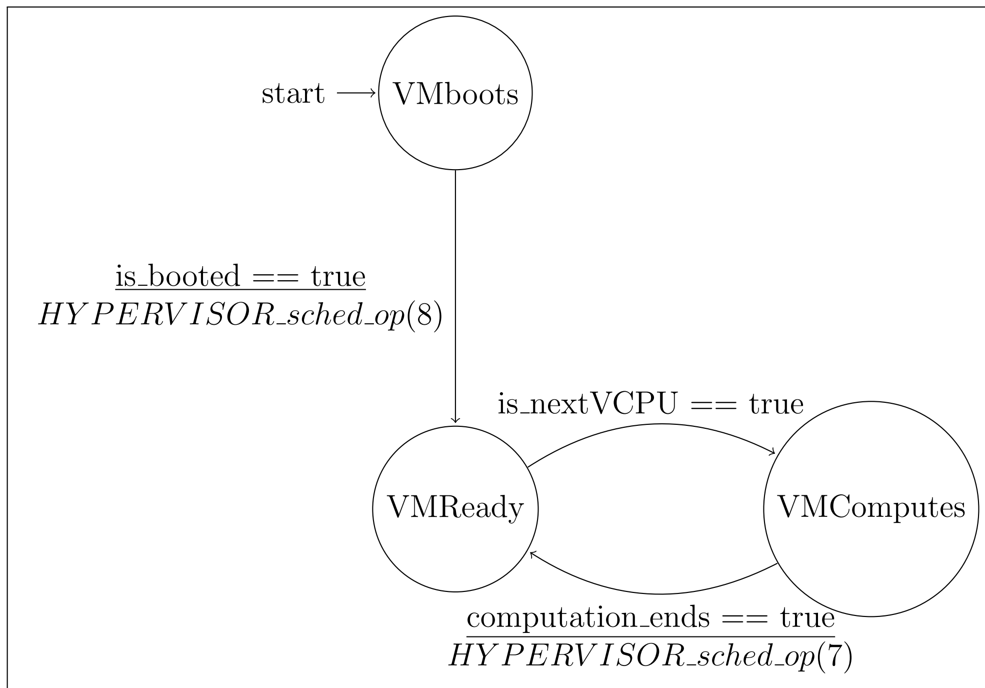


Abbildung 37: Das Hypercall-Modell. Berechnungsende = 7, Boot = 8

fen zu legen bzw. aufzuwecken; `.pick_cpu` ordnet eine VCPU einer CPU zu; durch `.adjust` können die jeweiligen VCPU-Parameter verändert werden;

Speicher Verwaltung Die Funktionen `.alloc_*` und `.free_*` allokieren/deallokieren den Speicher der entsprechenden Strukturen: `vdata` für VCPUs, `domdata` für Domains und `pdata` für CPUs. Zusätzlich initialisiert `.free_vdata` auch die RunQ und DefQ der jeweiligen CPU;

Scheduling Durch die `.do_schedule` Funktion wird das tatsächliche Scheduling-Verfahren durchgeführt;

Die Funktionen `.reset_period` und `.is_booted` sind nicht Teil der Xen-Scheduler Schnittstelle, sie wurden aber hinzugefügt, um die Zustände unseres Modells abzubilden (siehe Abbildung 37).

Um das Berechnungsende der VM zu kommunizieren, wurde ein zusätzlicher Hypercall definiert. Genaugenommen wurden zwei neue Befehle für den `HYPERVISOR_sched_op()` Hypercall implementiert.

Aus diesem Grund wurde es notwendig, den Hypervisor zu modifizieren. In gleicher Weise wie die Funktionen in der `schedule.c` auf Hypercalls reagieren, wurden die entsprechenden Befehle, die die VM sendet, implementiert. Zusätzlich wurde die Xen Scheduler Schnittstelle angepasst, um die von dem Hypercall aufgerufenen Funktionen in dem FP-Scheduler zu beschreiben (siehe Listing 15).

```
761 /* SCHEDOP_pg574 = 7, SCHEDOP_boot = 8*/
762 ret_t do_sched_op(int cmd, XEN_GUEST_HANDLE(void) arg) {
763   ret_t ret = 0;
764
765   switch ( cmd ) {
766
767   case SCHEDOP_pg574: {
768     sched_hypercall();    /* SCHED_OP(VCPU2OP(v), reset_period, v); */
769     break;
770   }
771
772   case SCHEDOP_boot: {
773     sched_boot();         /* SCHED_OP(VCPU2OP(v), is_booted, v); */
774     break;
775   }
776   /* Omissis*/
777 }
```

Listing 15: Neue Fälle für den `sched_op` Funktion in schedule.c

Der Fall `SCHEDOP_pg574` findet statt, wenn die VM ihren Berechnung durchgeführt hat; die `fp_reset()` Funktion wird aufgerufen und somit die VM in die DefQ hinzugefügt.

`SCHEDOP_boot` wird gesendet, wenn die VM bereit ist, um ihre Berechnung durchzuführen. Das ist für den FP-Scheduler notwendig, damit ab diesem Zeitpunkt die Dom0 im Falle einer WCET- oder Deadline-Überschreitung benachrichtigt werden kann und anschließend die entsprechenden Maßnahmen ergriffen werden können. Bisher wurden nur die Überwachungsparameter aktualisiert und ein *Soft-Flag* gesetzt, damit die die Nachricht "Still Booting" im Debug-Modus zu sehen ist.

Wenn dieser Hypercall angekommen ist und die VM eine WCET/Deadline überschritten hat, wird das Flag `wcet_missed` bzw. `deadline_missed` gesetzt und eine virtuelle Unterbrechung am Ende der `schedule()`-Funktion an die Dom0 gesendet (siehe Listing 16 und Kapitel 6.4).

```
1024 if (wcet_missed == 1) {
1025         /* Send VIRQ*/
```

```
1026            send_guest_global_virq(dom0, VIRQ_COMP_TIME);
1027            wcet_missed = 0;
1028        }
```

Listing 16: Virtuelle Unterbrechung im Falls einer Überschreitung in schedule.c

6.5.5. libxc und xend/xm Schnittstelle

Nach der Scheduler-Implementierung und der Hypervisor-Anpassung wurde auch die `libxc`- bzw. die `xend/xm`-Schnittstelle (In C bzw. Python implementiert) erweitert, um die Möglichkeit anzubieten, Informationen über den aktuellen Zustand des Schedulers und die Parameter `slice`, `period`, `priority` der VM abzufragen oder zu modifizieren.

Konkret wurde der Befehl `xm sched-fp` hinzugefügt, der im Dom0-Terminal ausführbar ist.

```
1  root@ubuntu-rechts:~# xm sched-fp -h
2  Usage: xm sched-fp [-d <Domain> [-s[=SLICE]|-P[=PERIOD]|-p[=PRIORITY]]]
3
4  Get/set fp scheduler parameters.
5    -d DOMAIN, --domain=DOMAIN       Domain to modify
6    -s SLICE, --slice=SLICE          Slice (ms)
7    -P PERIOD, --period=PERIOD       Period (ms)
8    -p PRIORITY, --priority=PRIORITY Priority (int)
9
10 root@ubuntu-rechts:~#xm sched-fp
11 Name                             ID  Slice(micros)  Period(micros)  Priority
12 Domain-0                          0       15.0            20.0        10000
13 ds_ubuntu_udp_no_wait_39         14      100.0           250.0          250
14
15 root@ubuntu-rechts:~#xm sched-fp -d14 -s150 -P300 -p300
16 Setting Variables:
17 Slice is
18 150
19 Period is
20 3000
21 Priority is
22 300
23 Name                             ID  Slice(micros)  Period(micros)  Priority
24 Domain-0                          0       15.0            20.0        10000
25 ds_ubuntu_udp_no_wait_39         14      150.0           300.0          300
```

Listing 17: `xm sched-fp` Befehl, Beispielausgabe

Das Aufrufen des Befehls funktioniert wie folgt:

1. in der xm-Library wird die Eingabe geparst;
2. die entsprechenden Parameter (set/get Informationen) werden an die xend Schnittstelle weitergeleitet;
3. *xend* leitet die Parameter der aufgerufenen Methode an die korrespondierenden libxc C-Funktionen weiter;
4. die libxc Library erzeugt die Scheduler-Hypercalls für die *set*- und *get*-Funktionen des Schedulers. Zusätzlich sorgt sie dafür, eine Verbindung zwischen Dom0 und Hypervisor aufzubauen und damit die korrekten Hypercalls zu senden;
5. der Hypervisor leitet den Scheduler-Hypercall an den Scheduler weiter;
6. durch die adjust() Methode des Schedulers (siehe Listing 14) werden schließlich die Parameter entweder gesetzt oder gelesen.

6.6. Gastsystem

(Gregor Kotainy) Aus Kapitel 6.5 geht hervor, dass der Hypervisor angepasst wurde, um auf zwei spezielle *Hypercalls* der Gastsysteme zu reagieren. Damit stellt unser resultierendes System dem Benutzer eine Schnittstelle zur Verfügung, mit der er sein Gastsystem auf Einhaltung der Deadline kontrollieren lassen kann. Zunächst hat das System Zeit zu booten, Daten zu sammeln und teilt dem Scheduler anschließend seine Bereitschaft mit dem Boot-*Hypercall* (siehe Abbildung 37) mit. Anschließend sollte es seinen zyklischen *Hypercall* senden, um abgeschlossene Berechnungen innerhalb der Periode zu signalisieren.

Definition Hypercall In Analogie zum *System call* (auch *Syscall* genannt) für den Kernel unter Linux, ist der *Hypervisor call* (auch *Hypercall* genannt) ein Funktionsaufruf (*Software-Trap*) von einer *VM* zum Hypervisor. Die Gastsysteme können *Hypercalls* für den Zugriff auf privilegierte Operationen verwenden, wie zum Beispiel das Aktualisieren der *Pagetables* [28]. So können privilegierte Operationen von Benutzeranwendungen durch vertrauenswürdigen Code behandelt werden.

Hypercalls sind asynchron, um Prozesse und andere VMs nicht zu blockieren [16].

Dabei ist der Aufruf - wie *Syscalls* - synchron, wobei die Rückrichtung über Ereignis-Kanäle realisiert ist, die ihrerseits asynchrone Queues sind. Im Vergleich zum Linux Kernel mit etwa 300 *Syscalls* werden in *XEN* etwa 50 *Hypercalls* definiert.

Im Unterschied zu bekannten Virtualisierungsplattformen wie *KVM* (direkte *Hypercall* Instruktionen), *VMWARE* (nutzt *VMI*[22]) und *Hyper-V* (beschreibt *MSR*[23]) wird in *XEN* eine *Hypercall page* beschrieben. Während dies in HW-virtualisierten Gastsystemen aufwändig ist, kann auf paravirtualisierten Gastsystemen die Unix abstrahierte Schnittstelle `/proc/xen` genutzt werden.

7. Beispielapplikation

(Dominic Wirkner) Um die Leistungsfähigkeit der entwickelten Plattform zu zeigen, musste die Projektgruppe ein geeignetes Beispiel für eine Benutzerapplikation finden. Die Aufgabe der Beispiel-Applikation ist in erster Linie natürlich die Echtzeitfähigkeit der Plattform zu demonstrieren, indem eine vordefinierte Antwortzeit garantiert werden kann. Des weiteren müssen implementierte Fehlertoleranzkonzepte im Einsatz gefordert werden.
Eine wichtige Rolle spielt jedoch auch der geeignete thematische Kontext, in dem die Plattform professionell eingesetzt werden könnte. Das Umfeld vom großen Stromnetzen, wie es bereits in Kapitel 2.1 als Beispiel diente, erfüllt dieses Kriterium. Im speziellen diente die Idee der Erkennung von Kurzschlüssen als Grundlage der Applikation. Im professionellen Einsatz werden diese auch als Distanzschutz bezeichnet.

7.1. Beschreibung des Distanzschutzes

Die Aufgabe eines Distanzschutzes ist es, in einem Stromnetz einen Kurzschluss festzustellen. Dazu erhält er von einem Sensor Werte für Strom und Spannung, welche üblicherweise im Kontext von Wechselstrom durch zwei sinusförmige Kurven beschrieben werden können. Ein Distanzschutz puffert eine vordefinierte Menge dieser Werte, approximiert auf Basis einer Modellfunktion die zugrunde liegenden Kurven und berechnet im Anschluss daran die zugehörige Impedanz.

[22]mehr zu VMI: `http://en.wikipedia.org/wiki/Virtual_machine_interface`
[23]mehr zu MSR: `http://en.wikipedia.org/wiki/Microsoft_Reserved_Partition`

Im Falle eines Kurzschlusses sinkt die Spannung in Richtung Fehlerstelle, wodurch sich die Impedanz bei gleichem oder größer werdenden Strom verringert. Unterschreitet sie dabei einen vorher definierten Schwellenwert, so deutet dies für den Distanzschutz auf einen Kurzschluss hin und dieser löst aus. Die Auslösung muss dabei innerhalb einer vorgegebenen Zeit nach Auftreten des Kurzschlusses erfolgen, um Schäden am Stromnetz zu auszuschließen.
Das Szenario eines solchen Distanzschutzes eignet sich daher sehr gut, um die Echtzeitfähigkeit der implementierten Plattform zu demonstrieren. Eine Auslösung kann nur dann rechtzeitig erfolgen, wenn dem Distanzschutz ausreichend Zeit zur Verfügung gestellt wird, um Sensorpakete zu verarbeiten und die Impedanz zu berechnen. Die Masterarbeit von Björn Keune [42] diente hierbei als Inspiration und Vorlage für die Umsetzung. Für die elektrotechnischen Details sei daher auf seine Arbeit verwiesen.

7.2. Das Szenario

Die Netzfrequenz im europäischen Raum beträgt 50 Hz und damit dauert eine Schwingung von Strom oder Spannung 20 ms. Dem Distanzschutz von Björn Keune dienten jeweils 80 Messwerte als Grundlage zur Approximation der sinusförmigen Kurven. Da die Modellfunktion genau eine Schwingung erwartet, muss demnach der Distanzschutz alle 250 μs jeweils einen Messwert für Strom und Spannung erhalten.
Auf der anderen Seite sollte ein Kurzschluss demnach nach spätestens 80 Messwerten, sprich 20 ms, erkannt werden. Die VDN-Richtlinie [24] für solche Schutzfunktionen schreibt eine Auslösezeit von 30 ms vor. Theoretisch ist die Erfüllung der Richtlinie also problemlos möglich. Die Herausforderung besteht nun darin, dass die Implementierung eines Distanzschutzes, als Applikation der beschriebenen Plattform, diese Richtlinie ebenfalls einhalten kann.

[24] VDN-Richtlinie für digitale Schutzsysteme *www.vde.com/de/fnn/dokumente/documents/richtlinie-digitale-schutzsysteme_vdn2003-11.pdf*

7.3. Implementierung

Die implementierte Umsetzung des oben beschriebenen Distanzschutzes besteht im wesentlichen aus drei Teilen:

- **Sensor:** Generiert Messwerte anhand der Modellfunktion
- **Aktor:** Registriert die Auslösung der Schutzfunktion
- **Distanzschutz:** Die eigentliche Schutzfunktion; implementiert als virtuelle Maschine

Sensor und Aktor Entscheidend für die Messung der vorgegebenen Zeit ist eine synchrone Uhr zwischen Sensor und Aktor. Der Einfachheit halber wurden diese beiden Teile daher als ein Programm implementiert. Der Start der Zeitmessung ist das Auftreten des Kurzschlusses; das Ende hingegen der Zeitpunkt, wenn das Paket für die Auslösung beim Aktor eingetroffen ist.

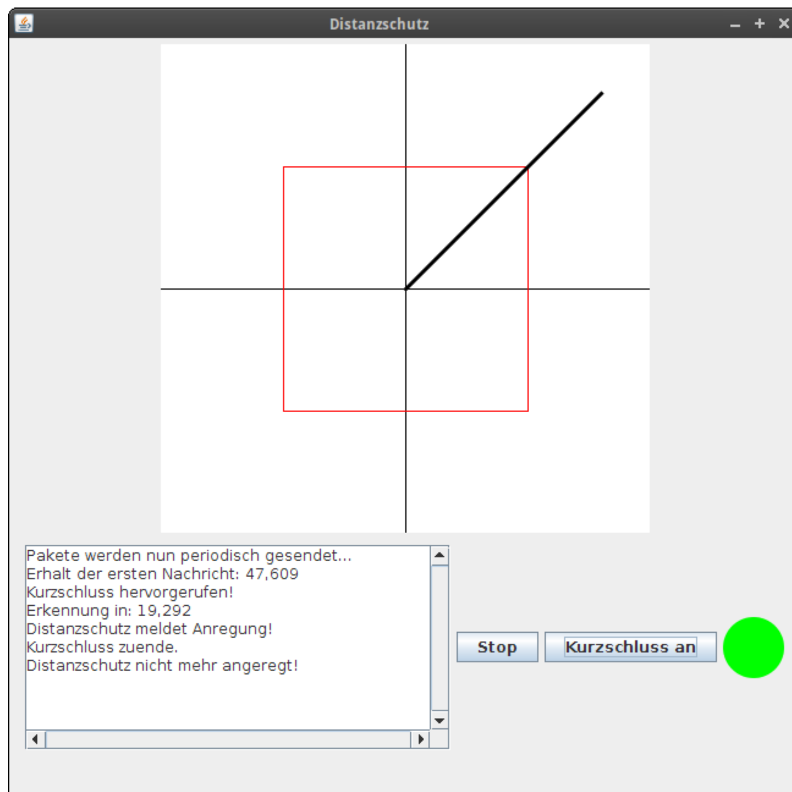


Abbildung 38: grafische Version der Sensor/Aktor-Implementierung

Zu Anschauungszwecken wurde eine grafische Variante in Java implementiert (siehe Abbildung 38), welche neben dem Zustand der Auslösung (ja/nein) auch die berechnete Impedanz darstellt. Als komplexer Wert kann sie in der komplexen Ebene als Zeiger dargestellt werden. Des weiteren wird der Schwellwert, in der Implementierung vereinfacht zu einem Quadrat, als rote Linie dargestellt. Über eine Schaltfläche lässt sich das Senden von Netzwerkpaketen an bzw. aus schalten. Eine andere Schaltfläche dient dazu, zwischen normalen Messwerten und denen eines Kurzschlusses zu wechseln.
Da in der grafischen Variante Kurschlüsse manuell hervorgerufen werden müssen, wurde zusätzlich eine Variante in Form einen Skriptes in Python implementiert, um eine automatisierte Evaluation auch über längere Zeiträume durchzuführen.

Distanzschutz Die Applikation an sich hat folgenden Ablauf (siehe Abbildung 39): Zunächst werden die Messwerte für Strom und Spannung vom Sensor per Netzwerk-Socket empfangen und in den Ringpuffer gespeichert. An dieser Stelle wurde sowohl das TCP- sowie das UDP-Protokoll getestet. Das TCP-Protokoll bietet gegenüber UDP einige Mechanismen, um die Übertragung von Paketen zu garantieren. Dafür muss jedoch theoretisch ein Zeitverlust eingeplant werden, welcher sich aber bei der angestrebten Rate von einem Paket alle 250 μs nicht bemerkbar machte. Die Übertragung der Messwerte mit UDP ist dafür näher an der realen Umsetzung mittels netzwerk-integrierter Sensoren. In diesem Fall muss die Applikation jedoch Paketverluste verkraften können.

Auf Basis der letzten 80 Messwerte, welche im Ringpuffer gespeichert sind, werden nun die Sinus-Kurven für Strom und Spannung approximiert und deren Phasor berechnet. Anschließend kann mit Hilfe der beiden Phasoren die Impedanz bestimmt und geprüft werden. Die Berechnungsergebnisse und der Wert der Auslösung werden dann an den Aktor gesendet.
Um das Konzept in die Plattform integrieren zu können, sind zwei Ergänzungen am Ablauf nötig. Wie beschrieben, ist eine Berechnung nur möglich, wenn bereits 80 Messwerte im Ringpuffer vorliegen. Der umliegenden Plattform muss die Situation, dass der Distanzschutz bereit ist eine Auslösung zu erkennen, separat über einen Hypercall mitgeteilt werden, damit künftige Überschreitungen der Berech-

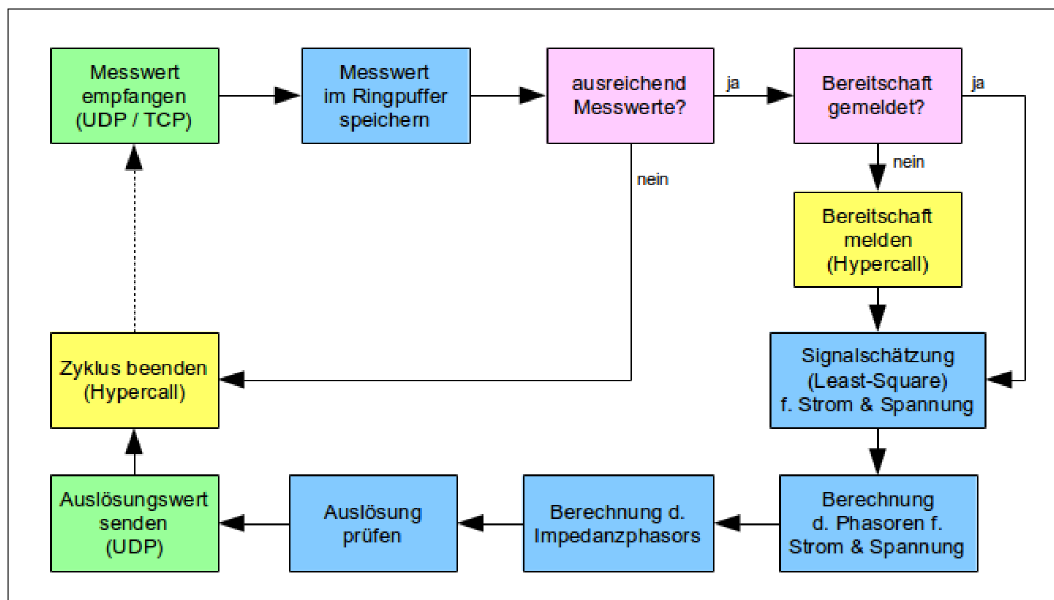


Abbildung 39: interne Prozedur des Distanzschutzes

nungszeit gemeldet werden. Ein weiterer Hypercall muss im Anschluss an das Senden der Ergebnisse an den Aktor erfolgen, um das Ende eines Berechnungslaufs zu signalisieren. Abhängig von diesem Hypercall kann der lokale Scheduler dann eine Zeitüberschreitung erkennen. Die Schutzfunktion selbst wurde in der Programmiersprache C implementiert. Neben einem Geschwindigkeitsvorteil in der Ausführung gegenüber Skriptsprachen oder Java bot diese Entscheidung auch den Vorteil, die selbe Implementierung in verschiedenen Gastsystemen zu testen, denn bei minimalen Systemen wird die Applikationen direkt in den Kernel integriert.

8. Testkonzepte

(Daniel Stoller) In diesem Kapitel sollen die im Rahmen der Projektgruppe eingesetzten Methodiken zum Testen des Systems und seiner Komponenten erläutert werden.

Zunächst werden die verwendeten Testverfahren speziell für das Frontend in Kapitel 8.1 vorgestellt, mit deren Hilfe auch Systemtests durchgeführt werden können. Diese Tests überprüfen nicht nur das korrekte Verhalten einer bestimmten Komponente, sondern das erfolgreiche Zusammenspiel mehrerer Systemkomponenten und werden in Kapitel 8.2 erläutert.

Da die Load Management Unit ein zentraler, komplexer Teil des Systems ist und damit ein hohes Fehlerpotenzial bietet, wird sie in Kapitel 8.3 einem gesonderten Test unterzogen.

8.1. Frontend

(Daniel Stoller) Das Frontend stellt die Schnittstelle unseres Systems zum Benutzer dar und unterliegt daher besonderen Anforderungen, deren Einhaltung getestet werden muss.

Dazu gehört die Robustheit gegenüber fehlerhaften Eingaben seitens des Benutzers; diese sollen erkannt und dem Benutzer auf eine für den Benutzer verständliche Art und Weise mitgeteilt werden. Abgesehen davon darf eine solche Fehleingabe allerdings keine weitere Aktion hervorrufen, damit der Rest des Systems davon unberührt bleibt. Somit müssen alle Benutzereingaben immer auf ihre Korrektheit überprüft werden, um die Stabilität des Systems zu gewährleisten.

Eine weitere Anforderung an das Frontend ist, dass auf korrekte Benutzereingaben zuverlässig innerhalb einer annehmbaren Zeitspanne mit einer Meldung reagiert wird. In unserem Fall wird diese Anforderung erfüllt, indem die Antworten vom System auf Benutzerbefehle in Form einer XML-Datei auf das NAS abgelegt und dort vom Frontend in regelmäßigen Abständen abgeholt und als Popup-Meldung auf der Webseite des Benutzers angezeigt werden.

Eine Möglichkeit, die oben genannten Anforderungen an das Frontend zu testen, ist die manuelle Eingabe von Befehlen auf der web-basierten grafischen Oberfläche. Dadurch können sehr einfach einige Tests, die mitunter auch spezielle Kombinationen von Eingaben beinhalten und damit besondere Fälle abdecken, durchgeführt werden. Allerdings ist diese Art des Testens sehr mühsam und zeitaufwändig, wodurch nur wenige Tests durchgeführt und damit potenziell viele Testfälle nicht abgedeckt werden können. Außerdem neigen Personen, die selbst an der Entwicklung des Systems beteiligt sind, eher zur Erstellung von Tests, die vom System korrekt behandelt werden können und keine Systemfehler aufdecken.

Daher wurde in der Projektgruppe ein weiteres Testkonzept entwickelt, welches auf der automatischen Ausführung von Benutzerbefehlen basiert. Mit Hilfswerkzeugen

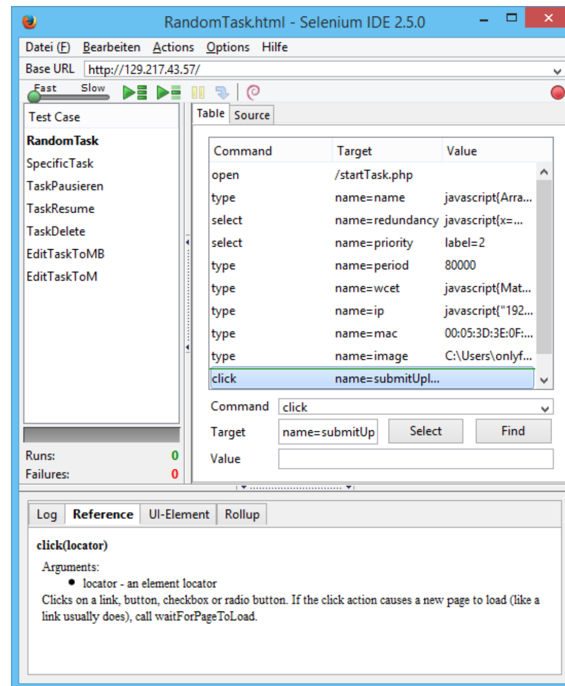


Abbildung 40: Selenium IDE mit in der PG verwendeter, geöffneter Testsuite

können Benutzereingaben generiert und auf der Weboberfläche simuliert werden, um beispielsweise automatisch Eingabeformulare abzuschicken. Konkret haben wir uns für die Werkzeug-Sammlung „Selenium“ entschieden [25], da es eine einfache Möglichkeit zur Browser-Automatisierung bietet. Selenium besteht aus zwei verschiedenen Teilen: „Selenium IDE“ sowie „Selenium WebDriver“. Als integrierte Entwicklungsumgebung für Selenium Skripte bietet das Selenium IDE in Form eines Firefox-Plugins eine grafische Oberfläche zum Aufnehmen, Bearbeiten und Abspielen von Testfällen, welche aus einer Folge von Browser-Aktionen und Abfragen bestehen. Dagegen ist der Selenium WebDriver eine Programmierschnittstelle zur Steuerung von Browsern und wird als Bibliothek für diverse Programmiersprachen angeboten. Für die Anwendung in der PG stellte sich die Selenium IDE als eine bezüglich des Funktionsumfangs ausreichende und einfache Lösung heraus, während der Selenium WebDriver durch seinen größeren Funktionsumfang als zu komplex bewertet wurde.

Die Abbildung 40 zeigt die grafische Oberfläche von Selenium IDE, in der eine in

[25] http://docs.seleniumhq.org/

der PG verwendete „Testsuite“ als Sammlung von Tests zum Abdecken von Anwendungsfällen (siehe Kapitel 8.2) geöffnet ist.

Auf der linken Seite ist die Liste der zur Testsuite gehörenden Testfälle zu sehen. Diese decken häufige Anwendungsfälle wie zum Beispiel das Anhalten oder Löschen von Diensten ab. Angewählt davon ist der Start eines Dienstes mit teilweise zufällig ausgewählten Parametern („Random Task“), sodass auf der rechten Seite die genaue Abfolge der im Browser durchgeführten Aktionen angezeigt wird und bearbeitet werden kann. Durch einen Klick in der oben platzierten Menüleiste können diese Testfälle auomatisch ausgeführt werden und sparen somit viel Zeit beim Testen des Systems im Vergleich zu manueller Eingabe. Dies macht sich vor allen Dingen bei Belastungstests bemerkbar, bei denen viele Dienste auf einmal gestartet und später wieder gestoppt werden müssen. Durch die Nutzung von zufällig generierten Parametern wird außerdem ein breiteres Spektrum möglicher Systemeingaben und eventuell damit verbundener Fehler abgedeckt.

Im nächsten Kapitel 8.2 wird genauer auf die in der PG durchgeführten Systemtests, die teilweise mithilfe der Selenium IDE bewerkstelligt wurden, und deren Resultate eingegangen.

8.2. Systemtests

(Hülya Kaplan, Parinas Nassiri) Um zu überprüfen, ob das System auch richtig funktioniert, wurden manuelle Tests durchgeführt. Dazu gehören die sogenannten Anwendungsfälle, die aus den üblichen Standardaktionen bestehen, und die Fehlerfälle, die ungewollter Weise auftreten können. Im Rahmen der Anwendungsfälle wurden Funktionalitäten wie das Starten und Stoppen eines Dienstes getestet. Zwischendurch wurde der laufende Dienst pausiert und anschließend wieder fortgeführt. In keinem Fall traten unerwartete Situationen auf. Auch das Hinzufügen und das Löschen eines Dienstes verliefen einwandfrei. Dabei wurden sämtliche Tests mit unterschiedlichen Status und Redundanzen durchgeführt.

Fehlerfälle Nachdem im oberen Teil des Kapitels auf die Testfälle eingegangen wurde, werden im folgenden Teil des Kapitels die von uns untersuchten Fehlerfälle beschrieben.

Zur Übersicht der Fehlerfälle, die evaluiert werden sollten, wurde eine Liste mit mehreren Fällen erstellt. Hierfür wurden die Fehlerfälle zwecks Übersichtlichkeit nach den verschiedenen Modulen des Systems gruppiert. Die Liste beinhaltet folgende Gruppierungen: Discovery, LMU, Webserver, Hardware, NAS, Verzeichnisscanner, VM, Monitoring, Grabvirq und Alle. Die Gruppe „Alle“ beinhaltet Fehlerfälle, die sich auf sämtliche Module des Systems beziehen können. Die vollständige Liste befindet sich im Anhang A.

In dieser Liste befinden sich Fehler, die bei den jeweiligen Modulen auftreten können. In der ersten Spalte befinden sich die jeweiligen Komponenten des Systems, nach denen gruppiert worden ist. In der zweiten Spalte sind die möglichen Fehlerfälle der jeweiligen Komponenten aufgeführt. Hier sind mögliche Fehlfunktionen und Störungen der Komponenten angegeben, welche durch Einflüsse aus der Umgebung, wie beispielsweise Stromausfall oder menschliche Fehler auftreten können. In der dritten Spalte sind die erwarteten Ergebnisse geschildert. Hierbei handelt es sich um die gewünschten Reaktionen, die auftreten sollen, wenn bestimmte Fehlerfälle eintreten. Hier wird die von uns gewünschte Vorgehensweise der Komponenten bei Problemen geschildert. Diese Vorgehensweise beinhaltet beispielsweise einen Neustart oder eine Meldung durch das Frontend. Die vierte und letzte Spalte beschreibt das Ergebnis, der von uns manuell ausgeführten Tests. Hier wird die tatsächliche Reaktion der Fehlerfälle und die gewünschte Reaktion verglichen.
Die Fehlerfälle wurden manuell erzeugt und ausgewertet. Dafür wurden beispielsweise die XML-Dateien manuell manipuliert. Jedoch sind einige Fehlerfälle dabei, die nicht nachgestellt werden können, da sie technisch nicht erzwungen werden konnten. Ein Beispiel dafür ist ein Timeout beim Upload einer ISO-Datei. Dies konnte auch nach mehrmaligen Versuchen nicht nachgestellt werden, da auch die größten ISO-Dateien, die zur Testzwecken erstellt wurden, kein Timeout bewirkt haben. Es sind noch einige Fehlerfälle vorhanden, zu denen aus zeitlichen Gründen keine Ergebnisse vorliegen.
Zudem existieren Fehlerfälle, für die keine vernünftige Reaktion erwartet werden kann, wie zum Beispiel das Fehlen von Skripten, denn sobald ein Skript nicht vorhanden ist, kann das davon abhängige System nicht mehr funktionieren und somit kann auch keine Fehlermeldung oder ähnliches realisiert werden. Ein weiteres Beispiel ist der Ausfall des Verzeichnisscanners, wofür keine Maßnahmen implementiert sind und ein manueller Neustart erforderlich ist. Das Testen würde an dieser Stelle

kein Ergebnis liefern. Ein weiterer Fehlerfall ist, dass keine Log-Dateien vom Monitoring vorhanden sind und das Frontend somit keine Statistiken des Systems anzeigen kann.
Einige (wenige) Fehlerfälle zeigten nicht das gewünschte Ergebnis. Dies betrifft zum Beispiel den Fall, dass die LMU bei einem Fehlschlag der Erzeugung der eigenen Redundanz (d.h. bei der Reproduktion von sich selbst) und bei der Verfügbarkeit von nur einem Host, diese Fehler nicht protokolliert.

8.3. Load Management Unit

(Heng Liu) Die *Load Management Unit (LMU)* spielt eine zentrale Rolle in dem entwickelten System, damit die Echtzeitfähigkeit von Diensten gewährleistet werden kann. Jegliche Interaktion zwischen dem Frontend den Hosts erfolgt indirekt über die LMU. Darüber hinaus muss die LMU auf Dienst- oder Hostausfälle reagieren. Aus diesem Grund wurden die Tests der LMU eine besondere Bedeutung beigemessen.

In diesem Kapitel werden neun Testfälle beschrieben, die alle grundlegenden Funktionen der LMU und einige Fehlerfälle abdecken sollen. Alle Tests können mittels eines Skripts automatisch durchgeführt werden. Bei jedem Testfall soll das Skript mit unterschiedlicher Konfiguration von Hosts oder VMs bis zum zwei Mal durchgeführt werden. Nach den Tests ist es möglich, dass die Verteilung von der LMU ausgegeben werden kann. Im folgenden Kapitel wird jeder Testfall beschrieben und kurz erklärt, wie der jeweilige Test durchzuführen ist. Im Anhang F wird der Ablauf und das Resultat der Tests detailliert definiert und beschreiben.

Test 1 Dienst starten Dieser einfache Test ist sehr wichtig, weil es sich bei „Dienst starten“ um eine Basisfunktion des Systems handelt. Wenn ein Dienst mit einer Redundanz gestartet werden soll, muss die LMU richtig reagieren und den Start bearbeiten.

Bei dem Test können vorher definierte Dienste erfolgreich gestartet werden, wenn ausreichend Ressourcen zur Verfügung stehen. Nach dem Test wird eine Rückmeldung für die entsprechenden Dienste von der LMU zurückgeschickt, die in einer CSV-Datei definiert werden können. Bei erfolgreichem Starten eines Dienstes erhält man den

Fehlercode „0: Alles OK“ als Rückmeldung, die in einer CSV-Datei zu schreiben ist. Außerdem können weitere unterschiedliche Fehlermeldungen zurückgegeben werden.

Test 2 Dienst starten, dann pausieren In diesem Testfall wird ein Dienst zuerst gestartet und soll anschließend pausiert werden. Ein laufender Dienst wurde erfolgreich pausiert, wenn die LMU in der relevanten Action.xml richtig gearbeitet hatte. Um zu beurteilen, ob dieser Test erfolgreich war, kann die Datei heap_dump.csv analysiert werden. Nach dem Pausieren des Dienstes sollte die heap_dump.csv leer sein; dies bedeutet, dass der Dienst nicht mehr ausgeführt wird. Diese Funktion konnte erfolgreich getestet werden.

Test 3 Dienst starten, dann stoppen/löschen Dieser Test ist sehr ähnlich zum zweiten Test. Ein Dienst wird am Anfang gestartet und schließlich wird er gestoppt/-gelöscht. Dies wurde erfolgreich im fertigen System getestet. Die leere heap_dump.csv zeigt das korrektes Resultat für diesem Testfall.

Test 4 Gleicher Dienst wird zwei mal gestartet Bei diesem Fehlerfall muss die LMU entsprechende Fehlercodes generieren, da die Ausführung eines zweiten Dienstes mit gleichem Namen nicht möglich ist. Falls zwei Dienste mit gleichem Namen gestartet werden sollen, so wird eine Task.xml und Action.xml mit gleichem Inhalt für jeden Task generiert und der LMU übergegeben. Nachdem der erste Dienst erfolgreich gestartet wurde, muss die LMU das Starten des zweiten Dienstes ablehnen und Fehler zurücksenden. Die Fehlercodes wurden in einer CSV-Datei gespeichert.

Test 5 WCET größer als Periode Bei diesem Fall wird es ebenfalls Fehler auftreten, weil die Länge der WCET nicht die Periode überschreiten darf. Nach dem Test wurde die korrekte Fehlermeldung von der LMU generiert.

Test 6 Belastungstest Belastungstest bedeutet, dass beliebig viele Dienste in einem kurzen Zeitraum hintereinander gestartet werden können. Ob die LMU die Dienste erfolgreich starten kann, hängt von den verfügbaren Hosts, VMs und Cores ab. Der Belastungstest ist eines der wichtigsten Testmodule für die LMU.

Der Ablauf dieses Tests lässt sich in zwei Teile unterteilen. In Test 1/2 betrachtet man, dass vier Hosts für die Ausführung der Dienste zur Verfügung standen. Wie die Ausgabe der LMU anzeigt, konnten alle Dienste mit verschiedenen Konfigurationen auf den unterschiedlichen CPU-Cores der Hosts ausgeführt werden. Es könnten potenziell noch einige weitere Dienste ausgeführt werden, weil es bei einigen Cores ausreichende Auslastung zur Verfügung stand. Darum war dieser Test erfolgreich.

Des Weiteren wurde der Test 2/2 durchgeführt. Bei diesem Fall standen nur drei Hosts für den Test zur Verfügung, wobei die Konfiguration der VMs der Dienste identisch war. Aus den Ausgaben wurde es erkannt, dass einige Dienste nicht gestartet wurden. Der Grund liegte an den nicht ausreichenden Ressourcen für die Dienste, z.B. belastung13, belastung18 und belastung19. belastung13 braucht die CPU-Auslastung in Höhe von 37.5% und sollte mit Redundanz von Master+Backup+Node gestartet werden. Nach dem Starten von belastung12 wurden die Ressourcen durch den Rescheduling-Algorithmus unter der Verwendung von Live-Migration wieder verteilt. Trotzdem konnte die freie Ressource in Höhe von 37.5% auf drei unterschiedlichen Hosts bzw. Cores nicht gefunden werden. Aus diesem Grund wurde belastung13 nicht gestartet. Bei belastung18 und belastung19 war die Situation ähnlich. Somit wurde die Belastungsfähigkeit von LMU erfolgreich getestet und unsere Erwartungen erfüllt.

Test 7 Konfigurationstest Dieser Test wurde entworfen, um zu zeigen, dass ein Dienst mit Master-Backup- oder Master-Backup-Node-Redundanz nicht gestartet werden kann, falls momentan nur ein Host zur Verfügung steht. Am Ende sollte die LMU den Dienst zurückweisen und entsprechende Fehlercodes abschicken.

Test 8 Fehlerhafte Task.xml Bei diesem Test wird eine fehlerhafte Task.xml generiert. Nach der Durchführung des Tests wurde immer der gleiche Fehler generiert („25: Es wurde keine task.xml gefunden. Die action.xml wird nach old verschoben.“), da keine richtige Task.xml generiert wurde.

Test 9 Fehlerhafte Action.xml Bei diesem Test wird eine fehlerhafte Action.xml generiert. Eine fehlerhafte Action.xml bedeutet dabei, dass der enthaltene Befehl nicht zur LMU übertragen werden kann, weil dieser möglicherweise in der Action.xml

falsch übergetragen wurde. Es wird der Fehler („27: Die action.xml ist Fehlerhaft.“) von der LMU zurückgegeben.

9. Evaluation

9.1. Lokales Scheduling

(Vasco Fachin)

Der RM Scheduler ist ein wichtiger Teil der virtualisierten Ausführungsplattform. Aus diesem Grund wurde das korrekte Scheduling der VMs überprüft; die Wahl der VM mit höchster Priorität und die Einhaltung der angegeben Slice/Periode sind von zentraler Bedeutung für das implementierte System.

9.1.1. Task Example

Als VM wurden Mini-OS und CiAO Betriebssysteme verwendet. Einerseits weil sie einfach konfigurierbar sind, andererseits weil sie klein und booten in drei/vier Sekunden, d.h. mehrere VMs können mit wenig Aufwand implementiert und erzeugt werden.

Zusätzlich sind diese virtuellen Maschinen unabhängig, d.h. sie brauchen keine externen Daten, um ihre Berechnung durchzuführen.

9.1.2. Ablauf der VMs

Um den korrekten Ablauf der Tasks anzuzeigen, wurde der Scheduler mit zusätzlichen Debugging-Funktionen erweitert. Diese sind normalerweise durch die `printk` - Methode des Linux Kernels implementiert und damit die Informationen in den *xm dmesg* Puffer geschrieben werden können.

Der Debugging-Modus enthält drei verschiedene Stufen und zeigt, an welchen Stellen des Schedulings die VM steht:

1. die Hauptmethoden `init`, `destroy` und `sleep` und die Methoden `reset_`

`period`, `is_booted` für das Hypercall Handling. Zusätzlich werden die Timings der VM ausgedruckt;

2. RunQ und DefQ `insert/remove` Funktionen;
3. `do_schedule` der VM (außer Dom0) und `wake`: da diese Methode mindestens jede 10 Mikrosekunden aufgerufen werden kann, sind sie zu einer anderen Stufe zugeordnet;

Booting - Listing 18 Die Domain wird initialisiert und die entsprechenden VCPUs erzeugt (Zeile 1 und 2). Ab Zeile 3, als die VCPU in die RunQ sortiert wird, ist sie schedulebar. Während dieser Zeit wird die VM überwacht: die VCPU bekommt immer maximal die entsprechende Slice innerhalb einer Periode: eine verpasste WCET oder Deadline wird erkannt und der aktuelle Zyklus unterbrochen: das System verschiebt die VM in die DefQ (Zeile 6), erzeugt eine Meldung (Zeile 7) und schedult die VCPU in der neuen Periode wieder (Zeile 9). Durch dieses Verfahren wird garantiert, dass auch wenn die VM mehr CPU-Leistung wegen des Booting bräuchte, wird sie blockiert und somit die anderen VM nicht beeinflusst.

Das geschieht bis der Scheduler den Booting-Hypercall bekommt (Zeile 12): ab diesem Zeitpunkt wird die VM richtig überwacht.

Listing 18: Booting/Migration Ablauf

```
1  in fp_init_domain
2  fp_sleep for dom 5
3  Domain: 5 inserted in RunQ
4  Actual Time: 128895316247. New start: 128887545288
5  New Period for domain: 5, Used CPU and Period: 1002054|8970959
6  Domain: 5 inserted in DefQ
7  Period Reset for missing wcet but still in booting
8  Domain 5 removed from DEFQ: Now 128917344497 New Start 128910327176
9  Domain: 5 inserted in RunQ
10 /* Omissis*/
11 Domain: 5 inserted in RunQ
12 Domain 5 has booted
13 Actual Time: 318772500595. New start: 318784497684
14 New Period for domain: 5, Used CPU and Period: 2911|2911
15 Domain: 5 inserted in DefQ
16 Hypercall for Domain 5
17 /* and so on */
```

Normaler Ablauf - Listing 19 Das Listing zeigt, wie die entsprechenden Funktionen zwischen zwei Hypercalls aufgerufen werden: der neue Startpunkt der VM wird berechnet, die laufenden Zeiten für Statistik werden gespeichert und die VM wird in die DefQ verschoben (Zeilen 2, 3 und 4).

Dann, als die VM ihre neue Periode anfängt (Zeile 4), wird sie wieder in die RunQ einsortiert und, da sie die einzige VCPU der CPU ist, sofort gescheduled (Zeilen 5 und 6). Das bis zum nächsten Hypercall (Zeile 9).

Listing 19: Normaler Ablauf

```
1  Hypercall for domain 9
2  Actual Time: 229602312176. New start: 229603401788
3  New Period for domain: 9, Used CPU and Period: 54178|110388
4  Domain: 9 inserted in RunQ
5  Domain 9 removed from DEFQ: Now 229603426095 New Start 229603401788
6  Domain: 9 inserted in RunQ
7  Domain: 9,Current CPU and Period used: 0|56437
8  Domain: 9,Current CPU and Period used: 5367|61804
9  Domain: 9,Current CPU and Period used: 42605|99042
10 Hypercall for domain 9
```

Shut Down - Listing 20 Beim *Shutting-Down* wird zuerst die VM schlafen gelegt und dann die Domain-Objekte werden zerstört, d.h. der Speicherplatz wird deallokiert (Zeilen 5 und 6).

Listing 20: Poweroff/Migration Ablauf

```
1 Hypercall for domain 7
2 Actual Time: 409791211814. New start: 409792241277
3 New Period for domain: 7, Used CPU and Period: 88127|170537
4 Domain: 7 inserted in DefQ
5 fp_sleep for dom 7
6 in fp_destroy_domain
```

Fehler: WCET und Deadline - Listings 21, 22 und 23 Wichtig ist, dass die VM bei der Überschreitung ihre WCET für diesen Zyklus nicht mehr gescheduled wird, damit sie die anderen Domains nicht beeinflusst. Im Prinzip, das gleiche Verfahren wie bei der Ankunft eines Hypercalls gefolgt wird: die `do_schedule` erkennt die Überschreitung (Zeile 10), berechnet der Startpunkt des nächsten Zyklus (Zeilen 11 und 12), verschiebt die VCPU in die DefQ (Zeile 13) und beim nächsten Zyklus wird die VCPU wieder gescheduled (Zeile 14).

Das kann z.B. passieren, wenn die VM einen Software-Fehler enthält und damit die Berechnung verzögert wird.

Listing 21: WCET Überschreitung

```
1  Domain: 2 inserted in DefQ
2  Domain 2 removed from DEFQ: Now 48981589015 New Start 48981534987
3  Domain: 2 inserted in RunQ
4  Domain: 2,Current CPU and Period used: 0|77086
5  Domain: 2,Current CPU and Period used: 6297|83383
6  /* Omissis */
7  Domain: 2,Current CPU and Period used: 620754|697840
8  Domain: 2,Current CPU and Period used: 672638|749724
9  WCET,2,700000,5000000,724422,801508
10 Actual Time: 48982768647. New start: 48986966153
11 New Period for domain: 2, Used CPU and Period: 725408|802494
12 Domain: 2 inserted in DefQ
13 Domain 2 removed from DEFQ: Now 48987008901 New Start 48986966153
```

Bei der Deadline Überschreitung wird allerdings die VM nicht blockiert (Zeile 9 und 10). Das weil die VCPU nicht mehr als ihre WCET berechnet hat und deswegen die anderen Domains nicht beeinflussen kann. Der Fehler wird nur an den Hypervisor und dann Dom0 signalisiert. Das kann passieren, wenn z.B. das Sensornetz ein Problem hat und die Pakete entweder zu spät oder nicht an die VM ankommen.

Listing 22: Deadline Überschreitung

```
1  Actual Time: 184191761439. New start: 184191722941
2  New Period for domain: 2, Used CPU and Period: 139233|138498
3  Domain: 2 inserted in DefQ
4  Domain 2 removed from DEFQ: Now 184191821016 New Start 184191722941
5  Domain: 2 inserted in RunQ
6  Domain: 2,Current CPU and Period used: 0|112776
7  Deadline,2,100000,100000,3781,116557
8  Domain: 2,Current CPU and Period used: 0|16557
9  Domain: 2,Current CPU and Period used: 18507|35064
10 Domain: 2,Current CPU and Period used: 36986|53543
11 Domain: 2,Current CPU and Period used: 55363|71920
12 Domain: 2,Current CPU and Period used: 73755|90312
13 Hypercall for domain 2
14 Actual Time: 184192093793. New start: 184192076155
15 New Period for domain: 2, Used CPU and Period: 101081|117638
16 Domain: 2 inserted in DefQ
```

Anschließend zeigt die Listing 23, wie ein Fehler beim Setzen der *Slice* und *Periode* (beide zu niedrig gesetzt) bei dem RM-Scheduler gehandelt wird.

Listing 23: WCET und Deadline Überschreitung

```
Domain 2 removed from DEFQ: Now 137432751281 New Start 137432668375
Domain: 2 inserted in RunQ
Domain: 2,Current CPU and Period used: 0|82735
Domain: 2,Current CPU and Period used: 3701|86436
Domain: 2,Current CPU and Period used: 7114|89849
Domain: 2,Current CPU and Period used: 10467|93202
Domain: 2,Current CPU and Period used: 13674|96409
Domain: 2,Current CPU and Period used: 16720|99455
Deadline,2,100000,100000,19772,102507
Domain: 2,Current CPU and Period used: 0|2507
Domain: 2,Current CPU and Period used: 2858|5365
Domain: 2,Current CPU and Period used: 5605|8112
Domain: 2,Current CPU and Period used: 8302|10809
Domain: 2,Current CPU and Period used: 10860|13367
Domain: 2,Current CPU and Period used: 71276|73783
WCET,2,100000,100000,110084,112591
Actual Time: 137433398999. New start: 137433385484
New Period for domain: 2, Used CPU and Period: 111008|113515
Domain: 2 inserted in DefQ
Domain 2 removed from DEFQ: Now 137433458646 New Start 137433385484
```

Priorität Ein entschiedener Punkt eines FP-Scheduler ist die korrekte Wahl der nächsten auszuführenden VCPU im Bezug auf ihre Priorität. Insbesondere für den RM-Scheduler, ist es wichtig, dass wenn die VCPU wieder schedulebar ist (d.h. die VCPU wird von der DefQ in die RunQ verschoben), wählt RM die mit der höchsten Priorität zuerst.

Das ist in Listing 24 zu sehen. Hier sind zwei Domains am Laufen: Domain 5 mit Priorität 8600 und Domain 6 mit Priorität 8500, also Dom5 hat eine größere Priorität im Bezug auf Dom6. Bis Zeile 11 läuft nur Dom6, ab Zeilen 11 und 12 wird Dom5 wieder schedulebar, da ihre neue Periode anfängt und wird sofort gescheduled. Das bis zum Ende ihrer Berechnung (Zeilen 21-24) und dann bekommt Dom6 die CPU wieder.

Listing 24: Priority mit 2 VMs

```
root@ubuntu-mitte-rechts:~# xm sched-fp
Name                              ID   Slice(us)  Period(us)  Priority
Domain-0                           0        15.0        20.0     10000
ciao-hypercall1                    5       900.0      1400.0      8600
ciao-hypercall2                    6      1000.0      1500.0      8500


Domain: 6,Current CPU and Period used: 329002|1250675
Domain: 6,Current CPU and Period used: 385933|1307606
```

```
Domain: 6,Current CPU and Period used: 442786|1364459
Domain 5 removed from DEFQ: Now 875768866156 New Start 875768819682
Domain: 5 inserted in RunQ
Domain: 5,Current CPU and Period used: 0|103293
Domain: 5,Current CPU and Period used: 665|103958
Domain: 5,Current CPU and Period used: 1384|104677
Domain: 5,Current CPU and Period used: 2142|105435
Domain: 5,Current CPU and Period used: 2859|106152
Domain: 5,Current CPU and Period used: 3629|106922
Domain: 5,Current CPU and Period used: 4308|107601
Domain: 5,Current CPU and Period used: 5031|108324
Hypercall for domain 5
Actual Time: 875769273632. New start: 875770525006
New Period for domain: 5, Used CPU and Period: 45333|148626
Domain: 5 inserted in DefQ
Domain: 6,Current CPU and Period used: 499605|1426840
Domain: 6,Current CPU and Period used: 499994|1427229
Domain: 6,Current CPU and Period used: 500288|1427523
```

Im Gegenteil wird in Listing 25 eine Domain mit niedrigerer Priorität wieder schedulebar (Zeilen 10 un 11): hier der Scheduler wählt immer noch die VCPU mit höchster Priorität (Zeile 12).

Listing 25: VM 7 mit niedriger Priorität wird wieder schedulbar

```
root@ubuntu-mitte-rechts:~# xm sched-fp
Name                          ID   Slice(us)  Period(us)  Priority
Domain-0                       0      15.0       20.0     10000
ciao-hypercall1                5     900.0     1400.0      8600
ciao-hypercall3                7     900.0     1800.0      8200


Domain: 5,Current CPU and Period used: 43218|90487
Domain: 5,Current CPU and Period used: 43725|90994
Domain 7 removed from DEFQ: Now 1044545251723 New Start 1044545247641
Domain: 7 inserted in RunQ
Domain: 5,Current CPU and Period used: 44206|91475
Domain: 5,Current CPU and Period used: 47531|94800
Domain: 5,Current CPU and Period used: 107971|155240
Domain: 5,Current CPU and Period used: 139412|186681
```

9.1.3. Das Zeitquantum

Neben der Auswahl des nächsten Tasks muss ein Xen Scheduler auch ein Zeitquantum zurückliefern. Das Zeitquantum bestimmt, wie lange die nächste VCPU maximal berechnen darf, vorausgesetzt, dass sie nicht durch eine Unterbrechung

unterbrochen wird.

In SEDF und Credit wird das Quantum bei jedem Zyklus dynamisch gesetzt. In RM ist es aber statisch definiert, um das Overhead der `do_schedule` Funktion zu reduzieren.

Insgesamt wurden 3 verschiedenen Quanta evaluiert: $1\mu s$, $10\mu s$ und $100\mu s$. Das $1\mu s$ wurde sofort ausgeschlossen: in diesem Fall war das Overhead zu groß und die VM konnte noch nicht mal Booten.

Als nächsten wurde das $100\mu s$ Quantum getestet. Obwohl die VM mit dieser Auflösung weniger Berechnungszeit brauchte, ist sie für eine Echtzeitkritischeapplikation nicht geeignet. Bei einer Periode von z.B. $500\mu s$, kann die `do_schedule` des RM Schedulers zu spät erkennen (maximal nach $100\mu s$), dass die VM eine neue Periode angefangen hat.

Aus diesem Grund wurde das Zeitquantum auf $10\mu s$ gesetzt: dies hat ein größeres Overhead in Bezug auf $100\mu s$, aber es bietet eine bessere Auflösung des Schedulingsverfahrens.

9.1.4. Zeit in der do_schedule

Weiterhin wurde die RM-Scheduler `do_schedule` Funktion mit den anderen Xen-Schedulern verglichen. Die Zeiten wurden vor und nach dem Aufruf der Funktion vom Hypervisor gemessen. Wie bereits erwähnt, wird diese Funktion mindestens jede 10 μs aufgerufen, d.h. wenn ihres Overhead deutlich größer als SEDF oder Credit ist, könnte der RM-Scheduler die Performance des System stark beeinflussen und damit die Echtzeitsanforderungen der Applikationen nicht garantieren.

Die Tests wurden jeweils mit drei laufenden VMs und für insgesamt zwei Minuten durchgeführt, d.h. im Durchschnitt wurde die `do_schedule` Funktion ca. 2,4 Millionen Mal aufgerufen. Als Test-VM wurde ein voll-virtualisiertes CiAO genommen mit eingeschalteten Hypercalls, sodass das Modell unseres Systems (der Hypercall signalisiert dem RM-Scheduler das Berechnungsende) unterstützt wird.

Die Ergebnisse (siehe Tabelle 2) zeigen, dass trotz die notwendige Aktualisierung der VCPU-Zeitparameter, die beim jeden Zyklus der Funktion erfolgt, ist die Geschwindigkeit des Rate-Monotinic Schedulers mit den Xen-Schedulern vergleichbar; RM ist

	RM	SEDF	Credit
Zeit in do_schedule	171.24 ns	118.45 ns	205.33 ns

Tabelle 2: Do Schedule Overhead im Vergleich

sogar schneller als Credit: das liegt wahrscheinlich an dem komplexeren Weight/Cap Modell.

Der Vergleich zeigt auch, dass SEDF schneller die nächste VCPU berechnet: allerdings bietet SEDF nicht die Möglichkeit, eine VM zu verdrängen, wenn sie mit ihrer Berechnung fertig wird. Das bedeutet, dass die VM immer *mindestens* ihre Slice als Berechnungszeit bekommt; in dem RM-Scheduler die VM bekommt aber *maximal* ihre Slice als Berechnungszeit. Das System hat eine niedrige Auslastung und kann somit die verbleibenden Resources für z.B. nicht echtzeitkritische Anwendungen verwenden.

9.1.5. Evaluation

Zum Abschluss wurde das Schedulersverhalten und insbesondere die Isolation der VMs getestet. Isolation bedeutet, dass falls ein Fehler in einer VM auftaucht (WCET bzw. Deadline Überschreitung), werden die anderen laufenden VMs nicht betroffen.

Als Domain wurde Mini-OS mit einer einfache `for`-Schleife verwendet. Am Ende der Berechnung wird einen Hypercall gesendet.

Die Schleife hat eine WCET von 300 μs. Die Perioden wurde unterschiedlich gesetzt, um verschiedene Prioritäten an die VMs zuzuordnen.

Die Tabelle 3 zeigt eine richtige Einstellung des Systems, d.h. die gesamste CPU-Auslastung liegt unter 69% (Siehe 6.5). Drei Tests wurden durchgeführt, jeweils mit einer, zwei und drei VMs. Der Scheduler hat immer die VMs richtig gescheduled und keine WCET oder Deadline wurde überschreitet.

Tabelle 4 zeigt das Verhalten des Systems im Fall einer Domain-Überlastung, d.h. eine Domain braucht mehr Rechnerzeit als ihr gegeben wurde. Die Tests wurden in diesem Fall mit insgesamt 10 VMs und für 10 Minuten durchgeführt.

Beim Test 4.1 wurden die Parameter richtig gesetzt. Die gesamte CPU-Auslastung

# Task	Core Auslastung	Parameter	Priorität	WCET	Deadline
1	60 %	300/500	-	0	0
2	65%	300/900	1	0	0
		300/950	2	0	0
3	69%	300/900	1	0	0
		300/1300	2	0	0
		300/1400	3	0	0

Tabelle 3: Isolation in Rate-Monotonic Scheduler - 10 Minuten Laufzeit

ist dann unter 69%. Keine Deadline oder WCET wurde überschrittet. Beim Test 4.2 wurde die Dom10 mit einer kleineren Slice als erwartet gestartet: die Domain verpasst fast immer ihre WCET aber beeinflusst nicht die anderen VMs. Genauso ist beim Test 4.3. In diesem Fall wurde die VM mit der höchsten Priorität falsch initialisiert und verpasst dann immer ihre WCET. Außerdem kann es beobachtet werden, dass im Fall einer Domain-Überlastung, hat die Priorität der VMs keine Bedeutung: die Domain bekommt nie mehr als ihre Slice und beeinflusst das gesamte System nicht.

Setup	Dom1	Dom1	Dom3	Dom4	Dom5	Dom6	Dom7	Dom8	Dom9	Dom10
Priorität	1	2	3	4	5	6	7	8	9	10
Korrekte Parameter	0-0	0-0	0-0	0-0	0-0	0-0	0-0	0-0	0-0	0-0
Priorität	1	2	3	4	5	6	7	8	9	10
Dom10 Overload	0-0	0-0	0-0	0-0	0-0	0-0	0-0	0-0	0-0	14537-0
Priorität	1	2	3	4	5	6	7	8	9	10
Dom1 Overload	14527-0	0-0	0-0	0-0	0-0	0-0	0-0	0-0	0-0	0-0

Tabelle 4: Domain-Überlastung: die anderen VM werden beim Setzen von falschen Parametern (WCET und Periode) nicht beeinflusst. Die Nummer zeigen wie viele WCETs bzw. Deadlines überschritten worden waren. (10 Min. Laufzeit)

Anschließend wurde das Verhalten im Fall einer System-Überlastung beobachtet (Tabelle 5). Bei Tests 5.1 und 5.2 wurden jeweils mit zwei VMs durchgeführt; in 5.1 überlastet die VM mit der höchsten Priorität das System und in 5.2 die mit der niedrigeren Priorität. In beiden Fällen wird immer die zweite VM verdrängt und damit die mit der höchsten Priorität ausgeführt.

Tests 5.3 und 5.4 hatten insgesamt 5 Domains. In 5.3 ermöglicht die Dom1 keine andere VM, da sie allein 60% der CPU braucht. In diesem Falls ist Dom1 die einzige VM die ohne Probleme gelaufen ist. Die angegebene 300 μs Slice ist die WCET der Task, aus diesem Grund verpasste die Dom2 nur einmal ihre Deadline und Dom3 insgesamt dreimal.

Test 5.5 zeigt zum Abschluss, dass wenn eine VM mit niedrigerer Priorität das System überlastet, nur diese Domain wird verdrängt und die schedulbaren Domains mit höchsten Prioritäten werden immer richtig gescheduled.

# Tasks	Core Auslastung per Domain	Parameter (WCET/Periode)	Priorität	# der WCET-Überschreitung	# der Deadline-Überschreitungen
2	75%	300/400	1	0	0
	60%	300/500	2	0	170508
2	30%	300/1000	1	0	0
	88%	2200/2500	2	0	54570
5	60%	300/500	1	0	0
	30%	300/1000	2	0	1
	27%	300/1100	3	0	3
	25%	300/1200	4	0	78576
	23%	300/1300	5	0	146862
5	16,6%	300/1800	1	0	0
	15,7%	300/1900	2	0	0
	15%	300/2000	3	0	0
	14,2%	300/2100	4	0	0
	69%	2200/3200	5	0	57545

Tabelle 5: System-Überlastung - 5 Minuten Laufzeit

9.2. Ausfallzeiten der verschiedenen Redundanzkonzepte

(Tim Harde) In diesem Kapitel sollen die Ausfallzeiten analysiert und bewertet werden, die sich durch die Verwendung der unterschiedlichen Hochverfügbarkeitskonzepte ergeben. Insbesondere steht hierbei der Vergleich der beiden Konzepte *Standalone Master* und *Master + Backup* (siehe auch Kapitel 5.2) im Vordergrund, wobei sowohl auf die unterschiedlichen Failover-Zeiten als auch auf die verursachte Systemlast näher eingegangen wird.

9.2.1. Versuchsaufbau

In diesem Abschnitt soll zunächst auf den Versuchsaufbau eingegangen werden.

Standalone Master Das Hochverfügbarkeitskonzept sieht bei Verwendung einer einzelnen Master-VM vor, dass diese im Falle eines Absturzes oder Host-Ausfalls auf einem anderen Host neu gestartet wird. Da diese neu gestartete virtuelle Maschine durch den Reboot über keinerlei Zustandsinformationen der alten VM verfügt, beginnt die Systemausführung bei Null.

Abbildung 41 zeigt den zeitlichen Ablauf des Versuchs. Die Master-VM wurde als *Standalone Master* (also ohne Redundanz) gestartet. Durch die Ausführung eines *xm destroy* wird ein Absturz der virtuellen Maschine simuliert, welches durch das lokale Monitoring erkannt und an das globale Monitoring kommuniziert wird. Durch die LMU wird schließlich ein Neustart der virtuellen Maschine ausgelöst, so dass der durch die VM bereitgestellte Dienst nach realtiv kurzer Downtime wieder verfügbar ist.

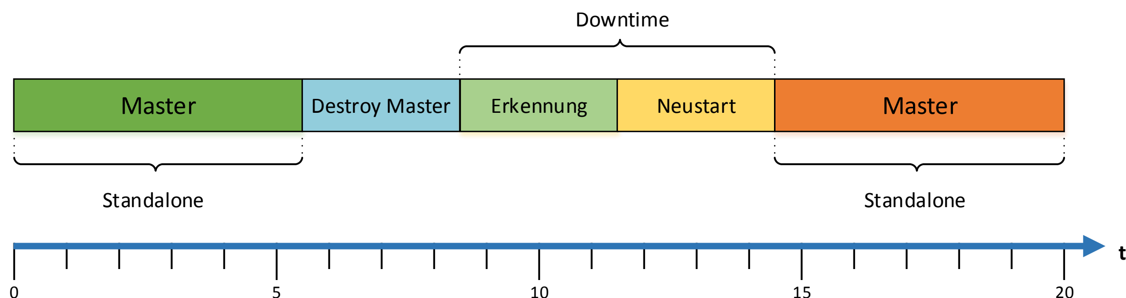


Abbildung 41: Zeitlicher Ablauf (Standalone Master).

Die Bestimmung der Downtime erfolgt über einen einfachen Mechanismus (siehe Abbildung 42). Für die Tests wurden spezielle Images der entsprechenden Betriebssystemversionen (Alpine Linux, CiAO, Tinycore und Ubuntu) erstellt, die nach dem Booten einen TCP-Port öffnen und auf eingehende Verbindungen warten. Nach dem Starten der VM wird gewartet, bis ein zwischen dem Benchmark-Skript und der virtuellen Maschine ein TCP Drei-Wege-Handshake erfolgreich durchgeführt werden konnte - die VM befindet sich in einem betriebsbereiten Zustand. Nach dem *xm destroy* wird ein Timer gestartet, der angehalten wird, sobald wieder ein TCP-

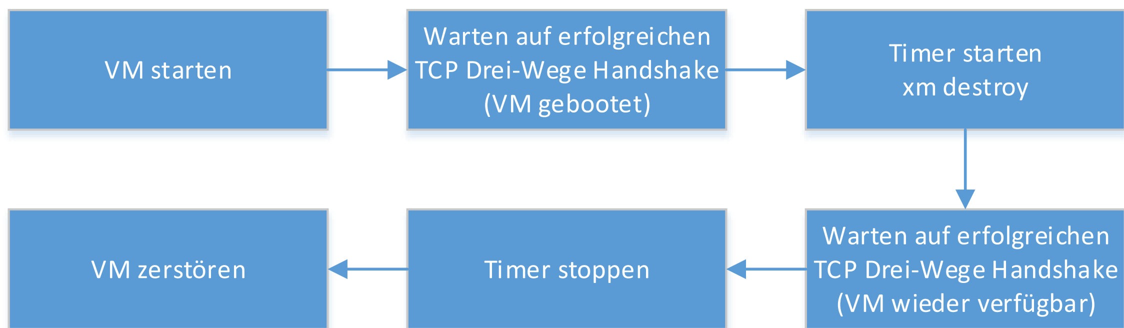


Abbildung 42: Versuchsablauf (Standalone Master).

Handshake zu Stande gekommen ist. Auf diese Weise kann die genaue Downtime der virtuellen Maschine (also die effektive Wiederherstellungszeit) ermittelt werden.

Master + Backup (Remus) Bei Verwendung des Redundanzkonzeptes *Master + Backup* wird *Remus* eingesetzt, um in regelmäßigen Abständen den aktuellen Systemzustand der Master-VM zu einer Backup-VM zu replizieren. Diese Backup-VM kann dann im Falle eines Ausfalls der Master-VM die Aufgaben übernehmen.

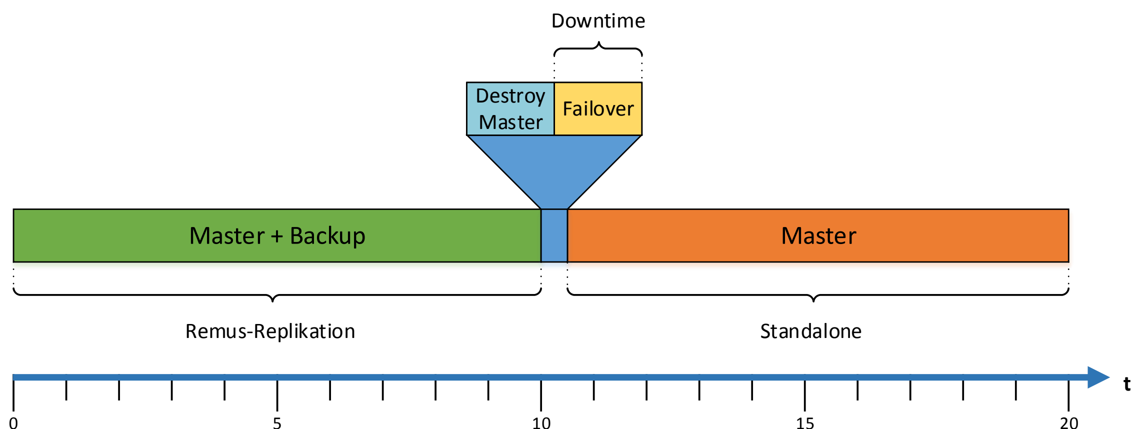


Abbildung 43: Zeitlicher Ablauf (Master + Backup).

Der zeitliche Ablauf (siehe Abbildung 43) unterscheidet sich hier grundlegend von dem zuvor erläuterten Redundanzkonzept. Beide virtuellen Maschinen (Master und Backup) werden gestartet; auch in diesem Fall wird (zehn Sekunden nach Start des Tests) ein Absturz der Master-VM durch die Ausführung eines *xm destroy* simuliert. Durch die laufende Remus-Replikation kommt es zu einem *Failover*, die Backup-VM

übernimmt die Rolle der Master-VM. Die Wiederherstellung der Redundanz (durch Erstellung einer neuen Remus-Replikation) soll an dieser Stelle nicht näher betrachtet werden, da hier die gleichen Kommunikationswege und Erkennungsmechanismen wie bei der Evaluation der Ausfallzeiten eines Standalone Masters verwendet werden und somit die eine ähnliche Wiederherstellungszeit angenommen werden kann.

Die Bestimmung der Downtime erfolgt in disem Fall durch die Verwendung eines etwas komplizierteren Mechanismus (siehe Abbildung 44). Für diese Tests wurden ebenfalls spezielle Images der entsprechenden Betriebssystemversionen (Alpine Linux, CiAO, Tinycore und Ubuntu) erstellt.

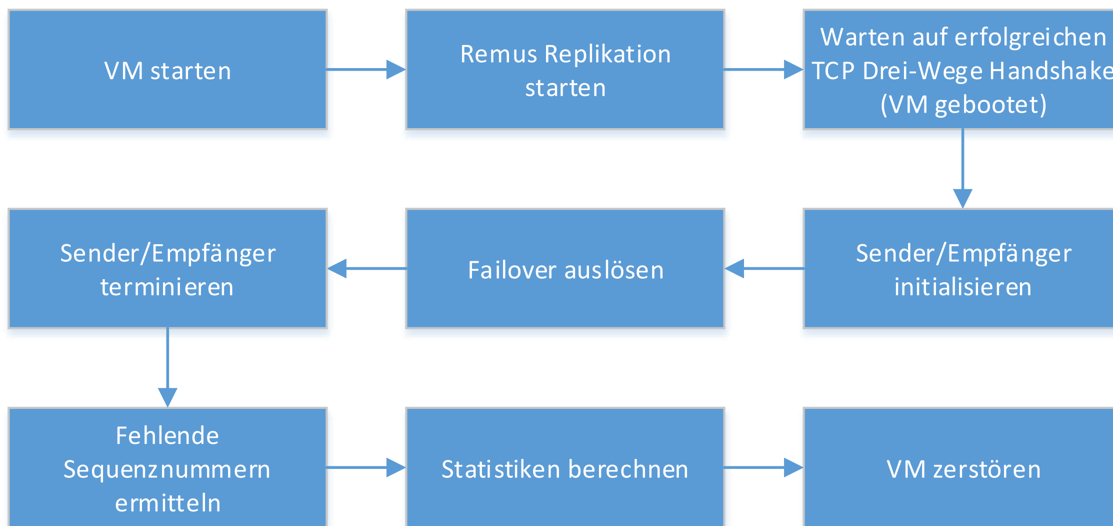


Abbildung 44: Versuchsablauf (Master + Backup).

Nach dem Booten wird erneut ein TCP-Port geöffnet und auf eingehende Verbindungen gewartet; nach einem erfolgreichem TCP Drei-Wege-Handshake ist sichergestellt, dass sich die VM in einem betriebsbereiten Zustand befindet. Anschließend wird von einem Sender-Skript eine Sequenz von Paketen über einen UDP-Socket an die virtuelle Maschine versendet - diese Pakete enthalten als Payload die aktuelle Sequenznummer. Innerhalb der VM wird das entsprechende Paket verarbeitet, die Sequenznummer extrahiert und an die IP zurückgeschickt, von der das empfangene Paket versendet wurde. Dieses von der VM verschickte Paket wird von einem Empfänger-Skript verarbeitet und sowohl die empfangene Sequenznummer als auch die Empfangszeit protokolliert.

Durch die erhobenen Daten lässt sich genau bestimmen, welche Sequenznummern durch den Failover verloren gegangen sind. Des Weiteren ist es möglich, die genaue Downtime der virtuellen Maschine zu bestimmen.

9.2.2. Darstellung der Ergebnisse und Auswertung

Standalone Master Insgesamt wurden für jedes der evaluierten Host-Betriebssysteme (Alpine, CiAO, Tinycore und Ubuntu) je 100 Testdurchläufe durchgeführt und ausgewertet. Das System arbeitete dabei insgesamt sehr zuverlässig; jeder Ausfall einer virtuellen Maschine wurde vom Monitoring erkannt und die ausgefallene VM daraufhin von der LMU erfolgreich neu gestartet.

Abbildung 45 zeigt die Auswertung zu den durchgeführten Versuchen. Insgesamt zeigen sich keine signifikanten Unterschiede in der Wiederherstellungszeit zwischen den einzelnen Host-Betriebssystemen, da ein Großteil der benötigten Zeit für die Erkennung der Fehlersituation, Kommunikation zwischen den einzelnen Systemkomponenten und die schlussendlich erfolgte Reaktion auf den entsprechenden Ausfall benötigt wurde.

Lediglich Alpine Linux kann sich mit einer deutlich niedrigeren maximalen Wiederherstellungszeit von knapp über 9 Sekunden von den anderen drei analysierten Gastbetriebssystemen absetzen. Des Weiteren ist die Standardabweichung zwischen den einzelnen Wiederherstellungszeiten im Vergleich zu den anderen Betriebssystemen deutlich geringer.

Master + Backup (Remus) Insgesamt wurden für jedes Host-Betriebssystem (Alpine, CiAO, Tinycore und Ubuntu) je 10 Testdurchläufe mit unterschiedlichen Remus-Replikationsintervallen (40, 50, 60, 80, 100 und 200 Millisekunden) und Zwischenankunftszeiten der Pakete (10, 5, 4, 3, 2, 1, 0.5, 0.25 Millisekunden) durchgeführt und ausgewertet.

Abbildung 46 und Tabelle 6 zeigen die ermittelten Failover-Zeiten. Insgesamt zeigt sich, dass die voll paravirtualisierten Gastbetriebssysteme (Alpine und Ubuntu) gegenüber den HVM-Gastbetriebssystemen (CiAO und Tinycore) im Mittel erheblich stabilere Failover-Zeit besitzen.

Alpine-Linux verfügt über eine sehr gute Performance - insgesamt lässt sich durch eine Erhöhung des Replikationsintervalls die durchschnittliche Failover-Zeit immer weiter verbessern. Die Failover-Zeit von Ubuntu verschlechtert sich hingegen deutlich, wenn das Replikationsintervall weniger als 80 Millisekunden beträgt.

Tinycore schneidet im Vergleich zu CiAO als HVM-Gastbetriebssystem im Mittel deutlich besser ab. Des Weiteren überschreiten die maximalen Failover-Zeiten die Marke von einer Sekunde nur knapp.

Abschließend soll in diesem Kapitel noch ein Blick auf die durch die Remus-Replikation verursachte Systemlast geworfen werden. Detaillierte Diagramme zu den einzelnen Versuchen finden sich im Anhang G.

CPU-Auslastung Insgesamt zeigt sich, dass die paravirtualisierten Systeme deutlisch weniger CPU-Last verursachen. Bei einem Replikationsintervall von 40 ms benötigen beide während der Replikation zwischen 15 und 20% der Rechenzeit eines

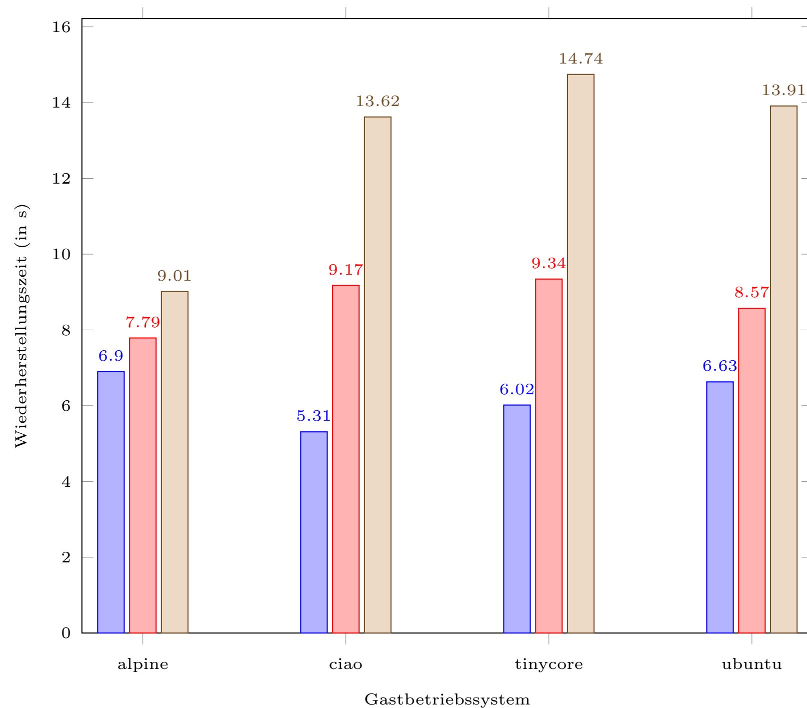


Abbildung 45: Minimale (blau), durchschnittliche (rot) und maximale (braun) Wiederherstellungszeit in Abhängigkeit vom verwendeten Gastbetriebssystem.

Intervall	Betriebssystem	Minimum (ms)	Durchschnitt (ms)	Maximum (ms)
40	Alpine	78.71	95.00	143.39
	CiAO	210.87	639.97	1917.86
	Tinycore	216.62	371.28	1266.22
	Ubuntu	97.97	265.61	4709.71
50	Alpine	78.35	97.73	127.18
	CiAO	243.04	749.30	1237.16
	Tinycore	228.13	535.89	1207.73
	Ubuntu	98.37	264.61	2709.71
60	Alpine	77.72	106.16	328.12
	CiAO	219.66	1000.74	3427.77
	Tinycore	220.65	391.67	1081.84
	Ubuntu	99.87	214.15	3775.43
80	Alpine	80.03	122.04	168.56
	CiAO	214.70	1038.61	3363.13
	Tinycore	236.74	480.78	1100.44
	Ubuntu	100.39	126.04	169.46
100	Alpine	86.25	137.25	274.95
	CiAO	241.90	718.50	1097.80
	Tinycore	209.25	385.86	1098.91
	Ubuntu	101.32	144.05	179.34
200	Alpine	81.02	165.57	463.85
	CiAO	221.77	552.78	1163.55
	Tinycore	204.32	396.87	1159.42
	Ubuntu	99.52	141.14	262.39

Tabelle 6: Minimale, durchschnittliche und maximale Failover-Zeiten der einzelnen Gastbetriebssysteme in Abhängigkeit vom verwendeten Remus-Replikationsintervall.

einzelnen Kerns, während die Auslastung bei einem Replikationsintervall von 200 ms auf Werte zwischen 5 und 10% zurückgeht. Diese CPU-Auslastung ist dabei beinahe unabhängig von der verwendeten Zwischenankunftszeit der einzelnen Netzwerkpakete.

Die HVM-Gastbetriebssysteme zeigen bei niedrigeren Zwischenankunftszeiten (>2 ms) ein ähnliches Verhalten; auch hier liegt die CPU-Auslastung bei einem Replikationsintervall von 40 ms bei ca. 20%. Bei höheren Zwischenankunftszeiten ändert sich dieses Verhalten allerdings immens. CiAO verursacht bei einem Intervall von 40 ms bei einer Zwischenankunftszeit von 1/0.5/0.25 ms eine CPU-Auslastung von

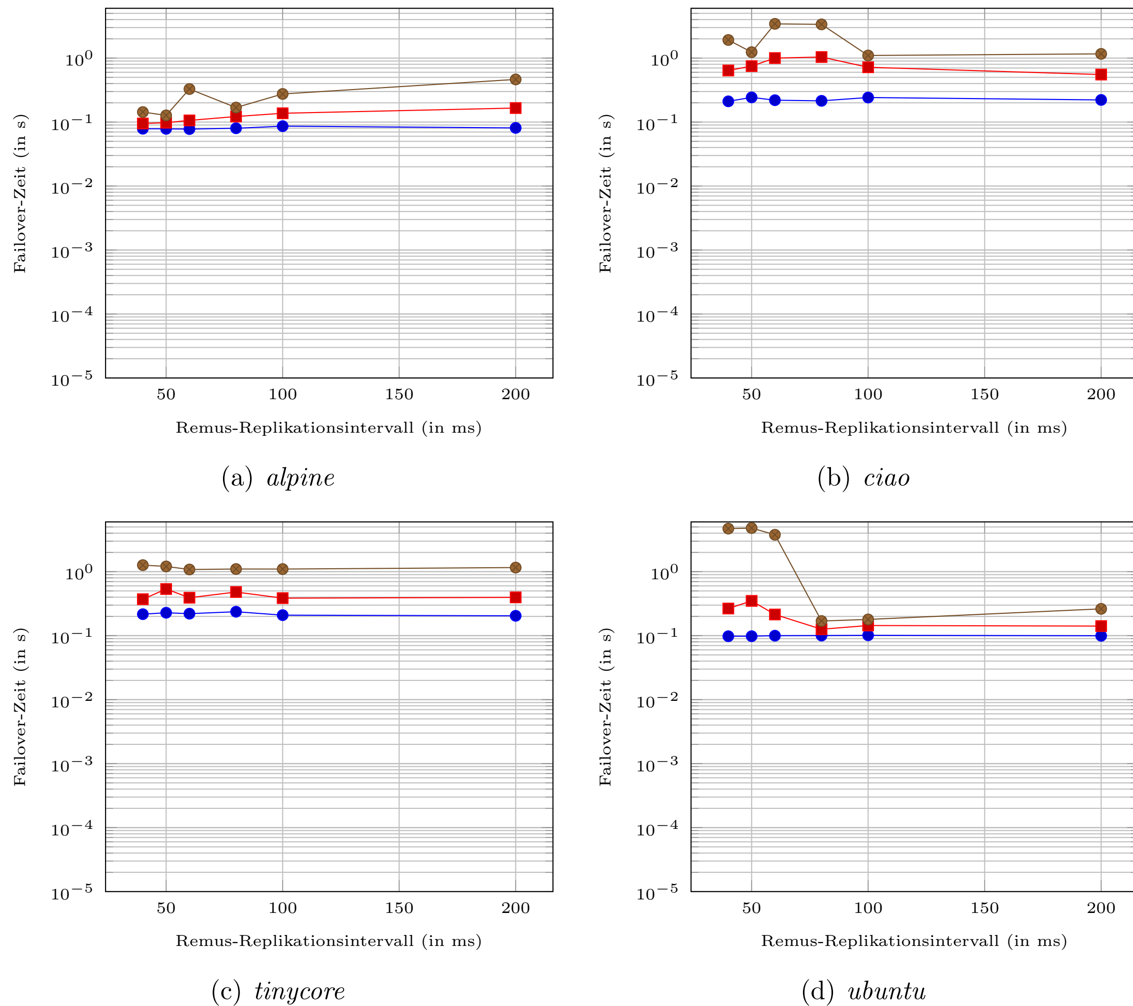


Abbildung 46: Minimale (blau), durchschnittliche (rot) und maximale (braun) Failoverzeiten der einzelnen Gastbetriebssysteme in Abhängigkeit vom verwendeten Remus-Replikationsintervall.

40/60/95%, bei einem Replikationsintervall von 200 ms beträgt die Auslastung 30/50/95%.

Tinycore zeigt ein ähnliches Verhalten, allerdings ist der beobachtete Effekt nicht ganz so ausgeprägt. Hier wird bei Zwischenankunftszeiten von 0.001/0.0005/0.00025 und einem Replikationsintervall von 40 ms eine CPU-Auslastung von 25/40/50% erreicht; bei einem Replikationsintervall von 200 ms eine CPU-Auslastung von 10/15/30% verursacht.

Die Ursache für dieses Verhalten ist vermutlich in der Emulation der Intel E1000-Netzwerkkarte durch die qemu-Instanz zu suchen, die für jede HVM in der Dom0 ausgeführt werden muss. Des Weiteren wird durch alle Gastbetriebssysteme zum Zeitpunkt des Failovers (Sowohl auf Master- als auch auf dem Backup-Host) einen deutlicher Peak in der CPU-Auslastung verursacht.

Netzwerk-Auslastung Die Netzwerkauslastung zeigt im Vergleich zur CPU-Auslastung ein ziemlich konträres Verhalten. Alpine und Ubuntu verursachen eine deutlich höhere Auslastung der Netzwerk-Verbindung durch die Remus-Replikation.

Die Netzwerkauslastung liegt bei Alpine bei einem Replikationsintervall von 40 ms bei 200-300 Mbit/s und bei eine bei einem Replikationsintervall von 200 ms bei 50-70Mbit/s, bei Ubuntu betragen diese Werte sogar 225-400 Mbit/s (bei 40 ms) und 60-100 Mbit/s (bei 200 ms).

CiAO und Tinycore schneiden in diesem Belang deutlich besser ab. Liegt die von Tinycore verursachte Auslastung bei 60-100 Mbit/s (bei 40 ms) und 15-30 Mbit/s (bei 200 ms), so liegt die durch CiAO verursachte Last mit 25 Mbit/s (bei 40 ms) bzw. 5 Mbit/s (bei 200 ms) noch deutlich darunter.

Die deutliche niedrigere Datenrate, die zur Übertragung des Systemzustandes benötigt wird, ist offensichtlich in der Minimalität von von CiAO und Tinycore im Vergleich zu Alpine bzw. Ubutnu zu suchen.

9.3. Auslösezeiten der Beispielapplikation

(Dominic Wirkner) In diesem Kapitel wird diskutiert, inwiefern die Umsetzung des Distanzschutzes (siehe Kapitel 7) und dessen Ausführung auf der in diesem Projekt entwickelten Plattform den Anforderungen einer realen Umgebung entsprechen kann. Der Distanzschutz wurde in diverse Gastsysteme integriert und dessen Antwortzeitverhalten mit diversen Messungen untersucht, welche im folgenden beschrieben werden.

9.3.1. Versuchsaufbau

Die Applikation wurde in verschiedene Gastsysteme integriert, um einerseits einen Vergleich zwischen Hardware-Virtualisierung (HVM) und Paravirtualisierung (PV) zu ermöglichen. Andererseits sollte ein Betriebssystem, welches nur aus einem minimalen Kernel besteht, einem vollwertigen Linux-System gegenübergestellt werden. Die Tabelle 7 gibt einen Überblick über die verwendeten Gastsysteme.

	CiAO	Mini-OS	TinyCore	Ubuntu	Alpine
Virtualisierung	HVM	PV	HVM	PV	PV
maßgeschneiderter Kernel	ja	ja	nein	nein	nein
Image-Größe	2 MB	1 MB	50 MB	1,5 GB	260 MB
Hypercalls möglich	ja	ja	nein	ja	ja
Migration möglich	ja	nein	ja	ja	ja

Tabelle 7: Verwendete Betriebssysteme während der Evaluation

Die Gastsysteme wurden nicht redundant, d.h. ohne eine ständige Replikation durch Remus, auf einem der Hosts ausgeführt und hatten gleiche Voreinstellungen für Periode ($250\,\mu$s) und WCET ($100\,\mu$s). Für die Betriebssysteme TinyCore, Alpine und Ubuntu wurden zwei Versionen der Paketübertragung implementiert, um diesen zusätzlichen Aspekt zu vergleichen: Eine Version mit dem TCP-Protokoll und blockierendem Warten und eine Version mittels dem UDP-Protokoll und nicht-blockierendem Warten. Leider ermöglicht der Netzwerk-Stack in der verwendeten Version von Mini-OS kein nicht-blockierendes Warten, so dass hier auf die UDP-Version verzichtet werden musste.
Die Auslösezeiten wurden von einem in Python implementierten Skript ermittelt, welches auf einem separaten Rechner ausgeführt wurde und sowohl als Sensor wie auch als Aktor diente. Das Betriebssystem dieses Rechners war Linux in Form von Ubuntu 12.10. Andere Dienste wurden auf diesem Rechner nicht ausgeführt. Die Spezifikation der verwendeten Hardware von Virtualisierungs- und Sensor-Host sind in Tabelle 1 aufgeführt.

9.3.2. Ablauf der Messungen

Zunächst wurde das Gastsystem auf einem der Plattform zugehörigen Hosts gestartet. Da für diesen Versuch nur die Auswirkungen des lokalen Schedulings wichtig

waren, geschah dies nicht über das Frontend, sondern per Kommandozeile. Anschließend wurde das Sensor-Aktor-Skript gestartet. In der Initialisierungsphase erstellt das Skript 80 Messwerte für Spannung und Strom; jeweils sowohl für den Normalzustand als auch für die Kurzschluss-Situation.
Im Anschluss beginnt der Sensor zunächst 800 Pakete des Normalzustandes zu senden (10 Schwingungen = 200 ms). Die Anzahl an Paketen ist mehr als ausreichend, um die Applikation zu initialisieren und die ersten Berechnungsergebnisse zu empfangen. Diese ersten Ergebnisse bestätigen den Normalzustand und garantieren zusätzlich, dass die Applikation bereit für die Erkennung eines Kurzschlusses ist.

Danach wird eine Zeitmessung gestartet und der Applikation werden Strom- und Spannungswerte eines Kurzschlusses übermittelt. Sobald das Signal eines erkannten Kurzschlusses empfangen wird, wird die Messung gestoppt. Die gemessene Zeit wird für eine spätere Auswertung zwischengespeichert und der Applikation werden wieder Werte des Normalzustandes gesendet. Damit die Applikation immer gleiche Berechnungen durchführt, sind die Messwerte nummeriert und ein Kurzschluss wird immer mit dem ersten Paket der Kurzschluss-Sequenz gestartet. D.h. auch wenn bereits 800 Pakete der Normalsequenz gesendet wurden und eine neue Messung erfolgen kann, wird die Normalsequenz erst beendet, bevor das erste Paket der Kurzschluss-Sequenz gesendet wird.
Der Versuch endet nach einer vorher definierten Zeit und die gesammelten Auslösezeiten werden ausgegeben. Für diese Evaluation wurde ein Zeitraum von zwei Stunden eingestellt.

9.3.3. Auswertung

Die Tabelle 8 zeigt die Ergebnisse eines Testlaufs über zwei Stunden. Jede implementierte Kombination von Betriebssystem (TinyCore, Ubuntu, Alpine und Mini-OS) und Netzwerk-Protokoll (TCP oder UDP) erkannte während dieser Zeit ca. 29000 Kurzschlüsse. Aus den Daten lassen sich folgende Schlüsse ziehen:
Jede Implementierung benötigt mindestens zwischen 19 bis 20 ms für eine Erkennung. Das bedeutet, dass in dem oben beschriebenen Szenario mindestens 75 Messwerte nötig sind, damit der berechnete Impedanz-Phasor unter den Schwellwert fällt.

Die Daten belegen auch, dass die Verwendung von UDP und TCP keinen entschei-

Betriebssystem	Empfang	Min	Avg	Max	# Auslösungen	über 30 ms
TinyCore (HVM)	TCP	19,9	21,1	45,9	29521	3
	UDP	19,4	20,5	46,6	29521	6
Ubuntu (PV)	TCP	19,0	19,3	65,2	29337	3
	UDP	19,1	19,5	39,5	29228	2
Alpine (PV)	TCP	19,1	19,3	38,8	29315	3
	UDP	19,1	19,5	39,7	29246	2
Mini-OS (PV)	TCP	19,0	19,2	35,8	29355	2

Tabelle 8: Ergebnisse eines 2 Std. Testprogramms

denden Unterschied macht. Ebenso verhält es sich mit der Verwendung eines maßgeschneiderten Kernels gegenüber eines umfangreichen Linux-Betriebssystems. Eher wirkt sich die Form der Virtualisierung auf die Auslösezeiten aus: TinyCore löst im Vergleich zu den paravirtualisierten Implementierungen im Durchschnitt 1 bis 2 ms später aus. Insgesamt liegen die gemessenen Zeiten jedoch sehr dicht beieinander.
Leider musste bei dieser Messung das Betriebssystem CiAO außen vor gelassen werden. Dieses hielt der Übertragung von 4000 Paketen pro Sekunde nicht stand und stürzte in diesem Test reproduzierbar ab. Es wurden daher Messungen mit geringeren Übertragungsraten durchgeführt. Näheres dazu im späteren Kapitel 9.4.

Rätselhafte Ausreißer Ein Aspekt der Ergebnisse konnte bisher noch nicht vollständig geklärt werden. Die maximalen Auslösezeiten sind im Vergleich zum Durchschnitt sehr groß. Betrachtet man z.B. den maximalen Wert von 65,2 ms beim paravirtualisierten Ubuntu, stellt sich die Frage, wie es möglich sein kann, dass selbst nach einer Übertragung von mehr als drei vollen Schwingungen eines Kurzschlusses keine Auslösung stattfand. In einigen anderen Testläufen (z.B. über 24 Stunden) wurden sogar dreistellige Maxima gemessen. Es lässt sich bisher keine obere Schranke für eine Auslösung ausmachen, obwohl diese theoretisch bei etwas über 20 ms liegen müsste, wenn man die Übertragung von 80 Messwerten, deren Berechnung und den Transport der Rückantwort mit einbezieht.
Zur Untersuchung dieses Phänomens wurde daher ein lokaler Testlauf über zwei Stunden durchgeführt. Bei diesem Testlauf wurde eine Implementierung des Distanzschutzes in C nativ auf einem Linux-System ausgeführt. Gleichzeitig dazu wurde das Python-Skript zur Messung ausgeführt und die Pakete wurden nur lokal übertragen.

Die Ergebnisse zeigten ähnliche Werte, wie es bei einer virtualisierten Implementierung der Fall ist.
Daraus lässt sich folgern, dass eine derartige Verzögerung nicht in der Paketübertragung durch das Netzwerk oder der Virtualisierungstechnologie XEN zu vermuten ist. Vielmehr wird der Fehler in der Zeitmessung an sich liegen, denn die Ausführung eines Python-Skripts in einem nicht echtzeitfähigen Betriebssystem ist eventuell nicht zuverlässig genug, um exakte Messungen im Bereich weniger Millisekunden vorzunehmen.
Die bisherigen Ergebnisse zeigen jedoch, dass ein Kurzschluss zu 99,9999% innerhalb der geforderten 30 ms erkannt werden kann und dabei das genutzte Gastsystem, sowie die Paketübertragungs- und Virtualisierungstechnologie prinzipiell keinen entscheidenden Unterschied ausmachen.

Fazit Für eine Anwendung in der Architektur der Projektgruppe eignen sich bisher nur die Implementierungen auf Basis von Ubuntu und Alpine. Das Betriebssystem CiAO ist leider nicht stabil genug für den Dauerbetrieb. Auf der anderen Seite war es der Projektgruppe bis zum Abschluss nicht möglich eine Unterstützung für Hypercalls in TinyCore zu implementieren. Mini-OS scheidet auf Grund der Tatsache aus, dass es durch die fehlende Möglichkeit der Migration nicht redundant und damit hochverfügbar ausgeführt werden kann.

9.4. Auslösezeiten der Beispielapplikation in CiAO

(Gregor Kotainy) An dieser Stelle sollen die in Kapitel 9.3 fehlenden Auslösezeiten von *CiAO* erläutert werden.

Objektiv betrachtet bietet das Betriebssystem alle für die Implementierung der Distanzschutzapplikation notwendigen Voraussetzungen. Es ist - neben dem paravirtualisierten Mini-OS - das kleinste Hardware-virtualisierte Gastsystem mit einer maximalen Gesamtgröße von 2, 5 MB. Damit liegt es um Faktor 20 unter der Größe des ebenfalls HW-virtualisierten TinyCore, dessen Kernel nicht minimal ist. Außerdem ist es uns gelungen aus *CiAO* heraus *Hypercalls* durchzuführen.

Subjektiv gesehen besteht das einzige Manko darin, dass das System derzeit einen Flaschenhals bei der Behandlung von Netzwerk-*Interrupts* hat, die leider zu lan-

ge behandelt werden. Bei der Untersuchung der Technologien im ersten Semester sind wir nicht davon ausgegangen, dass genau dies ein Problem darstellen würde. Schließlich kann der vorhandene IP-Stack sowohl UDP als auch TCP Pakete senden und empfangen. Es sollte sich erst später herausstellen, dass eine Beispielapplikation gewählt wurde, die einer Empfangsrate von $250\mu s$ standzuhalten hat. Wenn man das Betriebssystem *CiAO* allerdings mit einer derartig hohen Senderate unter Beschuss nimmt, stürzt es irgendwann ab, weil der Puffer für die Speicherung der Netzwerkpakete über läuft und die Behandlung der *Interrupts* die gesamte Laufzeit in Anspruch nimmt, sodass es nicht zum Verarbeiten der bereits empfangenen Pakete im Anwendungstask kommen kann. Zwar lässt sich dieses Problem jederzeit reproduzieren, wobei der Zeitpunkt des Absturzes variabel ist und das *Debuggen* unter diesem Vorwand erschwert wird.

Schade, dass es uns von daher nur möglich war mit einer geringeren Paketrate zu evaluieren, da die aus den Ausgaben des Schedulers gemessene Algorithmus-Berechnungslaufzeit des Phasors nur $30\mu s$ dauerte. Subtrahiert man diesen Wert von den eintreffenden Paketen, blieben $220\mu s$ zum Abholen der neuen Daten. Eine weitergehende Untersuchung des Problems im IP-Stack ist in Kapitel 10 zu ungelösten Problemen aufgeführt. Die besten Ergebnisse konnten wir in *CiAO* erzielen, indem wir den Empfangsmodus auf TCP gestellt haben. Dieser Umstand hat zwar nicht geholfen, das oben geschilderte Problem der langen *Interrupt*-Behandlungen zu lösen, führte jedoch dazu, dass bei halber Senderate des Sensors alle Pakete empfangen werden konnten. Einen Eindruck soll die folgende Ausgabe des Sensors geben:

Insgesamt konnten keine sinnvollen, „stabilen“ Tests durchgeführt werden, die länger als 5-10 Minuten liefen. Wir haben die Raten $1000\mu s$, $750\mu s$ und $500\mu s$ testen können. Die Periode ist gleich der Senderate, da blockierendes TCP verwendet wurde. Die Slice musste bei den schnellen Senderaten auf die Periode gesetzt werden, da der Test sonst nicht funktionierte und *CiAO* abstürzte. Da die maximale Slice nur 70% der Periode sein darf, ist nur der erste Test mit einer Periode von $1000\mu s$ legitim.

Rate 1000μs,	**Periode 1000μs,**	**Slice 700μs, Testing time 5min**
	Test Details	
75-80ms:	309	(99.7%)
80-85ms:	1	(0.3%)
	Test Summary	
Min: 75.829	Avg: 76.701	Max: 80.304
0 of 310 Failures	(0.0%) detected after Deadline of 120.0 ms	
	Network Stats	
0 of 297635	(0.0%) Packets lost!	

Rate 750μs,	**Periode 750μs,**	**Slice 700μs, Testing time 5min**
	Test Details	
55-60ms:	350	(84.7%)
60-65ms:	61	(14.8%)
65-70ms:	2	(0.5%)
	Test Summary	
Min: 56.993	Avg: 58.379	Max: 66.008
0 of 413 Failures	(0.0%) detected after Deadline of 90.0 ms	
	Network Stats	
0 of 396583	(0.0%) Packets lost!	

Rate 500μs,	**Periode 500μs,**	**Slice 500μs, Testing time 30s**
	Test Details	
35-40ms:	60	(98.4%)
40-45ms:	1	(1.6%)
	Test Summary	
Min: 38.601	Avg: 39.328	Max: 40.302
0 of 61 Failures	(0.0%) detected after Deadline of 60.0 ms	
	Network Stats	
0 of 59428	(0.0%) Packets lost!	

10. Ungelöste Probleme

In diesem Kapitel sollen ungelöste und bestehende Probleme aufgeführt werden, deren Lösung nicht bekannt ist, oder die nur sehr zeitaufwändig mit nötigem Spezialwissen zu beheben sind.

10.1. Load Management Unit

Beim starten von vielen Master und Backup Diensten, tritt vermehrt ein REMUS_-FAILOVER auf. Dies bedeutet, dass der Remus Prozess abgestürzt ist und dadurch von der aktuellen Master VM kein Backup erzeugt wird. Dies wird der LMU durch ein REMUS_FAILOVER signalisiert. Bei einem REMUS_FAILOVER verfährt die LMU wie bei einem VM_DOWN. Es wird also zuerst das Backup beendet, dann eine neue Replikation von der Master VM gestartet. Dieser Prozess dauert ca. 15-20 Sekunden.

Durch das Fluten der Event-Pipe mit der Fehlermeldung kommt es zu einer Inkonsistenz zwischen den aktuell laufenden Diensten und den Diensten, die die LMU in ihrer Datenstruktur gespeichert hat. Um das Problem zu lösen ist eine Funktion vorhanden, die von allen Hosts die aktuell laufende Konfiguration abfragt und gegeben falls neue Dienste startet oder zu viel Laufende Dienste beendet. Diese Funktion widerspricht jedoch dem Konzept, nach dem das System konzipiert wurde. Daher ist die Funktion deaktiviert.

10.2. CiAO

Aufgrund der Sonderbehandlung der Auslösezeiten des Distanzschutzes in *CiAO*, die aus Kapitel 9.4 hervorgeht, sollen die ungelösten Probleme und Beobachtungen bezüglich des Betriebssystems an dieser Stelle festgehalten werden.

Konfiguration Für die erzielten Ergebnisse wurde immer eine minimale Konfiguration des Betriebssystems verwendet, wozu die sämtliche Abschaltung von zusätzlichen Features gehört (z.B. ICMP Echo Reply). An sehr vielen Stellen fehlte diesbezüglich

eine Dokumentation der Features.

Allgemeine Implementierung Die Implementierung der Netzwerk-Sockets in *CiAO* entspricht nicht der in modernen Betriebssystemen verwendeten *Berkeley sockets*[26] (auch *BSD sockets* genannt), was den Einstieg im Umgang mit Sockets erschwerte [35]. TCP brachte Schwierigkeiten bei der Verwendung mehrerer Tasks in einer Applikation, so dass das System einfror.

Hinzu kommt die manchmal fragliche Verwendung von Aspekten und die Verteilung jener Quelltext-Dateien über mehrere Verzeichnisse. So musste zunächst aufwändig festgestellt werden, welche Funktionalität eine gewobene Klasse bereitstellt.

Distanzschutz Die Erkennung der Auslösung des Distanzschutzes, die mit einer Senderate von $1ms$ pro Paket des Sensors erzielt wurde, ist um Faktor vier schlechter als die Echtzeiterkennung mit einer Senderate von $250\mu s$ pro Paket. Da sich alle langsameren Senderaten erfolgreich erkennen lassen, lässt sich dementsprechend eine obere Schranke von Faktor vier klar definieren. Sollte der VM bei einer Paketrate von $500\mu s$ die CPU zu 100% zur Verfügung stehen, die WCET und Periode der Paketrate entsprechen, lässt sich sogar eine Erkennung erreichen, die „nur“ Faktor zwei schlechter als Echtzeit ist. Ein schnelleres Ergebnis konnte aufgrund der vielen möglichen Störungsverursacher nicht erzielt werden, da der Grund nicht gefunden werden konnte.

Beobachtungen und mögliche Fehlerquellen Zum einen ist das Gastsystem HW-virtualisiert und verfügt damit über keinen paravirtualisierten Netzwerktreiber. Zum anderen kann man anhand der *xm dmesg* Ausgaben - in Listing 26 angedeutet - des *Hypervisors* in der *Dom0* sehen, dass die VM beim Empfang eines Pakets ca. acht mal geweckt wird.

```
1 [...]
2 in fp_vcpu_wake Domain: 3,Current CPU and Period used: 32917|72816
3 [...]
```

Listing 26: Ca. acht vcpu_wake der Domain beim Empfang eines Netzwerkpaketes

[26] mehr zu Berkeley sockets: http://en.wikipedia.org/wiki/Berkeley_sockets
Siehe auch Linux man pages: http://linuxmanpages.com/

Da dies in anderen Gastsystemen nicht der Fall ist, kann davon ausgegangen werden, dass das Problem entweder an der Implementierung des Intel E1000 Netzwerktreibers oder einem Polling-Konzept / System-Trap liegt.

Die obigen Ergebnisse wurden nur mit Empfangsprotokoll TCP erreicht. Da eine Migration einer via TCP empfangenden Applikation fehlschlägt, blieb nur der Empfang per UDP, welcher zu großen Paketverlusten geführt hat, die in einem minimalen Task gemessen wurden. Wir versuchten zunächst drei Strategien zu entwickeln, die einen Paketverlust verkraften können.

Minimaler Task Um mögliche Implementierungsfehler der komplexeren Distanzschutz-Applikation auszuschließen wurde ein auf das Minimum reduzierter Task in *CiAO* entwickelt, der darauf spezialisiert war Pakete in einer Endlosschleife zu empfangen. Dabei wurden die Protokolle UDP und TCP in getrennten Applikationen getestet. Der Test zeigte, dass die VM bereits nach wenigen Minuten empfangen abgestürzt ist und die entwickelten Strategien zum Umgang mit Verlusten nutzlos geworden sind.

Eine Ausgabe im IP-Stack bewies, dass der auf 10.000 Pakete aufgestockte Ringpuffer bereits nach kurzer Zeit voll war, da die VM nur mit dem Interrupt beschäftigt war und eine Verarbeitung wegen höherer Priorität der Sonderbehandlung nicht stattfinden konnte. Des Weiteren konnte festgestellt werden, dass UDP bei einer Senderate von $250\mu s$ einen Paketverlust von über 8% aufwies. Dies deckt sich in etwa mit den Distanzschutz Tests. Außerdem zeigt ein Blick in den Hauptzweig der *CiAO*-Repository, dass erst kurz zuvor ein *Memory Leak* beim Freigeben von Speicher allozierter UDP Pakete beseitigt wurde. TCP hatte zwar keinen Paketverlust und erlaubte das Wiederverwenden eines eigenen Puffers, was mögliche *Memory Leaks* umging, aber auch keinen Erfolg brachte, da bei der Paketrate in Echtzeit-Geschwindigkeit ein sofortiger Absturz folgte.

11. Projektorganisation

(Dominic Wirkner) Aufgrund der Zahl der Projektmitglieder und der Dauer des Projektes ist es unumgänglich eine Form der Organisation zu etablieren. Zudem ist es ein

zusätzliches Lernziel der Veranstaltung, neben den Inhalten des Projektthemas, die Organisation von IT-Projekten zu üben. Im Folgenden werden daher die Kernpunkte der durchgeführten Projektorganisation erläutert.

Orientierungsphase Nachdem die Auswahl der Mitglieder beendet war, wurden während eines ersten Treffens Vortragsthemen vergeben, welche im wesentlichen den theoretischen Grundstock für die Durchführung des Projektes bildeten.
Die Gruppe entschied sich dazu, diese Vorträge im Rahmen einer Seminarfahrt zu hören, welche an einem Wochenende im nordrhein-westfälischen Haltern stattfand. Neben der Vermittlung von Grundwissen bot dies eine ausgezeichnete Möglichkeit des Kennenlernens.
Zum Ende der Seminarfahrt wurde ein Projektleiter gewählt, dessen Aufgabe es war, neben der Organisation der Teams, den Kontakt zu den Betreuern zu halten. Des weiteren wurde zwei wöchentliche Termine vereinbart. Einer dieser Termine diente dazu den Betreuern neue Ergebnisse zu präsentieren und mit diesen aktuelle Probleme zu diskutieren. In der frühen Phase des Projektes halfen diese Treffen besonders eine ausreichende Anforderungsanalyse durchzuführen. Im Rahmen der ersten Treffen wurden zudem zusätzliche Recherchen von den Mitgliedern präsentiert, welche zum Großteil die Analyse vorhandener Technologien zum Inhalt hatten.
Der andere wöchentliche Termin wurde von der Gruppe intern (ohne Anwesenheit der Betreuer) dazu genutzt, Probleme zu diskutieren und sich auf das Treffen mit den Betreuern vorzubereiten.

Aufteilung der Mitglieder Während in der Anforderungsanalyse die Komponenten des Systems erarbeitet wurden, entschied sich die Gruppe für eine Aufteilung und Spezialisierung der Mitglieder in Kleingruppen. Es wurden vier Teams gebildet, welche sich jeweils mit einem der Kernaspekte beschäftigten: Benutzerschnittstelle, Monitoring des Systems, Steuerung des Systems und Entwicklung von Benutzer-Applikationen. Zu jeder Kleingruppe gehörte ein Verantwortlicher, welcher als Ansprechpartner für Projektleiter und Betreuer diente. Nicht leitende Mitglieder wurden jeweils zwei Gruppen zugeteilt, um eine bessere Kommunikation der Kompetenzen zwischen den Gruppen zu ermöglichen.
Zu Beginn des zweiten Semesters wurde diese Aufteilung gelockert. Dies ist der Tat-

sache geschuldet, dass die vier Kernbereiche nicht unabhängig von einander sind und, durch die ungleiche Verteilung der Kompetenzen, die Entwicklung unterschiedlich schnell voran schritt. Um Wartezeiten auf andere Gruppen zu minimieren, wurden einige Mitglieder daher bei Bedarf den entsprechenden Brennpunkten zugeteilt.

Berichtswesen Das Festhalten von Beschlüssen und recherchierten Informationen war ein wichtiger Bestandteil der Organisation.
Jedes Mitglied fertigte zu seiner Präsentation während der Seminarfahrt eine kurze Ausarbeitung an. Durch diese Ausarbeitungen und den Präsentationsfolien zu den durchgeführten Recherchen war das Grundwissen ausreichend gut fixiert und zugänglich für alle Mitglieder. Zu den zwei wöchentlichen Terminen wurde jeweils ein Protokoll angefertigt, welches insbesondere getroffene Entscheidungen festhalten sollte, um nachträgliche Diskussionen zu vermeiden.
Des weiteren wurden der Dokumentation, neben jeweils einer Präsentation zum Grob- und Feinkonzept, weitere Dokumente hinzugefügt, welche Funktionen und Inhalte des Systems definieren.

Genutzte Software Um die Projektorganisation zu unterstützen, wurden eine Reihe von Programmen eingesetzt.
Das ITMC der TU Dortmund stellte für das Projekt die Software Redmine zur Verfügung. Neben einem Wiki, in dem Protokolle und weitere Dokumente organisiert werden konnten, bot diese auch die Möglichkeit zur Nutzung eines Ticketsystems, welches besonders zum Ende während der Testphase eingesetzt wurde. Das Versionskontrollsystem Subversion, ebenfalls vom ITMC zur Verfügung gestellt, wurde, wie in IT-Projekten üblich, sehr umfangreich genutzt.
Zusätzlich zum Angebot des ITMC wurde bei Treffen der Kleingruppen eine Software zur Kommunikation (Skype, Mumble) und zum gemeinsamen Editieren (VNC, Gobby) eingesetzt. Dies brachte den Vorteil, dass diese Treffen nicht immer persönlich und auch zu ungewöhnlichen Uhrzeiten stattfinden konnten, denn Terminfindung war besonders während der Vorlesungszeit ein großes Problem.

Meilensteine und Ablauf Die Arbeit am Projekt war, rückwirkend betrachtet, von sehr gemischter Produktivität gekennzeichnet. Ursache dafür war zum einen, dass

es eine wesentlich längere Zeit wie angenommen gedauert hat, die Anforderungen in einen zielgerichteten Projektplan umzusetzen. Zum Anderen hatten die sehr unterschiedlich ausgeprägten Vorkenntnisse der Mitglieder einen starken Einfluss. Einen kontinuierlichen homogenen Fortschritt in allen vier Kernbereichen zu erzielen war nur sehr schwer umzusetzen, aber Voraussetzung für die erfolgreiche und termingerechte Entwicklung des Prototyps.
Im Anschluss an die Anforderungsanalyse und mit Fertigstellung eines Grobkonzeptes, wurden folgende Meilensteine für das erste Semester festgelegt: Verfeinerung des Grobkonzeptes, Anfertigung eines Zwischenberichts und die Entwicklung eines Prototypen, welcher das Starten und Stoppen von Benutzer-Applikation ausführen konnte. Bis auf den letzten Punkt konnten die Meilensteine eingehalten werden. Ein funktionierender Prototyp wurde jedoch erst verspätet im zweiten Semester vorgeführt.
In der zweiten Hälfte des Projektes fokussierte sich die Gruppe auf die Implementierung des Prototyps, denn Anforderungen waren durch das Konzept hinreichend definiert. Meilensteine beinhalteten daher größtenteils die Fertigstellung einzelner wichtiger Funktionen. Zusätzlich waren entscheidende Meilensteine natürlich die Fertigstellung des gesamten Prototyps, der Abschluss der Test- und Evaluationsphase und die Abgabe dieses Endberichtes. Auch hier konnten nicht alle Meilensteine fristgerecht erreicht werden. Aufgrund starker Verzögerungen bei der Implementierung, starteten Test- und Evaluationsphase verspätet und kamen innerhalb des Projektes zu kurz.

Zusammengefasst lässt sich sagen, dass die Projektgruppe in Bezug auf die Organisation von IT-Projekten ihren Zweck erfüllt hat. Viele Teilnehmer hatten vor diesem Projekt nur sehr wenige oder gar keine Erfahrung mit der Projektarbeit in größeren Gruppen. Unvermeidbar wurden daher viele Fehler bei der Planung und Durchführung gemacht.
Entscheidend ist jedoch, dass, wie für eine Projektgruppe im Studium beabsichtigt, die Chance genutzt wurde, aus den begangenen Fehlern zu lernen und Aufgaben kommender Projekte besser einschätzen und planen zu können. Zudem hat jeder Teilnehmer den Umgang mit neuen Technologien und Projektwerkzeugen gelernt und damit seine Fähigkeiten erweitert.

12. Zusammenfassung

(Heng Liu) Um das Ziel der Projektgruppe zu erreichen, haben alle Projektgruppenteilnehmer im vergangenen Jahr zusammengearbeitet und beigetragen. Zuerst wurden einige vorhandene Technologien evaluiert, die für den Einsatz in unserem System in Frage kommen. Anschließend wurde das System schrittweise entworfen. Im Rahmen des erarbeiteten Feinkonzeptes konnte die konkrete Infrastruktur genau geplant werden, die anschließend in Form eines Prototyps implementiert und abschließend evaluiert wurde.

Wegen des Virtualisierungsziels wurde eine virtualisierte Ausführungsplattform für CPS-Anwendungen entworfen und implementiert. Als Virtualisierungslösungen wurde Xen ausgewählt. Auf der Virtualisierungsinfrastruktur können minimale Gastsystemen ausgeführt werden. Diese Gastsysteme sind zu migrieren und parallel auszuführen. Mit Hilfe von Techniken, die in der Virtualisierungssoftware integrierten, können Fehlertoleranzmechanismen durch den Einsatz von Remus gewährleistet werden. Durch die Zusammenarbeit des Monitoring-Systems und der LMU ist das Fehlererkennungskonzept sichergestellt. Darüber hinaus wurde Frontend mit Webserver als die grafische Oberfläche umgesetzt, um die Host-Systeme steuern und überwachen zu können. Am Ende wurde ein Distanzschutz als Beispiel-Applikation implementiert, der die Echtzeitfähigkeit unserer Plattform demonstrieren kann. Wir konnten so zeigen, dass unseres Gesamtsystem vollständig implementiert wurde und funktionsfähig ist.

Nach der Implementierung und den Tests des Systems folgte die Evaluationsphase. Viele Komponenten und kritische Punkte unseres Systems wurden evaluiert. Details finden sich im Kapitel 9.

Trotzdem haben wir bisher noch einige Probleme nicht gelöst, die im Kapitel 10 näher erläutert werden. Für die hier dargestellten Probleme sollten in Zukunft noch Lösungen gefunden werden, um die Leistungsfähigkeit des Systems zu verbessern.

12.1. Mögliche Erweiterungen

(Heng Liu)

12.1.1. N-Version Programming

In der Phase der Entwicklung von Software sind Fehler unvermeidbar. Robustheit und Korrektheit spielen jedoch eine sehr wichtige Rolle für Software. Um mögliche konzeptionelle Software-Fehler zu eliminieren, braucht man eine entsprechende Technik dafür.

N-Version Programming ist eine solche Technik, die in der Softwarefehlertoleranz angewendet werden kann. Wie der Name schon andeutet, wird eine Anzahl von $n > 1$ unabhängig entwickelten Versionen eines Programmes gestartet und ihre Ergebnisse verglichen. Auf diese Art kann ein fehlerhaftes Ergebnis ignoriert und ein korrektes Ergebnis erzeugt werden. Bei der Berechnung ist ein Voter notwendig.

N-Version Programming bietet eine effiziente Methode zur Minimierung von Ausfall- und Fehlerwahrscheinlichkeiten der Software. Die Versionenunabhängigkeit fordert allerdings spezielle Anforderungen an die Softwareentwicklung. Wegen des schwer einschätzbaren Mehraufwandes kann das *N-Version Programming* meist nur für die wichtigen Teile einer Software eingesetzt werden. Außerdem ist *N-Version Programming* problematisch, wenn mehrere gültige Lösungen vorkommen. Obwohl es also Nachteile gibt, können Fehler mit dieser Technik reduziert werden.

Im Bezug auf die Projektgruppe kann *N-Version Programming* verwendet werden, um Redundanzmechanismen zu realisieren. Bisher gibt es kein *N-Version Programming* in unserem System, aber das ist eine mögliche Erweiterung nachher.

12.1.2. Verwendung von RT-Xen

Wie im Kapitel 4.6.2 schon detailliert erklärt wurde, ist RT-Xen ein Scheduler für Xen. Der Vorteil von RT-Xen ist, dass Xen-Hypervisor echtzeitfähig gemacht wird. Dadurch könnte eine Unterstützung für virtuelle Maschinen mit Echtzeitanforderungen gewährleistet werden.

Leider kam RT-Xen nicht zum Einsatz in unserer PG. Wegen einiger Gründe haben wir auf die erste Version von RT-Xen verzichtet. Danach die zweite Version auch von uns getestet und evaluiert wurde, die leider nicht funktioniert, wie man erwartet. Trotzdem steht RT-Xen somit als eine optionale Erweiterung in der Zukunft noch zur Verfügung.

12.1.3. Abhängige Tasks

Normalerweise sind voneinander unabhängige Tasks häufig zu sehen, die in jeder beliebigen Reihenfolge ausgeführt werden können. Andererseits gibt es voneinander abhängige Tasks (siehe Abbildung 47).

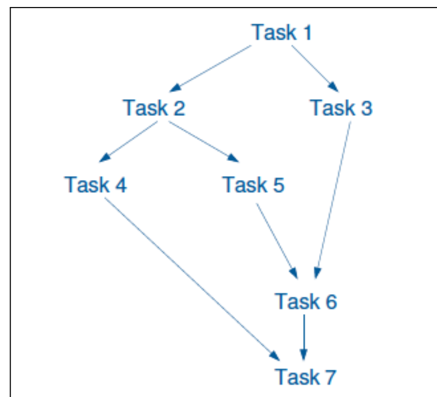


Abbildung 47: abhängige Tasks

Die Abbildung zeigt einen Taskgraphen. In diesem gerichteten Graph symbolisieren die Knoten die einzelnen Tasks; Kanten stellen die Relation der Abhängigkeiten dar (z.B. Task 2 hängt von Task 1 ab, usw.).

Zur Zeit können unabhängige Tasks mithilfe des verwendeten Schedulingverfahrens in unserem System gescheduled werden. Um abhängige Tasks zu unterstützen, muss ein anderes Schedulingmodell verwendet werden.

A. Dokumentation der LMU-Funktionen

PG 574

1.0

Erzeugt von Doxygen 1.8.4

Die Mär 4 2014 12:49:42

Inhaltsverzeichnis

1 Verzeichnis der Namensbereiche 1

1.1 Pakete 1

2 Klassen-Verzeichnis 3

2.1 Auflistung der Klassen 3

3 Datei-Verzeichnis 5

3.1 Auflistung der Dateien 5

4 Dokumentation der Namensbereiche 7

4.1 createxml-Namensbereichsreferenz 7

4.1.1 Dokumentation der Funktionen 7

4.1.1.1 create_heap_dump 7

4.1.1.2 createreply 7

4.2 host-Namensbereichsreferenz 7

4.3 scanadapter-Namensbereichsreferenz 8

4.3.1 Dokumentation der Funktionen 8

4.3.1.1 addnewhost 8

4.3.1.2 loadobject 8

4.3.1.3 move_task_xml 9

4.3.1.4 saveobject 9

4.3.1.5 start_task 9

4.3.1.6 startup 9

4.3.1.7 stop_all_hosts 9

4.3.1.8 stop_host 9

4.3.1.9 stop_task 9

4.3.2 Variablen-Dokumentation 10

4.3.2.1 BACKUP 10

4.3.2.2 M 10

4.3.2.3 MASTER 10

4.3.2.4 MB 10

4.3.2.5 MBN 10

4.3.2.6 NEUEZEILE 10
4.3.2.7 NODE 10
4.3.2.8 PAUSED 10
4.3.2.9 RUNNING 10
4.3.2.10 TRENNZEICHEN 10
4.4 scheduler-Namensbereichsreferenz 10
4.4.1 Dokumentation der Funktionen 13
4.4.1.1 addfault 13
4.4.1.2 addmessage 13
4.4.1.3 all_list_print 13
4.4.1.4 checkeverything 13
4.4.1.5 checkstate 14
4.4.1.6 copy_heap 14
4.4.1.7 coreload_refresh 14
4.4.1.8 coreload_update 14
4.4.1.9 delnodevm 14
4.4.1.10 delvm 14
4.4.1.11 delvmfromhost 15
4.4.1.12 find_place 15
4.4.1.13 found_check 15
4.4.1.14 getcorenumbertovm 15
4.4.1.15 getglobalvmlist 16
4.4.1.16 gethost 17
4.4.1.17 gethosttovm 17
4.4.1.18 getmasterhosttovm 17
4.4.1.19 getvmnumberonhost 17
4.4.1.20 heap_format 18
4.4.1.21 heap_quadrupel 18
4.4.1.22 heap_save 18
4.4.1.23 heap_tripel 18
4.4.1.24 heap_update 18
4.4.1.25 hostdown 19
4.4.1.26 isthereabackup 19
4.4.1.27 isthereamaster 19
4.4.1.28 isthereanode 19
4.4.1.29 isvmonhost 19
4.4.1.30 isvmwithstateonhost 20
4.4.1.31 list_print 20
4.4.1.32 load_check 20
4.4.1.33 migratevm 20

4.4.1.34 mkr 21
4.4.1.35 moveable_check 22
4.4.1.36 new_vm_redundancy_check 22
4.4.1.37 place_for_newvm 22
4.4.1.38 preparehostdown 23
4.4.1.39 preparemigratevm 24
4.4.1.40 preparevmchangedstate 24
4.4.1.41 prepearevmtorestart 24
4.4.1.42 print_heap 24
4.4.1.43 printd 24
4.4.1.44 redundancy_check 24
4.4.1.45 removehost 25
4.4.1.46 reschedule 25
4.4.1.47 schedulenewvm 25
4.4.1.48 shutdown 25
4.4.1.49 sort_heap_quadrupel 25
4.4.1.50 startnewbackupvm 26
4.4.1.51 startnewmastervm 26
4.4.1.52 startnewnodevm 26
4.4.1.53 stopvm 26
4.4.1.54 test_addfault 26
4.4.1.55 test_addfaultnumber 27
4.4.1.56 transmitstartbackup 27
4.4.1.57 transmitstartmaster 27
4.4.1.58 transmitstopvm 27
4.4.1.59 vm_migrate 27
4.4.1.60 vm_move 28
4.4.1.61 vm_to_move 28
4.4.1.62 vmdown 28
4.4.2 Variablen-Dokumentation 29
4.4.2.1 all_list 29
4.4.2.2 BACKUP 29
4.4.2.3 found_places_list 29
4.4.2.4 globlogging 29
4.4.2.5 globtesting 29
4.4.2.6 M 29
4.4.2.7 MASTER 29
4.4.2.8 maxload_of_core 29
4.4.2.9 MB 29
4.4.2.10 MBN 29

4.4.2.11 migrate_list 29
4.4.2.12 NEUEZEILE 29
4.4.2.13 NODE 29
4.4.2.14 temp_migrate_list 29
4.4.2.15 TRENNZEICHEN 29
4.5 tempreply-Namensbereichsreferenz 29
4.6 testen-Namensbereichsreferenz 30
4.6.1 Dokumentation der Funktionen 31
4.6.1.1 action_xml_error_test 31
4.6.1.2 belastung_test 31
4.6.1.3 check_reply 31
4.6.1.4 configuration_test 31
4.6.1.5 create_action_xml 31
4.6.1.6 create_action_xml_1 32
4.6.1.7 create_action_xml_2 32
4.6.1.8 create_directory 32
4.6.1.9 create_four_hosts_xml 32
4.6.1.10 create_one_host_xml 32
4.6.1.11 create_random_hosts_xml 32
4.6.1.12 create_task_xml 32
4.6.1.13 create_task_xml_0 33
4.6.1.14 create_task_xml_1 33
4.6.1.15 create_task_xml_2 33
4.6.1.16 create_three_hosts_xml 33
4.6.1.17 create_two_hosts_xml 33
4.6.1.18 error_check 34
4.6.1.19 failure_check 35
4.6.1.20 get_reply 35
4.6.1.21 get_reply_message 35
4.6.1.22 message_list 35
4.6.1.23 move_actionxml_and_replyxml 36
4.6.1.24 move_to_directory 36
4.6.1.25 start2times_task_test 36
4.6.1.26 start_pause_task_test 36
4.6.1.27 start_stop_task_test 36
4.6.1.28 start_task_test 36
4.6.1.29 task_xml_error_test 36
4.6.1.30 wcet_longer_period_test 37
4.6.2 Variablen-Dokumentation 37
4.6.2.1 pfad 37

4.7 transmitcommand-Namensbereichsreferenz . . . 37
4.7.1 Dokumentation der Funktionen . . . 38
4.7.1.1 close_socket . . . 38
4.7.1.2 connect_to_socket . . . 38
4.7.1.3 dom_create . . . 38
4.7.1.4 dom_create_cfg . . . 38
4.7.1.5 dom_destroy . . . 38
4.7.1.6 dom_migrate . . . 39
4.7.1.7 dom_notify_incoming_migration . . . 39
4.7.1.8 dom_notify_incoming_remus . . . 39
4.7.1.9 dom_notify_intended_failover . . . 39
4.7.1.10 dom_remus . . . 39
4.7.1.11 dom_set_period_slice . . . 40
4.7.1.12 dom_vcpu_pin . . . 40
4.7.1.13 host_check_sanity . . . 40
4.7.1.14 host_get_state . . . 40
4.7.1.15 transmit_message . . . 40
4.7.1.16 transmit_state . . . 41
4.7.2 Variablen-Dokumentation . . . 41
4.7.2.1 extendedSocketTimeout . . . 41
4.7.2.2 mySocket . . . 41
4.7.2.3 parser . . . 41
4.7.2.4 socketTimeout . . . 41
4.8 verzeichnisscanner-Namensbereichsreferenz . . . 41
4.8.1 Dokumentation der Funktionen . . . 42
4.8.1.1 check_xml_file . . . 42
4.8.1.2 checkrunningfiles . . . 42
4.8.1.3 execCommand . . . 42
4.8.1.4 getIP . . . 42
4.8.1.5 getsysteminfo . . . 43
4.8.1.6 initPipe . . . 43
4.8.1.7 is_host_in_xml . . . 43
4.8.1.8 ishostinheap . . . 43
4.8.1.9 load_hosts_xml_file . . . 43
4.8.1.10 load_hosts_xml_file_for_check . . . 43
4.8.1.11 load_taskxml . . . 43
4.8.1.12 load_xml_file . . . 44
4.8.1.13 readpipe . . . 44
4.8.1.14 runme . . . 44
4.8.1.15 start_rep_lmu . . . 44

4.8.1.16 start_webserver . . . 44
4.8.2 Variablen-Dokumentation . . . 44
4.8.2.1 BACKUP . . . 44
4.8.2.2 globimgpath . . . 44
4.8.2.3 lastchange . . . 44
4.8.2.4 lokalerpfad . . . 44
4.8.2.5 M . . . 44
4.8.2.6 MASTER . . . 44
4.8.2.7 MB . . . 45
4.8.2.8 MBN . . . 45
4.8.2.9 NEUEZEILE . . . 45
4.8.2.10 NODE . . . 45
4.8.2.11 pipe_name . . . 45
4.8.2.12 pipein . . . 45
4.8.2.13 testing . . . 45
4.9 vm-Namensbereichsreferenz . . . 45

5 Klassen-Dokumentation 47

5.1 host.Host Klassenreferenz . . . 47
5.1.1 Ausführliche Beschreibung . . . 48
5.1.2 Beschreibung der Konstruktoren und Destruktoren . . . 48
5.1.2.1 __init__ . . . 48
5.1.3 Dokumentation der Elementfunktionen . . . 48
5.1.3.1 __lt__ . . . 48
5.1.3.2 addcoreload . . . 48
5.1.3.3 addvm . . . 48
5.1.3.4 getallvms . . . 49
5.1.3.5 getcoreload . . . 49
5.1.3.6 getip . . . 49
5.1.3.7 getmaxcoreload . . . 49
5.1.3.8 getmaxload . . . 49
5.1.3.9 getmincoreload . . . 49
5.1.3.10 getminloadofcore . . . 49
5.1.3.11 getname . . . 49
5.1.3.12 getnumberofcores . . . 49
5.1.3.13 getposmaxcoreload . . . 50
5.1.3.14 getposmincoreload . . . 50
5.1.3.15 getvm . . . 50
5.1.3.16 getvmlist . . . 50
5.1.3.17 getvmlistcore . . . 50

5.1.3.18 getvmlistofcore . . . 50
5.1.3.19 getvmlistsort . . . 50
5.1.3.20 removecoreload . . . 51
5.1.3.21 removevm . . . 51
5.1.3.22 removevmlist . . . 51
5.1.3.23 setcoreload . . . 51
5.1.3.24 setvmstate . . . 51
5.1.4 Dokumentation der Datenelemente . . . 51
5.1.4.1 coreload . . . 51
5.1.4.2 ip . . . 51
5.1.4.3 maxload . . . 51
5.1.4.4 name . . . 51
5.1.4.5 numberofcores . . . 51
5.1.4.6 vmlist . . . 51
5.2 tempreply.Tempreply Klassenreferenz . . . 52
5.2.1 Ausführliche Beschreibung . . . 52
5.2.2 Beschreibung der Konstruktoren und Destruktoren . . . 52
5.2.2.1 __init__ . . . 52
5.2.3 Dokumentation der Elementfunktionen . . . 52
5.2.3.1 getfault . . . 52
5.2.3.2 getmessage . . . 52
5.2.3.3 setfault . . . 52
5.2.3.4 setmessage . . . 52
5.2.4 Dokumentation der Datenelemente . . . 53
5.2.4.1 fault . . . 53
5.2.4.2 message . . . 53
5.3 vm.VM Klassenreferenz . . . 53
5.3.1 Ausführliche Beschreibung . . . 54
5.3.2 Beschreibung der Konstruktoren und Destruktoren . . . 54
5.3.2.1 __init__ . . . 54
5.3.3 Dokumentation der Elementfunktionen . . . 54
5.3.3.1 getcreatedon . . . 54
5.3.3.2 getid . . . 55
5.3.3.3 getimgpath . . . 55
5.3.3.4 getip . . . 55
5.3.3.5 getload . . . 55
5.3.3.6 getmac . . . 55
5.3.3.7 getos . . . 55
5.3.3.8 getperiod . . . 55
5.3.3.9 getredundancy . . . 55

5.3.3.10 getstate 55
5.3.3.11 gettaskname 55
5.3.3.12 getwcet 55
5.3.3.13 setcreatedon 55
5.3.3.14 setimgpath 56
5.3.3.15 setip 56
5.3.3.16 setmac 56
5.3.3.17 setos 56
5.3.3.18 setstate 56
5.3.4 Dokumentation der Datenelemente 56
5.3.4.1 createdon 56
5.3.4.2 ident 56
5.3.4.3 imgpath 56
5.3.4.4 ip 56
5.3.4.5 mac 56
5.3.4.6 operating 56
5.3.4.7 period 56
5.3.4.8 redundancy 56
5.3.4.9 state 56
5.3.4.10 taskname 56
5.3.4.11 wcet 56

6 Datei-Dokumentation 57
6.1 createxml.py-Dateireferenz 57
6.2 host.py-Dateireferenz 57
6.3 scanadapter.py-Dateireferenz 58
6.4 scheduler.py-Dateireferenz 58
6.5 tempreply.py-Dateireferenz 62
6.6 testen.py-Dateireferenz 62
6.7 transmitcommand.py-Dateireferenz 63
6.8 verzeichnisscanner.py-Dateireferenz 64
6.9 vm.py-Dateireferenz 66

Index 67

Kapitel 1

Verzeichnis der Namensbereiche

1.1 Pakete

Hier folgen die Pakete mit einer Kurzbeschreibung (wenn verfügbar):

createxml 7
host 7
scanadapter 8
scheduler 10
tempreply 29
testen 30
transmitcommand 37
verzeichnisscanner 41
vm 45

Kapitel 2

Klassen-Verzeichnis

2.1 Auflistung der Klassen

Hier folgt die Aufzählung aller Klassen, Strukturen, Varianten und Schnittstellen mit einer Kurzbeschreibung:

host.Host
Host Objekte entahlten alle Informationen ueber die Hosts und eine Liste von VMs 47
tempreply.Tempreply
Wird zum generieren der reply.xml benoetigt 52
vm.VM
VM Objekte enthalten alle Informationen ueber die VM 53

Kapitel 3

Datei-Verzeichnis

3.1 Auflistung der Dateien

Hier folgt die Aufzählung aller Dateien mit einer Kurzbeschreibung:

createxml.py 57
host.py 57
scanadapter.py 58
scheduler.py 58
tempreply.py 62
testen.py 62
transmitcommand.py 63
verzeichnisscanner.py 64
vm.py 66

Kapitel 4

Dokumentation der Namensbereiche

4.1 createxml-Namensbereichsreferenz

Funktionen

- def create_heap_dump

  Erzeugt einen Heapdump.
- def createreply

  Erstellen einer reply.xml.

4.1.1 Dokumentation der Funktionen

4.1.1.1 def createxml.create_heap_dump (*filename* = 'heap_dump.csv')

Erzeugt einen Heapdump.

Dabei werden nur die Rechner ausgegeben auf denen sich mind. eine VM befindet

Parameter

filename	Dateiname vom heapdump.

4.1.1.2 def createxml.createreply (*message*, *timestamp*)

Erstellen einer reply.xml.

Parameter

message	String als Nachricht
timestamp	Zeitpunkt der action.xml

4.2 host-Namensbereichsreferenz

Klassen

- class Host

  Host Objekte entahlten alle Informationen ueber die Hosts und eine Liste von VMs.

4.3 scanadapter-Namensbereichsreferenz

Funktionen

- def stop_host
  Funktion sucht zu das passende Host Objekt und uebergibt es dem scheduler.
- def stop_all_hosts
  Dummy Funktion, macht nichts.
- def move_task_xml
- def start_task
  Staret einen Task und verschiebt die task.xml nach refused oder running.
- def stop_task
  Beendet einen Task.
- def saveobject
  Speichert ein Objekt.
- def loadobject
  Laedt ein Objekt.
- def addnewhost
  Fuegt dem Heap einen Host hinzu.
- def startup
  Initialisieren vom Heap, Replay und logging.

Variablen

- string NEUEZEILE = '\n '
  Trennzeichen von Zeilen fuer die reply.xml.
- int MASTER = 0
  State von dem Task.
- int BACKUP = 1
- int NODE = 2
- int MBN = 3
  Redundanz mit der die VM gestartet werden soll.
- int MB = 2
- int M = 1
- int RUNNING = 0
- int PAUSED = 1
- string TRENNZEICHEN = ';'

4.3.1 Dokumentation der Funktionen

4.3.1.1 def scanadapter.addnewhost (*host*)

Fuegt dem Heap einen Host hinzu.

Parameter

host	Hostobjekt, der dem Heap hinzugefuegt werden soll

4.3.1.2 def scanadapter.loadobject (*filename*)

Laedt ein Objekt.

Parameter

filename	Dateiname der Datei, die geladen werden soll

Rückgabe

gibt ein Objekt zurueck

4.3.1.3 def scanadapter.move_task_xml (*taskname*)

4.3.1.4 def scanadapter.saveobject (*obj, filename*)

Speichert ein Objekt.

Parameter

obj	Objekt das gespeichert werden soll
filename	Dateiname, in dem das Objekt gespeichert werden soll.

4.3.1.5 def scanadapter.start_task (*vmtostart, timestamp, imgpath, testing*)

Staret einen Task und verschiebt die task.xml nach refused oder running.

Parameter

vmtostart	VM Objekt, der zu startenden VM.
timestamp	Zeitpunkt, der in die reply.xml eingefuegt wird
testing	True falls im Debug Modus gestartet wurde. In diesem Fall werden keine Kommandos an die Rechner uebrtragen
imagpath	Pfad zu dem Image

4.3.1.6 def scanadapter.startup ( )

Initialisieren vom Heap, Replay und logging.

4.3.1.7 def scanadapter.stop_all_hosts ( )

Dummy Funktion, macht nichts.

4.3.1.8 def scanadapter.stop_host (*target, testing*)

Funktion sucht zu das passende Host Objekt und uebergibt es dem scheduler.

Parameter

target	Name (als String) des Hosts, der beendet werden soll.
testing	True falls im Debug Modus gestartet wurde. In diesem Fall werden keine Kommandos an die Rechner uebrtragen

4.3.1.9 def scanadapter.stop_task (*vmtostop, timestamp, testing*)

Beendet einen Task.

Parameter

vmtostart	VM Objekt, der zu startenden VM.
timestamp	Zeitpunkt, der in die reply.xml eingefuegt wird
testing	True falls im Debug Modus gestartet wurde. In diesem Fall werden keine Kommandos an die Rechner uebrtragen
vmtostop	Vm Objekt von der zu stppenden VM

4.3.2 Variablen-Dokumentation

4.3.2.1 int scanadapter.BACKUP = 1

4.3.2.2 int scanadapter.M = 1

4.3.2.3 int scanadapter.MASTER = 0

State von dem Task.

4.3.2.4 int scanadapter.MB = 2

4.3.2.5 int scanadapter.MBN = 3

Redundanz mit der die VM gestartet werden soll.

4.3.2.6 string scanadapter.NEUEZEILE = '\n '

Trennzeichen von Zeilen fuer die reply.xml.

4.3.2.7 int scanadapter.NODE = 2

4.3.2.8 int scanadapter.PAUSED = 1

4.3.2.9 int scanadapter.RUNNING = 0

4.3.2.10 string scanadapter.TRENNZEICHEN = ';'

4.4 scheduler-Namensbereichsreferenz

Funktionen

- def printd

  Erzeugt ein feingranulares Logging.
- def test_addfaultnumber

  Im Testmodus werden anstatt Strings Ints gespeichert.
- def test_addfault

  Falls ein Fehler aufgetreten ist, wird dieser hier gespeichert.
- def addmessage

  Die Nachricht wird nach dem Starten als reply.xml ausgegeben.
- def addfault

  Falls ein Fehler aufgetreten ist, wird dieser hier gespeichert.
- def gethosttovm

  Sucht zu einem VM Objekt das passendene Hostobjekt aus dem Heap.

- def gethost
  Sucht zu einem String den Host.
- def removehost
  Loescht ein Host aus dem heap.
- def getglobalvmlist
  Gibt eine Liste mit allen VM Objekten auf allen Hosts zurueck.
- def isthereamaster
  Gibt true zurueck, falls eine VM mit dem gleichen Namen existiert.
- def isthereabackup
  Gibt true zurueck, falls eine VM mit dem gleichen Namen existiert.
- def isthereanode
  Gibt true zurueck, falls eine VM mit dem gleichen Namen existiert.
- def isvmwithstateonhost
  Gibt zurueck, ob sich eine VM mit dem State und dem Namen auf dem Host befindet.
- def isvmonhost
  Gibt zurueck, ob sich eine VM mit dem Namen auf dem Host befindet.
- def getmasterhosttovm
  Es wird der Host zurueckgegeben, auf dem sich die MasterVM zu der uebergegeben VM befindet.
- def getvmnumberonhost
  Gibt den Index einer VM auf einem Host zurueck.
- def getcorenumbertovm
  Gibt den Kern zurueck, auf dem sich die VM befindet.
- def delvmfromhost
  Loescht eine VM aus der internen Datenstruktur.
- def delvm
  Loescht eine VM vom Host und aus der internen Datenstruktur.
- def delnodevm
  Loescht ein VM Objekt mit dem State NODE.
- def checkstate
  In dieser Funktion werden fehlende Redundanzen hergestellt.
- def migratevm
  Mirgiert eine VM.
- def preparemigratevm
  Diese Funktion bereitet das mirgieren von einer VM vor.
- def startnewmastervm
  Bereitet das Starten einer neuen Mastervm vor.
- def transmitstartmaster
  Startet eine VM auf einem Host.
- def startnewnodevm
  Node Objekt wird der Datenstruktur hinzugefuegt.
- def startnewbackupvm
  Bereitet das Starten einer Backup VM vor.
- def transmitstartbackup
  Uebertreagt das Kommando zum Starten einer Backup VM.
- def schedulenewvm
  Einsprungprunkt zum Starten einer neuen VM.
- def transmitstopvm
  Uebertraegt das Kommando zum Beenden einer VM.
- def stopvm
  Beendet eine VM.
- def preparehostdown

Wenn ein Host nicht mehr erreichbar ist, wird hier das Hostobjekt gesucht und weitergeleitet.

- def prepearevmtorestart

  Bereitet das Neustarten einer VM vor.
- def preparevmchangedstate

  Falls eine VM ihren Status wechselt, sucht diese Funktion die Objekte und beendet die Master VM.
- def vmdown

  Diese Funktion startet eine abgestuerzte VM neu.
- def hostdown

  Diese Funktion startet alle VMs eines Hosts neu.
- def shutdown

  Diese Funktion bereitet einen Host zum Herunterfahren vor.
- def checkeverything

  Es wird regelmaessig ueberprueft, ob der Heap mit der Realitaet uebereinstimmt.
- def print_heap
- def copy_heap

  Gibt eine Kopie des aktuellen Heaps zurueck.
- def heap_tripel

  Gibt eine, um Hostnamen und Coreindex erweiterte, Liste des aktuellen Heaps zurueck.
- def heap_quadrupel

  Gibt eine Liste des aktuellen Heaps zurueck.
- def sort_heap_quadrupel

  Gibt eine nach Core-Auslastung sortierte Heap-Liste zurueck.
- def all_list_print
- def list_print
- def mkr

  Das MaKeRoom wird vor dem Starten einer neuen VM ausgefuehrt.
- def find_place

  Wird von mkr aufgerufen.
- def place_for_newvm

  Prueft auf einem bestimmten Core, ob die neue VM draufpasst.
- def coreload_refresh
- def vm_to_move

  Sucht eine Stelle , wohin die angegebene VM verschoben werden kann.
- def vm_move

  In der angegebenen Heap-Liste wird die VM vom Quell-Host/Core zum Ziel-Host/Core verschoben.
- def new_vm_redundancy_check

  Prueft ob auf diesem Host bereits Platz fuer die neue VM reserviert/freigemacht wurde.
- def found_check

  Prueft, ob Core bereits reserviert wurde.
- def redundancy_check

  Prueft, ob Redundanz der zu verschiebenden VM bereits auf diesem Host ist.
- def load_check

  Prueft, ob die neue VM auf Core passt.
- def coreload_update

  Berechnen die Coreauslastung neu.
- def vm_migrate

  Migriert Schritt fuer Schritt die Migrate-Liste ab.
- def heap_format

  Bringt eine lange-erweiterte Heap-Liste in die normale Heapform.
- def heap_update

  Erhaelt eine erweiterte-lange-Heapliste und speichert sie als Heap.

- def heap_save

    Speichert eine Heap-Liste als Heap ab.
- def moveable_check

    Prueft, ob diese VM migrierbar ist.
- def reschedule

    Rebalanciert den Heap, in dem die Core-Auslastungen ausgeglichen werden.

Variablen

- string NEUEZEILE = '\n '

    Trennzeichen von Zeilen fuer die reply.xml.
- string TRENNZEICHEN = ';'

    Trennzeichen fuer die verschiedenen Fehlercodes.
- int MASTER = 0

    State von dem Task.
- int BACKUP = 1
- int NODE = 2
- int MBN = 3

    Redundanz mit der die VM gestartet werden soll.
- int MB = 2
- int M = 1
- globtesting = False
- globlogging = True
- int maxload_of_core = 1
- tuple all_list = list()
- tuple migrate_list = list()
- tuple found_places_list = list()
- tuple temp_migrate_list = list()

4.4.1 Dokumentation der Funktionen

4.4.1.1 def scheduler.addfault (*x*)

Falls ein Fehler aufgetreten ist, wird dieser hier gespeichert.

Parameter

x	Es wird ein bool erwartet.

4.4.1.2 def scheduler.addmessage (*str*)

Die Nachricht wird nach dem Starten als reply.xml ausgegeben.

Parameter

str	Nachricht, die in der reply.xml gespeichert werden soll.

4.4.1.3 def scheduler.all_list_print (*liste*)

4.4.1.4 def scheduler.checkeverything (*hostlist, heap, testing*)

Es wird regelmaessig ueberprueft, ob der Heap mit der Realitaet uebereinstimmt.

Parameter

heap	Heap, mit dem gearbeitet werden soll.
hostlist	Liste von Tripels, die aus Hostobjekten, VM Objekten und States von VMs bestehen
testing	True falls im Debug Modus gestartet wurde. In diesem Fall werden keine Kommandos an die Rechner uebrtragen

4.4.1.5 def scheduler.checkstate ( )

In dieser Funktion werden fehlende Redundanzen hergestellt.

Die Funktion wird nach jedem Hinzufuegen eines Hosts aufgerufen.

4.4.1.6 def scheduler.copy_heap ( )

Gibt eine Kopie des aktuellen Heaps zurueck.

Rückgabe

heap_list [(host_objekt, minimalste_auslastung_des_hosts), ...].

4.4.1.7 def scheduler.coreload_refresh (*copy_all_list, i*)

4.4.1.8 def scheduler.coreload_update (*element*)

Berechnen die Coreauslastung neu.

(Wichtig nach dem Migrieren)

Parameter

element	aus der grossen Heap-Liste (Hostname, Hostobjekt, Coreindex, Coreauslastung, VM-Liste)

Rückgabe

Element (Hostname, Hostobjekt, Coreindex, neuberechneter_Coreauslastung, VM-Liste)

4.4.1.9 def scheduler.delnodevm (*vmobjekt, heap*)

Loescht ein VM Objekt mit dem State NODE.

Parameter

vmobjekt	Vm Objekt das entfernt werden soll. Es reicht der VM Name.
heap	Heap mit dem gearbeitet werden soll.

4.4.1.10 def scheduler.delvm (*lokvm, heap*)

Loescht eine VM vom Host und aus der internen Datenstruktur.

Parameter

lokvm	Vm Objekt, das geloescht werden soll.
heap	Heap mit dem gearbeitet werden soll.

4.4.1.11 def scheduler.delvmfromhost (*lokvm*, *lokhost*)

Loescht eine VM aus der internen Datenstruktur.

Parameter

lokvm	VM Objekt, das aus der internen Datenstruktur geloescht werden soll.
lokhost	Host Objekt, von dem die VM geloscht werden soll.

4.4.1.12 def scheduler.find_place (*new_load*)

Wird von mkr aufgerufen.

Geht die sortierte Heap-Liste durch und prueft, ob auf einem Core genug Platz fuer die gegebene VM-Groesse vorhanden ist.

Parameter

new_load	Die Auslastung, die die neue VM verursachen wird.

Rückgabe

True, falls genug Platz auf einem Core vorhanden ist. False, falls nirgends genug Platz vorhanden ist.

4.4.1.13 def scheduler.found_check (*hostname*, *coreindex*)

Prueft, ob Core bereits reserviert wurde.

(Wichtig beim Migrieren der anderen VMs, die zwar auf einen reservierten Host duerfen, aber nicht auf einen reservierten Core).

Parameter

hostname	Name des Hosts
coreindex	Index des Cores in diesem Host

Rückgabe

True, Core nicht reserviert. False, Core bereits reserviert.

4.4.1.14 def scheduler.getcorenumbertovm (*lokvm*, *lokhost*)

Gibt den Kern zurueck, auf dem sich die VM befindet.

Parameter

lokvm	Vm Objekt, zu dem der Kern zurueckgegben werden soll.
lokhost	Host, auf dem sich die VM befindet.

Rückgabe

core Kern, auf dem sich die VM befindet.

4.4.1.15 def scheduler.getglobalvmlist (*heap*)

Gibt eine Liste mit allen VM Objekten auf allen Hosts zurueck.

Parameter

heap	Heap, von dem alle VMs zurueckgegeben werden sollen.

Rückgabe

golbalvmlist Liste mit allen VMs auf allen Hosts.

4.4.1.16 def scheduler.gethost (*host, heap*)

Sucht zu einem String den Host.

Parameter

host	es wird der Name des Hosts als String erwartet.
heap	Der heap, in dem der Host gefunden werden soll.

Rückgabe

Es wird bei erfolgreicher Suche ein Hostobjekt zurueckgegeben.

4.4.1.17 def scheduler.gethosttovm (*vmtofind, heap*)

Sucht zu einem VM Objekt das passendene Hostobjekt aus dem Heap.

Parameter

vmtofind	VM Objekt mit mindestens Namen und State.
heap	Der Heap wird nicht veraendert.

Rückgabe

bei erfolgreicher Suche wird ein Hostobjekt zurueckgegeben, falls sie erfolglos war geschieht nichts.

4.4.1.18 def scheduler.getmasterhosttovm (*vmtofind, heap*)

Es wird der Host zurueckgegeben, auf dem sich die MasterVM zu der uebergegeben VM befindet.

Parameter

vmtofind	Vm Objekt, zu der der Masterhost zurueckgegeben werden soll.
heap	Heap mit dem gearbeitet werden soll.

Rückgabe

Es wird das Hostobjekt zurueckgegeben, auf dem sich die MasterVM zu der uebergegebene VM befindet. Falls keins gefunden wird, wird nichts zurueck gegegben.

4.4.1.19 def scheduler.getvmnumberonhost (*svm, akthost*)

Gibt den Index einer VM auf einem Host zurueck.

Falls die VM nicht gefunden wurde, wird -1 zurueckgegeben

Parameter

svm	VM, dessen Index auf der Hostliste zurueckgegeben werden soll.
akthost	Host, mit dem gearbeitet werden soll.

4.4.1.20 def scheduler.heap_format (*all_list*)

Bringt eine lange-erweiterte Heap-Liste in die normale Heapform.

Parameter

all_list	grosse Heapliste der Form [(hostname, hostobjekt, coreindex, corauslastung, [(vmobjekt, groesse),...]), ...]

Rückgabe

Normale Heap-Form der uebergebenen Liste: [(minimalste-coreauslastung, hostobj), ...] (Immernoch ein Listenelement pro Core)

4.4.1.21 def scheduler.heap_quadrupel ( )

Gibt eine Liste des aktuellen Heaps zurueck.

Rückgabe

heap-liste nach Hosts sortiert: [(hostname, hostobjekt, coreindex, corauslastung, aufsteigend_sortierte_-vmliste-des-cores), ...].

4.4.1.22 def scheduler.heap_save (*heap_list*)

Speichert eine Heap-Liste als Heap ab.

Parameter

heap_list	Heap in Listen-Form.

4.4.1.23 def scheduler.heap_tripel ( )

Gibt eine, um Hostnamen und Coreindex erweiterte, Liste des aktuellen Heaps zurueck.

Rückgabe

Heap-Liste nach Hosts sortiert: [(hostname, hostobjekt, coreindex, core_auslastung),...].

4.4.1.24 def scheduler.heap_update (*all_list*)

Erhaelt eine erweiterte-lange-Heapliste und speichert sie als Heap.

Parameter

all_list	grosse Heapliste der Form [(hostname, hostobjekt, coreindex, corauslastung, [(vmobjekt, groesse),...]), ...]

4.4.1.25 def scheduler.hostdown (*hostobj*, *testing*)

Diese Funktion startet alle VMs eines Hosts neu.

Parameter

hostobj	Host Objekt, von dem die VMs neugetartet werden sollen.
testing	True, falls im Debug Modus gestartet wurde. In diesem Fall werden keine Kommandos an die Rechner uebertragen.

4.4.1.26 def scheduler.isthereabackup (*backupvm*, *heap*)

Gibt true zurueck, falls eine VM mit dem gleichen Namen existiert.

Parameter

backupvm	Es wird ein VM Objekt mit mindestens Name und state erwartet
heap	Heap mit dem gearbeitet werden soll.

Rückgabe

True falls ein Backup vorhanden ist, sonst False.

4.4.1.27 def scheduler.isthereamaster (*mastervm*, *heap*)

Gibt true zurueck, falls eine VM mit dem gleichen Namen existiert.

Parameter

mastervm	Es wird ein VM Objekt mit mindestens Name und state erwartet.
heap	Heap mit dem gearbeitet werden soll.

Rückgabe

True falls ein Master vorhanden ist, sonst False.

4.4.1.28 def scheduler.isthereanode (*nodevm*, *heap*)

Gibt true zurueck, falls eine VM mit dem gleichen Namen existiert.

Parameter

nodevm	Es wird ein VM Objekt mit mindestens Name und state erwartet.
heap	Heap mit dem gearbeitet werden soll.

Rückgabe

True falls eine Node vorhanden ist, sonst False

4.4.1.29 def scheduler.isvmonhost (*lokvm*, *host*)

Gibt zurueck, ob sich eine VM mit dem Namen auf dem Host befindet.

Parameter

lokvm	VM Objekt, das mindestens einen Namen besitzt.
host	Hostobjekt, auf dem gesucht werden soll.

Rückgabe

True, falls eine VM mit dem Namen auf dem Host ist, sonst False.

4.4.1.30 def scheduler.isvmwithstateonhost (*lokvm, host*)

Gibt zurueck, ob sich eine VM mit dem State und dem Namen auf dem Host befindet.

Parameter

lokvm	VM Objekt, das mindestens den Namen und den State enthaelt.
host	Hostobjekt, auf dem gesucht werden soll.

Rückgabe

True falls eine VM mit dem State und Namen auf dem Host ist, sonst False.

4.4.1.31 def scheduler.list_print (*liste*)

4.4.1.32 def scheduler.load_check (*coreload, vm_load*)

Prueft, ob die neue VM auf Core passt.

Parameter

coreload	Coreauslastung
vm_load	Auslastung, die die VM verursacht.

Rückgabe

True, VM passt auf den Core. False, VM passt nicht auf den Core.

4.4.1.33 def scheduler.migratevm (*vmob, qhost, qcore, zhost, zcore*)

Mirgiert eine VM.

Es werden keine Aenderungen an der Datenstruktur vorgenommen.

Parameter

vmob	Die VM, die migriert werden soll
qhost	Quell Hostobjekt.
qcore	Quell Kern (int).
zhost	Ziel Hostobjekt.
zcore	Ziel Kern (int).

Rückgabe

Gibt True zurueck, wenn das Migrieren erfolgreich war. Im Fehlerfall wird False zurueckgegeben.

4.4.1.34 def scheduler.mkr (*new_vm*)

Das MaKeRoom wird vor dem Starten einer neuen VM ausgefuehrt.

Bei Bedarf wird Platz fuer die neue VM und ihre Redundanzen geschaffen.

Parameter

new_vm	Neues Objekt der VM, die gestartet werden soll.

Rückgabe

True, wenn genug Platz fuer die VM besteht/geschaffen wurde. False, wenn trotz Verschieben nicht genug Platz fuer die neue VM und Ihre Redundanzen besteht.

4.4.1.35 def scheduler.moveable_check (*vmobj*)

Prueft, ob diese VM migrierbar ist.

(Nur Stand-alone-Masters und Nodes)

Parameter

vmobj	Objekt der VM, die migriert werden soll.

Rückgabe

True, VM migrierbar. False, VM nicht migrierbar.

4.4.1.36 def scheduler.new_vm_redundancy_check (*hostname*)

Prueft ob auf diesem Host bereits Platz fuer die neue VM reserviert/freigemacht wurde.

(Sonst darf die Redundanz der neuen VM hier nicht drauf).

Parameter

hostname	Name des Hosts

Rückgabe

True, Host ist frei. False, Host ist bereits reserviert.

4.4.1.37 def scheduler.place_for_newvm (*copy_all_list, q, m, new_load*)

Prueft auf einem bestimmten Core, ob die neue VM draufpasst.

Falls nicht, werden dort solange die VMs verschoben, bis genug Platz ist oder nichts mehr zu verschieben ist.

Parameter

copy_all_list	Heap-Liste: [(hostname, hostobjekt, coreindex, corauslastung, aufsteigend_sortierte_-vmliste-des-cores), ...]
q	Index in der Heap-Liste, wo die neue VM drauf soll.
m	VM-Index in der VM-Liste vom Quell-Element. Zeigt, welche VM verschoben werden soll. Zu Beginn ist index auf 0 gesetzt.
new_load	Auslastung, die die neue VM verursacht.

Rückgabe

True, wenn genug Platz fuer die neue VM vorhanden bzw geschaffen wurde. False, falls nicht genug Platz fuer die neue VM vorhanden ist.

4.4.1.38 def scheduler.preparehostdown (*hostname*, *testing*)

Wenn ein Host nicht mehr erreichbar ist, wird hier das Hostobjekt gesucht und weitergeleitet.

Parameter

hostname	Ein STRING mit dem Hostnamen.
testing	True, falls im Debug Modus gestartet wurde. In diesem Fall werden keine Kommandos an die Rechner uebertragen.

4.4.1.39 def scheduler.preparemigratevm (*vmob, qhost, qcore, heap*)

Diese Funktion bereitet das mirgieren von einer VM vor.

Das Ziel ist "unbekannt."

Parameter

vmob	VM Objekt, das mirgiert werden soll.
qhost	Quellhost, von dem mirgiert werden soll
qcore	Quellcore, von dem migriert werden soll
heap	Heap mit dem gearbeitet werden soll

4.4.1.40 def scheduler.preparevmchangedstate (*vmname, hostname, testing*)

Falls eine VM ihren Status wechselt, sucht diese Funktion die Objekte und beendet die Master VM.

Parameter

vmname	STRING, Name der VM.
hostname	STRING, Namen des Hosts.
testing	True, falls im Debug Modus gestartet wurde. In diesem Fall werden keine Kommandos an die Rechner uebertragen.

4.4.1.41 def scheduler.prepearevmtorestart (*vmname, hostname, testing*)

Bereitet das Neustarten einer VM vor.

Parameter

vmname	STRING, VM-Namen
hostname	String, Host-Namen.
testing	True, falls im Debug Modus gestartet wurde. In diesem Fall werden keine Kommandos an die Rechner uebertragen.

4.4.1.42 def scheduler.print_heap (*heap*)

4.4.1.43 def scheduler.printd (*message*)

Erzeugt ein feingranulares Logging.

Zu jedem Eintrag im Log wird ein Heap Dump generiert.

Parameter

message	Log Nachricht

4.4.1.44 def scheduler.redundancy_check (*q_hostobj, q_coreindex, m_vmobj, z_hostobj, z_coreindex*)

Prueft, ob Redundanz der zu verschiebenden VM bereits auf diesem Host ist.

Parameter

q_hostobj	Quell-Host-Objekt. Von hier aus soll migriert werden.
q_coreindex	Quell-Core-Index auf dem Quell-Host. Von hier aus soll migriert werden.
m_vmobj	Objekt der zu migrierenden VM
z_hostobj	Ziel-Host-Objekt, wohin die VM soll
z_coreindex	Ziel-Core-index auf Ziel-Host, wo die VM hin soll.

Rückgabe

True, keine Redundanz der VM auf diesem Zielhost/Zielcore. False, Redundanz bereits vorhanden, hier kann die VM nicht drauf.

4.4.1.45 def scheduler.removehost (*host, heap*)

Loescht ein Host aus dem heap.

Parameter

host	Es wird ein Hostobjekt mit mindestens dem Namen des hosts erwartet.
heap	Der heap, in dem der Host geloescht werden soll.

4.4.1.46 def scheduler.reschedule ( )

Rebalanciert den Heap, in dem die Core-Auslastungen ausgeglichen werden.

4.4.1.47 def scheduler.schedulenewvm (*newvm, heap, testing*)

Einsprungprunkt zum Starten einer neuen VM.

Parameter

newvm	VM Objekt, das gestartet werden soll.
testing	True, falls im Debug Modus gestartet wurde. In diesem Fall werden keine Kommandos an die Rechner uebertragen.
heap	Heap mit dem gearbeitet werden soll.

4.4.1.48 def scheduler.shutdown (*hostobj, testing*)

Diese Funktion bereitet einen Host zum Herunterfahren vor.

Dafuer werden so viele VMs migriert, wie moeglich.

Parameter

hostobj	Host Objekt, dass heruntergefahren werden soll.
testing	True, falls im Debug Modus gestartet wurde. In diesem Fall werden keine Kommandos an die Rechner uebertragen.

4.4.1.49 def scheduler.sort_heap_quadrupel ( )

Gibt eine nach Core-Auslastung sortierte Heap-Liste zurueck.

Rückgabe

Heap-Liste aufsteigend nach Core-Auslastung sortiert: [(hostname, hostobjekt, coreindex, corauslastung, aufsteigend_sortierte_vmliste-des-cores), ...]

4.4.1.50 def scheduler.startnewbackupvm (*backupvm, heap*)

Bereitet das Starten einer Backup VM vor.

Parameter

backupvm	VM Objekt, das gestartet werden soll.
heap	Heap mit dem gearbeitet werden soll.

4.4.1.51 def scheduler.startnewmastervm (*mastervm, heap*)

Bereitet das Starten einer neuen Mastervm vor.

Es wird ein Host gesucht.

Parameter

mastervm	VM Objekt, fuer das ein Platz gesucht werden soll.
heap	Heap mit dem gearbeitet werden soll.

4.4.1.52 def scheduler.startnewnodevm (*nodevm, heap*)

Node Objekt wird der Datenstruktur hinzugefuegt.

Bei einer Node ist keine Uebertragung noetig.

Parameter

nodevm	VM Objekt, das gestartet werden soll.
heap	Heap mit dem gearbeitet werden soll.

4.4.1.53 def scheduler.stopvm (*vmtostop, heap, command, command1, testing*)

Beendet eine VM.

Parameter

vmstop	VM Objekt, das beendet werden soll.
heap	Heap mit dem gearbeitet werden soll.
command	Wird nur fuer das Log benoetigt, da die Funktion fuer Beenden und Pausieren gleich ist.
command1	Wird nur fuer das Log benoetigt, da die Funktion fuer Beenden und Pausieren gleich ist.
testing	True, falls im Debug Modus gestartet wurde. In diesem Fall werden keine Kommandos an die Rechner uebertragen.

4.4.1.54 def scheduler.test_addfault (*x*)

Falls ein Fehler aufgetreten ist, wird dieser hier gespeichert.

Parameter

x	Es wird ein bool erwartet.

4.4.1.55 def scheduler.test_addfaultnumber (*str*)

Im Testmodus werden anstatt Strings Ints gespeichert.

Parameter

str	Nachricht die in der reply.xml gespeichert werden soll.

4.4.1.56 def scheduler.transmitstartbackup (*masterhost, akthost, backupvm, mincorepos*)

Uebertreagt das Kommando zum Starten einer Backup VM.

Parameter

masterhost	Host Objekt, auf dem die Master VM zu der zu startenden Backup VM laeuft.
akhost	Host, auf dem die Backup VM erstellt werden soll.
backupvm	VM Objekt.
mincorepos	Kern, auf dem das Backup erzeugt werden soll.

Rückgabe

True, falls die Übertragung des Kommandos erfolgreich war.

4.4.1.57 def scheduler.transmitstartmaster (*akthost, mastervm*)

Startet eine VM auf einem Host.

Parameter

akthost	Hostobjekt auf dem die VM gestartet werden soll.
mastervm	Vm Objekt das gestartet werden soll

Rückgabe

Falls das Starten der Master VM erfolgreich verlief, wird True zurueckgegeben, sonst False.

4.4.1.58 def scheduler.transmitstopvm (*lokvm, akthost*)

Uebertraegt das Kommando zum Beenden einer VM.

Parameter

lokvm	VM Objekt, das beendet werden soll.
akthost	Host Objekt, auf dem die VM laeuft und beendet werden soll.

4.4.1.59 def scheduler.vm_migrate (*migrate_list*)

Migriert Schritt fuer Schritt die Migrate-Liste ab.

Am Ende wird der entsprechend veraenderte Heap abgespeichert.

Parameter

migrate_list	Migrations-Liste in der Form (Quell-hostname, Quell-hostobj, Quell-coreindex, Zu--Migrierende-vm_name, Zu-Migrierende-vm_objekt, Ziel_hostname, Ziel_hostobjekt, Ziel_-coreindex)

Rückgabe

True, komplette Mig-Liste wurde abgearbeitet. False, Fehler bei der Migration (die, bis zu der Stelle migrierten VMs werden dennoch im Heap angepasst).

4.4.1.60 def scheduler.vm_move (*copy_all_list, q, m, z*)

In der angegebenen Heap-Liste wird die VM vom Quell-Host/Core zum Ziel-Host/Core verschoben.

Dabei wird es in die temp_migrate_list in Form von (quell_hostname, quell_hostobjekt, quell_coreindex, zu_-migrierende_vmname, zu_migrierende_m_vmobjekt, ziel_hostname, ziel_hostobjekt, ziel_coreindex)) eingetragen.

Parameter

copy_all_list	Heap-Liste: [(hostname, hostobjekt, coreindex, corauslastung, aufsteigend_sortierte_-vmliste-des-cores), ...]
q	Index in der Heap-Liste. Von hier aus soll migriert werden.
m	Index in der VM-Liste des q-Elements. Zeigt, welche VM migriert werden soll.
z	Index in der Heap-Liste. Hierhin soll migriert werden.

4.4.1.61 def scheduler.vm_to_move (*copy_all_list, q, m*)

Sucht eine Stelle , wohin die angegebene VM verschoben werden kann.

Parameter

copy_all_list	Heap-Liste: [(hostname, hostobjekt, coreindex, corauslastung, aufsteigend_sortierte_-vmliste-des-cores), ...]
q	Index in der Heap-Liste. Von hier aus soll migriert werden.
m	Index in der VM-Liste des q-Elements. Zeigt, welche VM migriert werden soll.

Rückgabe

True, VM konnte verschoben werden. False, diese VM konnte nicht verschoben werden.

4.4.1.62 def scheduler.vmdown (*vmtorestart, lokhost, heap, testing, sendcommand*)

Diese Funktion startet eine abgestuerzte VM neu.

Parameter

vmtorestart	VM Objekt, das neu gestartet werden soll.
lokhost	Host Objekt, auf dem die VM lief.
testing	True, falls im Debug Modus gestartet wurde. In diesem Fall werden keine Kommandos an die Rechner uebertragen.

sendcommand	ueberfluessig
heap	Heap mit dem gearbeitet werden soll

4.4.2 Variablen-Dokumentation

4.4.2.1 tuple scheduler.all_list = list()

4.4.2.2 int scheduler.BACKUP = 1

4.4.2.3 tuple scheduler.found_places_list = list()

4.4.2.4 scheduler.globlogging = True

4.4.2.5 scheduler.globtesting = False

4.4.2.6 int scheduler.M = 1

4.4.2.7 int scheduler.MASTER = 0

State von dem Task.

4.4.2.8 int scheduler.maxload_of_core = 1

4.4.2.9 int scheduler.MB = 2

4.4.2.10 int scheduler.MBN = 3

Redundanz mit der die VM gestartet werden soll.

4.4.2.11 tuple scheduler.migrate_list = list()

4.4.2.12 string scheduler.NEUEZEILE = '\n '

Trennzeichen von Zeilen fuer die reply.xml.

4.4.2.13 int scheduler.NODE = 2

4.4.2.14 tuple scheduler.temp_migrate_list = list()

4.4.2.15 string scheduler.TRENNZEICHEN = ';'

Trennzeichen fuer die verschiedenen Fehlercodes.

4.5 tempreply-Namensbereichsreferenz

Klassen

- class Tempreply

    Wird zum generieren der reply.xml benoetigt.

4.6 testen-Namensbereichsreferenz

Funktionen

- def start_task_test
  Normaler Funktionstest der LMU von verschiedenen Diensten.
- def start_pause_task_test
  Funktionstest der LMU.
- def start_stop_task_test
  Test der LMU, bei dem der Dienst zuerst gestartet wird, dann gestoppt wird.
- def start2times_task_test
  Es wird zwei mal der gleiche Task gestartet.
- def wcet_longer_period_test
  Bei den Tasks ist WCET immer groesser als Periode.
- def configuration_test
  Die Host werden unterschiedlich Konfiguriert.
- def task_xml_error_test
  Es wird eine Fehlerhafte task.xml generiert.
- def action_xml_error_test
  Es wird eine fehlerhafte action.xml generiert.
- def belastung_test
  beliebig viele Tasks werden in einem Zeitraum hintereinander gestartet.
- def create_directory
  Hilfsfunktionen #.
- def move_to_directory
  Am Ende des Tests werden alle task.xml, action.xml, reply.xml und hosts.xml, die im Test generiert wurden, in den Testordner verschoben.
- def move_actionxml_and_replyxml
  Zwischen dem Test werden action.xml und reply.xml zum Testordner verschoben.
- def create_action_xml
  Erstellung von einer action.xml.
- def create_action_xml_1
- def create_action_xml_2
- def create_task_xml
  Erstellung von einer richtigen task.xml.
- def create_task_xml_0
  Erstellung von einer fehlerhaten task.xml.
- def create_task_xml_1
  Erstellung von einer fehlerhaten task.xml.
- def create_task_xml_2
  Erstellung von einer fehlerhaten task.xml.
- def create_random_hosts_xml
  Erstellung von einer hosts.xml, dass beliebig viele Hosts konfiguriert werden.
- def create_one_host_xml
  Erstellung von einer hosts.xml, dass nur ein Host verfuegbar ist.
- def create_two_hosts_xml
  Erstellung von einer hosts.xml, dass zwei Hosts verfuegbar sind.
- def create_three_hosts_xml
  Erstellung von einer hosts.xml, dass drei Hosts verfuegbar sind.
- def create_four_hosts_xml
  Erstellung von einer hosts.xml, dass vier Hosts verfuegbar sind.

- def get_reply
  reply.xml Abruf 5 mal, jeweils 5 Sekunden lang
- def check_reply
  Checkt ob reply.xml in lmu_outbox existiert.
- def get_reply_message
  Fehlercode auffangen.
- def message_list
  Fehlerliste erstellen.
- def failure_check
  Fehlervergleich.
- def error_check
  Ergebnis des Tests wird vergleichen, Es wird ueberprueft, ob es unsere Erwartungen erfuellen konnte.

Variablen

- string pfad = "/home/nas/lmu-test/"
  Testfaelle #.

4.6.1 Dokumentation der Funktionen

4.6.1.1 def testen.action_xml_error_test ( )

Es wird eine fehlerhafte action.xml generiert.

4.6.1.2 def testen.belastung_test ( )

beliebig viele Tasks werden in einem Zeitraum hintereinander gestartet.

Ob die LMU ueberbelastet ist, haengt es von verfuegaren Hosts, VMs und Cores.

4.6.1.3 def testen.check_reply (*reply_xml*)

Checkt ob reply.xml in lmu_outbox existiert.

Parameter

reply_xml	entsprechende reply.xml, die bereits von LMU zurueckgegeben und in "lmu_outbox" in Testordner generiert wurde

Rückgabe

True falls entsprechende reply.xml vorhanden in den Pfad ist, sonst False

4.6.1.4 def testen.configuration_test ( )

Die Host werden unterschiedlich Konfiguriert.

Dabei gibt es nicht immer ausreichende Ressourcen für die Dienste

4.6.1.5 def testen.create_action_xml (*testCommand*, *testTarget*, *timestamp*)

Erstellung von einer action.xml.

Parameter

testCommand	erwartete Befehl fuer den Task, der zur LMU schickt werden soll
testTarget	Name der Tasks
timestamp	Timestamp, um die entsprechende action.xml zu erzeugen

4.6.1.6 def testen.create_action_xml_1 (*testCommand, testTarget, timestamp*)

4.6.1.7 def testen.create_action_xml_2 (*testCommand, testTarget, timestamp*)

4.6.1.8 def testen.create_directory (*dir_name*)

Hilfsfunktionen #.

Erstellung von einem Testordner fuer den Testfall

Parameter

dir_name	Pfad des Testordners, der fuer jede Testfaelle erstellt werden soll

4.6.1.9 def testen.create_four_hosts_xml (*directory*)

Erstellung von einer hosts.xml, dass vier Hosts verfuegbar sind.

Parameter

directory	Pfad der zu generierter hosts.xml

4.6.1.10 def testen.create_one_host_xml (*directory*)

Erstellung von einer hosts.xml, dass nur ein Host verfuegbar ist.

Parameter

directory	Pfad der zu generierter hosts.xml

4.6.1.11 def testen.create_random_hosts_xml (*directory*)

Erstellung von einer hosts.xml, dass beliebig viele Hosts konfiguriert werden.

(mindestens eins und hoechstens vier)

Parameter

directory	Pfad der zu generierter hosts.xml

4.6.1.12 def testen.create_task_xml (*test_name, testRedundancy, testPriority, testPeriod, testWCET, testIP, testMAC, testImagPath, testCreatedOn, testOS*)

Erstellung von einer richtigen task.xml.

Parameter

test_name	Name der Tasks in einem Testfall
testRedundancy	Redundanz von dem Task
testPriority	Prioritaet von dem Task
testWCET	Worst Case Execution Time von dem Task
testIP	IP-Adresse von dem Task
testMAC	MAC-Adresse von dem Task
testImagPath	Pfad der Image-Datei
testCreatedOn	Timestamp, die task.xml generiert wurde

4.6.1.13 def testen.create_task_xml_0 (*test_name, testRedundancy, testPriority, testPeriod, testWCET, testIP, testMAC, testImagPath, testCreatedOn*)

Erstellung von einer fehlerhaten task.xml.

Parameter

ebenfalls	wie create_task_xml

4.6.1.14 def testen.create_task_xml_1 (*test_name, testRedundancy, testPriority, testPeriod, testWCET, testIP, testMAC, testImagPath, testCreatedOn*)

Erstellung von einer fehlerhaten task.xml.

Parameter

ebenfalls	wie create_task_xml

4.6.1.15 def testen.create_task_xml_2 (*test_name, testRedundancy, testPriority, testPeriod, testWCET, testIP, testMAC, testImagPath, testCreatedOn*)

Erstellung von einer fehlerhaten task.xml.

Parameter

ebenfalls	wie create_task_xml

4.6.1.16 def testen.create_three_hosts_xml (*directory*)

Erstellung von einer hosts.xml, dass drei Hosts verfuegbar sind.

Parameter

directory	Pfad der zu generierter hosts.xml

4.6.1.17 def testen.create_two_hosts_xml (*directory*)

Erstellung von einer hosts.xml, dass zwei Hosts verfuegbar sind.

Parameter

directory	Pfad der zu generierter hosts.xml

4.6.1.18 def testen.error_check (*reply_name*, *err*, *ts*, *directory*)

Ergebnis des Tests wird vergleichen, Es wird ueberprueft, ob es unsere Erwartungen erfuellen konnte.

Parameter

reply_name	Name der generierten reply.xml
err	Fehlercodes aus reply.xml
ts	Timestamp, die entsprechende reply.xml generiert wurde
directory	Pfad sich die entsprechende reply.xml befindet

Rückgabe

True falls erwartete Fehlercodes in der Fehlerliste enthalten, sonst False

4.6.1.19 def testen.failure_check (*message_string*)

Fehlervergleich.

Parameter

message_string	ausfuehrlicher Inhalt von der Fehlerliste

Rückgabe

True falls Fehlerliste aus reply.xml nur '0' enthaelt, also 'Alles OK.', sonst False

4.6.1.20 def testen.get_reply (*reply_xml*)

reply.xml Abruf 5 mal, jeweils 5 Sekunden lang

Parameter

reply_xml	entsprechende reply.xml, die bereits von LMU zurueckgegeben und in "lmu_outbox" in Testordner generiert wurde

Rückgabe

True falls erwartete reply.xml zwischen die 5 mal Abrufe gefunden hat

4.6.1.21 def testen.get_reply_message (*reply_name, ts, directory*)

Fehlercode auffangen.

Parameter

reply_name	Name der generierten reply.xml
ts	Timestamp, die entsprechende reply.xml generiert wurde
directory	Pfad sich die entsprechende reply.xml befindet

Rückgabe

True falls failure_check(message_string) True ist, sonst False

4.6.1.22 def testen.message_list (*messages, ts, directory*)

Fehlerliste erstellen.

Parameter

messages	Inhalt vom Ergebnis der reply.xml, also welche Fehlercodes in reply.xml beschrieben wurden
ts	Timestamp, die entsprechende reply.xml generiert wurde
directory	Pfad sich die entsprechende reply.xml befindet

4.6.1.23 def testen.move_actionxml_and_replyxml (*test_directory*, *action_xml*, *reply_xml*)

Zwischen dem Test werden action.xml und reply.xml zum Testordner verschoben.

Parameter

test_directory	Pfad des generierten Ordners fuer die Testfaelle
action_xml	zu verschiebender action.xml Datei
reply_xml	zu verschiebender reply.xml Datei

4.6.1.24 def testen.move_to_directory (*test_directory*, *task_xml*, *action_xml*, *reply_xml*, *hosts_xml*)

Am Ende des Tests werden alle task.xml, action.xml, reply.xml und hosts.xml, die im Test generiert wurden, in den Testordner verschoben.

Parameter

test_directory	Pfad des generierten Ordners fuer die Testfaelle
task_xml	zu verschiebender task.xml Datei
action_xml	zu verschiebender action.xml Datei
reply_xml	zu verschiebender reply.xml Datei
hosts_xml	zu verschiebender hosts.xml Datei

4.6.1.25 def testen.start2times_task_test ( )

Es wird zwei mal der gleiche Task gestartet.

4.6.1.26 def testen.start_pause_task_test ( )

Funktionstest der LMU.

Der Dienst wird zuerst gestartet, dann pausiert

4.6.1.27 def testen.start_stop_task_test ( )

Test der LMU, bei dem der Dienst zuerst gestartet wird, dann gestoppt wird.

4.6.1.28 def testen.start_task_test ( )

Normaler Funktionstest der LMU von verschiedenen Diensten.

4.6.1.29 def testen.task_xml_error_test ( )

Es wird eine Fehlerhafte task.xml generiert.

4.6.1.30 def testen.wcet_longer_period_test ( )

Bei den Tasks ist WCET immer groesser als Periode.

4.6.2 Variablen-Dokumentation

4.6.2.1 string testen.pfad = "/home/nas/lmu-test/"

Testfaelle #.

Allgemeiner Pfad fuer alle Testfaelle

4.7 transmitcommand-Namensbereichsreferenz

Funktionen

- def connect_to_socket

  Attempts to establish a connection to a remote TCP-socket.
- def transmit_message

  Pickles the passed command list and attempts to transmit it to the remote LMU interface.
- def transmit_state

  Pickles the passed command list and attempts to transmit it to the remote LMU interface.
- def close_socket

  Attempts to close the global TCP-Socket mySocket.
- def dom_create_cfg

  Creates the configuration file for the new domain.
- def dom_create

  Creates a new domain on the remote host.
- def dom_destroy

  Destroys a running domain on the remote host.
- def dom_migrate

  Migrates a running domain to another host.
- def dom_remus

  Creates a remus replication of the specified domain with the specified replication interval (default: 500).
- def dom_set_period_slice

  (SEDF only) Adapts period and slice for the specified domain name.
- def dom_vcpu_pin

  Pins the specified domains VCPU to the PCPU.
- def dom_notify_incoming_migration

  Notifies the remote host about an incoming VM migration.
- def dom_notify_incoming_remus

  Notifies the remote host about an incoming remus replication.
- def dom_notify_intended_failover

  Notifies the remote host about an intended remus failover.
- def host_check_sanity

  Check sanity of remote host.
- def host_get_state

  Check sanity of remote host.

Variablen

- mySocket = None
- int socketTimeout = 5
- int extendedSocketTimeout = 20
- tuple parser = OptionParser()

4.7.1 Dokumentation der Funktionen

4.7.1.1 def transmitcommand.close_socket ( )

Attempts to close the global TCP-Socket mySocket.

4.7.1.2 def transmitcommand.connect_to_socket (*destination_ip, destination_port*)

Attempts to establish a connection to a remote TCP-socket.

Parameter

destination_ip	Remote IP address
destination_port	Remote Port

4.7.1.3 def transmitcommand.dom_create (*xen_server_ip, xen_server_port, service_name*)

Creates a new domain on the remote host.

Please note that the configuration file has to exist for this command to succeed. If no configuration has been created yet, use the 'dom_create_cfg'-command to create a configuration before issuing this command.

Parameter

xen_server_ip	Remote IP address
xen_server_port	Remote Port
service_name	Service Name

4.7.1.4 def transmitcommand.dom_create_cfg (*xen_server_ip, xen_server_port, service_name, ip, mac, pcpu, period, slice, priority, ostype*)

Creates the configuration file for the new domain.

Parameter

ip	IP address of virtual machine
mac	MAC address of virtual machine
pcpu	Physical CPU to pin VCPU to
period	Period of new virtual machine
slice	Slice of new virtual machine
priority	Priority of new virtual machine

4.7.1.5 def transmitcommand.dom_destroy (*xen_server_ip, xen_server_port, service_name*)

Destroys a running domain on the remote host.

Parameter

xen_server_ip	Remote IP address
xen_server_port	Remote Port
service_name	Service Name

4.7.1.6 def transmitcommand.dom_migrate (*xen_server_ip, xen_server_port, service_name, destination_host*)

Migrates a running domain to another host.

Parameter

xen_server_ip	Remote IP address
xen_server_port	Remote Port
service_name	Service Name
destination_host	Migration Destination Host

4.7.1.7 def transmitcommand.dom_notify_incoming_migration (*xen_server_ip, xen_server_port, service_name*)

Notifies the remote host about an incoming VM migration.

Parameter

xen_server_ip	Remote IP address
xen_server_port	Remote Port
service_name	Service Name

4.7.1.8 def transmitcommand.dom_notify_incoming_remus (*xen_server_ip, xen_server_port, service_name*)

Notifies the remote host about an incoming remus replication.

Parameter

xen_server_ip	Remote IP address
xen_server_port	Remote Port
service_name	Service Name

4.7.1.9 def transmitcommand.dom_notify_intended_failover (*xen_server_ip, xen_server_port, service_name*)

Notifies the remote host about an intended remus failover.

Parameter

xen_server_ip	Remote IP address
xen_server_port	Remote Port
service_name	Service Name

4.7.1.10 def transmitcommand.dom_remus (*xen_server_ip, xen_server_port, service_name, destination_host, replication_interval =* `500`)

Creates a remus replication of the specified domain with the specified replication interval (default: 500).

Parameter

xen_server_ip	Remote IP address
xen_server_port	Remote Port
service_name	Service Name
destination_host	Remus Destination Host
replication_-interval	Remus Replication interval

4.7.1.11 def transmitcommand.dom_set_period_slice (*xen_server_ip, xen_server_port, service_name, period, slice*)

(SEDF only) Adapts period and slice for the specified domain name.

Parameter

xen_server_ip	Remote IP address
xen_server_port	Remote Port
service_name	Service Name
period	New Period for selected service name
slice	New Slice for selected service name

4.7.1.12 def transmitcommand.dom_vcpu_pin (*xen_server_ip, xen_server_port, service_name, vcpu, pcpu*)

Pins the specified domains VCPU to the PCPU.

Parameter

xen_server_ip	Remote IP address
xen_server_port	Remote Port
service_name	Service Name
vcpu	Virtual CPU to pin
pcpu	Physical CPU to pin selected VCPU to

4.7.1.13 def transmitcommand.host_check_sanity (*xen_server_ip, xen_server_port*)

Check sanity of remote host.

Parameter

xen_server_ip	Remote IP address
xen_server_port	Remote Port

4.7.1.14 def transmitcommand.host_get_state (*xen_server_ip, xen_server_port*)

Check sanity of remote host.

Parameter

xen_server_ip	Remote IP address
xen_server_port	Remote Port

4.7.1.15 def transmitcommand.transmit_message (*data*)

Pickles the passed command list and attempts to transmit it to the remote LMU interface.

Parameter

data	Data to transmit

4.7.1.16 def transmitcommand.transmit_state (*data*)

Pickles the passed command list and attempts to transmit it to the remote LMU interface.

Parameter

data	Data to transmit

4.7.2 Variablen-Dokumentation

4.7.2.1 int transmitcommand.extendedSocketTimeout = 20

4.7.2.2 transmitcommand.mySocket = None

4.7.2.3 tuple transmitcommand.parser = OptionParser()

4.7.2.4 int transmitcommand.socketTimeout = 5

4.8 verzeichnisscanner-Namensbereichsreferenz

Funktionen

- def check_xml_file
  Es wird ueberprueft, ob eine action_∗.xml vorhanden ist.
- def load_taskxml
  Es wird eine task.xml geladen und ein VM Objekt zurueckgegeben.
- def load_xml_file
  Diese Funktion liest eine action.xml aus und startet die jeweilige Aktion.
- def ishostinheap
  Ueberprueft, ob sich der Host bereits im Heap befindet.
- def is_host_in_xml
  Ueberprueft, ob sich ein Hostobjekt in der host.xml befindet.
- def load_hosts_xml_file_for_check
  Gib den Inhalt der host.xml als Liste mit Hostobjekten zurueck.
- def load_hosts_xml_file
  Laedt die hosts.xml und ueberprueft ob die Hosts bereits im Heap sind.
- def getIP
  Liefert die aktuelle IP zurueck.
- def execCommand
  Fuehrt ein Kommando als Subprozess aus.
- def start_webserver
  Startet den Webserver.
- def start_rep_lmu
  Repliziert die LMU.
- def initPipe
  Initialisiert die Pipe.
- def readpipe
  Liest aus der Pipe.

- def checkrunningfiles

  Verschiebt task.xml und action.xml von runnig nach pending.
- def getsysteminfo

  Empfängt von allen hosts im Heap die aktuell laufenden VMs und überprüft, ob die aktuelle Konfiguration mit der aus dem Heap übereinstimmt.
- def runme

  Endlosschleife.

Variablen

- string NEUEZEILE = '\n '

  Trennzeichen von den Zeilen fuer die reply.xml.
- int MASTER = 0

  Status eines Tasks.
- int BACKUP = 1
- int NODE = 2
- int MBN = 3

  Redundanz mit der die VM gestartet werden soll.
- int MB = 2
- int M = 1
- testing = False
- string globimgpath = ''
- string pipe_name = '/tmp/pipe_globmon-lmu'
- pipein = None
- string lokalerpfad = '/home/nas/'
- int lastchange = 0

4.8.1 Dokumentation der Funktionen

4.8.1.1 def verzeichnisscanner.check_xml_file ( )

Es wird ueberprueft, ob eine action_∗.xml vorhanden ist.

Rückgabe

Falls eine action_∗.xml vorhanden ist, wird True zurueckgegeben

4.8.1.2 def verzeichnisscanner.checkrunningfiles ( )

Verschiebt task.xml und action.xml von runnig nach pending.

Diese Funktion wird nach dem ersten Start ausgefuehrt.

4.8.1.3 def verzeichnisscanner.execCommand (*cmd*)

Fuehrt ein Kommando als Subprozess aus.

4.8.1.4 def verzeichnisscanner.getIP ( )

Liefert die aktuelle IP zurueck.

4.8.1.5 def verzeichnisscanner.getsysteminfo ( )

Empfängt von allen hosts im Heap die aktuell laufenden VMs und überprüft, ob die aktuelle Konfiguration mit der aus dem Heap übereinstimmt.

Diese Funktion ist noch nicht getestet und wird daher nicht verwendet.

4.8.1.6 def verzeichnisscanner.initPipe ( )

Initialisiert die Pipe.

4.8.1.7 def verzeichnisscanner.is_host_in_xml (*hostobj*)

Ueberprueft, ob sich ein Hostobjekt in der host.xml befindet.

Parameter

hostobj	Hostobjekt, das ueberprueft werden soll

Rückgabe

True, falls sich der Host schon im Heap befindet

4.8.1.8 def verzeichnisscanner.ishostinheap (*hostob*)

Ueberprueft, ob sich der Host bereits im Heap befindet.

Ausserdem wird ueberprueft, ob der Host erreichbar ist und sich in der hosts.xml befindet. Es werden nur Hosts in den Heap eingefuegt, die in der hosts.xml vorhanden und erreichbar sind.

Parameter

hostob	Es wird ein Hostobjekt mit Namen und IP erwartet

Rückgabe

True falls sich ein Host mit dieser IP oder Namen bereits im heap befindet, sonst False

4.8.1.9 def verzeichnisscanner.load_hosts_xml_file ( )

Laedt die hosts.xml und ueberprueft ob die Hosts bereits im Heap sind.

Falls nicht, werden sie hinzugefuegt.

4.8.1.10 def verzeichnisscanner.load_hosts_xml_file_for_check ( )

Gib den Inhalt der host.xml als Liste mit Hostobjekten zurueck.

4.8.1.11 def verzeichnisscanner.load_taskxml (*pfad*)

Es wird eine task.xml geladen und ein VM Objekt zurueckgegeben.

Parameter

pfad	Es wird der Pfad zur task.xml erwartet

Rückgabe

Es wird ein VM Objekt zurueckgegeben

4.8.1.12 def verzeichnisscanner.load_xml_file (*fileName, ts*)

Diese Funktion liest eine action.xml aus und startet die jeweilige Aktion.

Parameter

fileName	Liest die action.xml mit dem entsprechendem Namen ein.
ts	Timestamp, um die entsprechende Reply zu erzeugen

4.8.1.13 def verzeichnisscanner.readpipe ( )

Liest aus der Pipe.

4.8.1.14 def verzeichnisscanner.runme ( )

Endlosschleife.

Hier wird die Pipe ausgelesen und auf neue action.xml und task.xml ueberprueft

4.8.1.15 def verzeichnisscanner.start_rep_lmu ( )

Repliziert die LMU.

4.8.1.16 def verzeichnisscanner.start_webserver ( )

Startet den Webserver.

4.8.2 Variablen-Dokumentation

4.8.2.1 int verzeichnisscanner.BACKUP = 1

4.8.2.2 string verzeichnisscanner.globimgpath = ”

4.8.2.3 int verzeichnisscanner.lastchange = 0

4.8.2.4 string verzeichnisscanner.lokalerpfad = ’/home/nas/’

4.8.2.5 int verzeichnisscanner.M = 1

4.8.2.6 int verzeichnisscanner.MASTER = 0

Status eines Tasks.

4.8.2.7 int verzeichnisscanner.MB = 2

4.8.2.8 int verzeichnisscanner.MBN = 3

Redundanz mit der die VM gestartet werden soll.

4.8.2.9 string verzeichnisscanner.NEUEZEILE = '\n '

Trennzeichen von den Zeilen fuer die reply.xml.

4.8.2.10 int verzeichnisscanner.NODE = 2

4.8.2.11 string verzeichnisscanner.pipe_name = '/tmp/pipe_globmon-lmu'

4.8.2.12 verzeichnisscanner.pipein = None

4.8.2.13 verzeichnisscanner.testing = False

4.9 vm-Namensbereichsreferenz

Klassen

- class VM

  VM Objekte enthalten alle Informationen ueber die VM.

Kapitel 5

Klassen-Dokumentation

5.1 host.Host Klassenreferenz

Host Objekte entahlten alle Informationen ueber die Hosts und eine Liste von VMs.

Öffentliche Methoden

- def __init__
- def __lt__
- def getname

  Gibt den Namen des Hosts zurueck.
- def getip

  Gibt die IP des Hosts zurueck.
- def getnumberofcores

  Gibt die Anzahl der Kerne zurueck.
- def getmaxload

  Gibt die maximale Auslastung der Cores zurueck.
- def setcoreload

  Setzt die Last fuer einen Kern.
- def getcoreload

  Gibt die Last des Kerns zurueck.
- def getmincoreload

  Gib die Minimale Auslastung aller Kerne zurueck.
- def getmaxcoreload

  Gib die Maximale Auslastung aller Kerne zurueck.
- def getposmincoreload

  Gibt den Kern mit der minimalen Auslastung zurueck.
- def getposmaxcoreload

  Gibt den Kern mit der maximalen Auslastung zurueck.
- def addcoreload

  Fuegt einem Kern Last hinzu.
- def removecoreload

  Entfernt von einem Kern Last.
- def addvm

  Fuegt der VM Liste ein VM Objekt hinzu.
- def getvmlist

  Gibt eine Liste aller VMs auf diesem Host zurueck.

- def getvmlistofcore

  Gibt alle VMs auf einem Kern zurueck.
- def removevm

  Entfernt das VM Objekt an einer bestimmten Stelle.
- def removevmlist

  Leert die VM Liste eines Hosts.
- def setvmstate

  Setzt den Status einer VM.
- def getallvms

  Gibt eine Liste die aus Tupeln mit Kern und VM Objekt besteht zurueck.
- def getvm

  Gibt das VM Objekt zurueck.
- def getminloadofcore

  Sucht die kleinste VM auf dem angegebenen Core.
- def getvmlistcore

  Gibt die VM-Liste eines bestimmten Cores zurueck.
- def getvmlistsort

  Gibt eine aufsteigend sortierte VM-Liste des angegebenen Cores zurueck.

Öffentliche Attribute

- name
- ip
- numberofcores
- maxload
- coreload
- vmlist

5.1.1 Ausführliche Beschreibung

Host Objekte entahlten alle Informationen ueber die Hosts und eine Liste von VMs.

5.1.2 Beschreibung der Konstruktoren und Destruktoren

5.1.2.1 def host.Host.__init__ (*self, name, ip, numberofcores, maxload* = `1.0`)

5.1.3 Dokumentation der Elementfunktionen

5.1.3.1 def host.Host.__lt__ (*self, other*)

5.1.3.2 def host.Host.addcoreload (*self, corenumber, load*)

Fuegt einem Kern Last hinzu.

5.1.3.3 def host.Host.addvm (*self, lokvm, core*)

Fuegt der VM Liste ein VM Objekt hinzu.

Parameter

lokvm	VM Objekt welches hinzugefuegt werden soll
core	Nummer des Kerns, auf dem die VM gestartet wurde

5.1.3.4 def host.Host.getallvms (*self*)

Gibt eine Liste die aus Tupeln mit Kern und VM Objekt besteht zurueck.

5.1.3.5 def host.Host.getcoreload (*self, corenumber*)

Gibt die Last des Kerns zurueck.

Parameter

corenumber	Nummer des Kerns, dessen Last zurueck gegeben werden soll

5.1.3.6 def host.Host.getip (*self*)

Gibt die IP des Hosts zurueck.

5.1.3.7 def host.Host.getmaxcoreload (*self*)

Gib die Maximale Auslastung aller Kerne zurueck.

5.1.3.8 def host.Host.getmaxload (*self*)

Gibt die maximale Auslastung der Cores zurueck.

5.1.3.9 def host.Host.getmincoreload (*self*)

Gib die Minimale Auslastung aller Kerne zurueck.

5.1.3.10 def host.Host.getminloadofcore (*self, core*)

Sucht die kleinste VM auf dem angegebenen Core.

Rückgabe

Gibt die kleinste VM auf diesem Kern zurueck

Parameter

core	Kern, von dem die VM zurueckgegeben werden soll

5.1.3.11 def host.Host.getname (*self*)

Gibt den Namen des Hosts zurueck.

5.1.3.12 def host.Host.getnumberofcores (*self*)

Gibt die Anzahl der Kerne zurueck.

5.1.3.13 def host.Host.getposmaxcoreload (*self*)

Gibt den Kern mit der maximalen Auslastung zurueck.

5.1.3.14 def host.Host.getposmincoreload (*self*)

Gibt den Kern mit der minimalen Auslastung zurueck.

5.1.3.15 def host.Host.getvm (*self*, *vmtofind*)

Gibt das VM Objekt zurueck.

Rückgabe

Gibt das VM Objekt zurueck

Parameter

vmtofind	VM Objekt, das zurueckgegeben werden soll

5.1.3.16 def host.Host.getvmlist (*self*)

Gibt eine Liste aller VMs auf diesem Host zurueck.

5.1.3.17 def host.Host.getvmlistcore (*self*, *core*)

Gibt die VM-Liste eines bestimmten Cores zurueck.

Rückgabe

VM-Liste in folgender Form: [(vm_name, vm_objekt, groesse-der-vm),...]

Parameter

core	Kern, dessen VM-Liste ausgegeben werden soll

5.1.3.18 def host.Host.getvmlistofcore (*self*, *lokcore*)

Gibt alle VMs auf einem Kern zurueck.

Parameter

lokcore	Kern, von dem die VMs zurueck gegeben werden sollen

Rückgabe

Gibt eine Liste mit VM objekten zurueck

5.1.3.19 def host.Host.getvmlistsort (*self*, *core*)

Gibt eine aufsteigend sortierte VM-Liste des angegebenen Cores zurueck.

Rückgabe

aufsteigend sortierte VM-Liste in der Form: [(vm_name, vm_objekt, groesse-der-vm),...]

Parameter

core	Kern, dessen VM-Liste man sortiert erhalten moechte

5.1.3.20 def host.Host.removecoreload (*self, corenumber, load*)

Entfernt von einem Kern Last.

5.1.3.21 def host.Host.removevm (*self, index*)

Entfernt das VM Objekt an einer bestimmten Stelle.

Parameter

index	Index des VM Objekts, das geloescht werden soll.

5.1.3.22 def host.Host.removevmlist (*self*)

Leert die VM Liste eines Hosts.

5.1.3.23 def host.Host.setcoreload (*self, corenumber, load*)

Setzt die Last fuer einen Kern.

Parameter

corenumber	Nummer des Kerns dessen Load geandert werden soll
load	Auslastung des kerns

5.1.3.24 def host.Host.setvmstate (*self, index, state*)

Setzt den Status einer VM.

Parameter

index	Index der VM dessen Status geandert werden soll
state	Status, auf den die VM gesetzt werden soll

5.1.4 Dokumentation der Datenelemente

5.1.4.1 host.Host.coreload

5.1.4.2 host.Host.ip

5.1.4.3 host.Host.maxload

5.1.4.4 host.Host.name

5.1.4.5 host.Host.numberofcores

5.1.4.6 host.Host.vmlist

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Datei:

- host.py

5.2 tempreply.Tempreply Klassenreferenz

Wird zum generieren der reply.xml benoetigt.

Öffentliche Methoden

- def __init__
- def getfault

  Gibt zuerueck, ob waehrend der ausfuehrung ein Fehler aufgetreten ist.
- def getmessage

  Gibt die Nachricht zurueck.
- def setmessage

  Fuegt eine Nachricht hinzu.
- def setfault

  Setzt einen Fehler.

Öffentliche Attribute

- fault
- message

5.2.1 Ausführliche Beschreibung

Wird zum generieren der reply.xml benoetigt.

5.2.2 Beschreibung der Konstruktoren und Destruktoren

5.2.2.1 def tempreply.Tempreply.__init__ (*self, fault, message*)

5.2.3 Dokumentation der Elementfunktionen

5.2.3.1 def tempreply.Tempreply.getfault (*self*)

Gibt zuerueck, ob waehrend der ausfuehrung ein Fehler aufgetreten ist.

5.2.3.2 def tempreply.Tempreply.getmessage (*self*)

Gibt die Nachricht zurueck.

5.2.3.3 def tempreply.Tempreply.setfault (*self, xyz*)

Setzt einen Fehler.

5.2.3.4 def tempreply.Tempreply.setmessage (*self, mes*)

Fuegt eine Nachricht hinzu.

5.2.4 Dokumentation der Datenelemente

5.2.4.1 tempreply.Tempreply.fault

5.2.4.2 tempreply.Tempreply.message

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Datei:

- tempreply.py

5.3 vm.VM Klassenreferenz

VM Objekte enthalten alle Informationen ueber die VM.

Öffentliche Methoden

- def __init__
  Das VM Objekt enthaelt Informationen ueber die VM selber.
- def getid
  veraltet.
- def getwcet
  Gibt die WCET der Vm zurueck.
- def getperiod
  Gibt die Periode der VM zurueck.
- def gettaskname
  Gibt den Namen der VM zurueck.
- def getload
  Gibt die Auslastung, die die VM verursacht wieder.
- def getstate
  Gibt den Status der VM, als M oder Backup oder Node, zurueck.
- def setstate
  Gibt der VM einen neuen Status.
- def getredundancy
  Gibt die Redundanz der VM zurueck (M, MB ode MBN)
- def getmac
  Gibt die Macadresse der VM zurueck.
- def setmac
  Setzt eine neue Macadresse fuer die VM.
- def getimgpath
  Gibt den Pfad des Images zu dieser VM zurueck.
- def setimgpath
  Setzt einen neuen Pfad fuer die image der VM.
- def getip
  Gibt die IP der VM zurueck.
- def setip
  Setzt eine neue IP fuer die VM.
- def getcreatedon
  Gibt Erstellungszeitpunkt der VM zurueck.
- def setcreatedon
  Setzt einen Erstellungszeitpunkt fuer die VM.

- def setos

  Setzt den Betriebssystemtyp.
- def getos

  Gibt das aktuelle Betriebssystem zurueck.

Öffentliche Attribute

- ident
- wcet
- period
- taskname
- redundancy
- state
- mac
- imgpath
- ip
- createdon
- operating

5.3.1 Ausführliche Beschreibung

VM Objekte enthalten alle Informationen ueber die VM.

5.3.2 Beschreibung der Konstruktoren und Destruktoren

5.3.2.1 def vm.VM.__init__ (*self*, *ident*, *taskname*, *redundancy*, *priority*, *period*, *wcet*, *ip*, *mac*, *imgpath*, *createdon*, *state*, *operating* = "ciao")

Das VM Objekt enthaelt Informationen ueber die VM selber.

Es stehen nur getter und setter Methoden zur Verfuegung, ausserdem ist auch ein direkter Zugriff moeglich.

Parameter

ident	Identifikationsnummer, wird jedoch nicht benutzt
taskname	Der Name des Tasks dient als Identifikation
redundancy	Die Redundanz, mit der die VM erzeugt werden soll. Also M(1), MB(2), MBN(3)
priority	Prioriteat der VM
period	Periode der VM
wcet	WCET der VM
ip	IP Adresse, die in die cfg geschrieben wird
mac	MAC Adresse, die in die
imgpath	Adresse zu dem Image
createdon	Zeitpunkt, an dem die xml Datei erstellt wurde
state	Der State ist initial leer. Erst wenn die VM gestartet wird, wird der State von dem VM Objekt geandert. Moegliche Werte: MASTER(0), BACKUP(1), NODE(2)
operating	Es wird für ciao eine andere Konfiguration benötigt, asl für MiniOS

5.3.3 Dokumentation der Elementfunktionen

5.3.3.1 def vm.VM.getcreatedon (*self*)

Gibt Erstellungszeitpunkt der VM zurueck.

5.3.3.2 def vm.VM.getid (*self*)

veraltet.

ID gibt es nicht mehr.

5.3.3.3 def vm.VM.getimgpath (*self*)

Gibt den Pfad des Images zu dieser VM zurueck.

5.3.3.4 def vm.VM.getip (*self*)

Gibt die IP der VM zurueck.

5.3.3.5 def vm.VM.getload (*self*)

Gibt die Auslastung, die die VM verursacht wieder.

(Auch VM-Groesse genannt)

5.3.3.6 def vm.VM.getmac (*self*)

Gibt die Macadresse der VM zurueck.

5.3.3.7 def vm.VM.getos (*self*)

Gibt das aktuelle Betriebssystem zurueck.

5.3.3.8 def vm.VM.getperiod (*self*)

Gibt die Periode der VM zurueck.

5.3.3.9 def vm.VM.getredundancy (*self*)

Gibt die Redundanz der VM zurueck (M, MB ode MBN)

5.3.3.10 def vm.VM.getstate (*self*)

Gibt den Status der VM, als M oder Backup oder Node, zurueck.

5.3.3.11 def vm.VM.gettaskname (*self*)

Gibt den Namen der VM zurueck.

5.3.3.12 def vm.VM.getwcet (*self*)

Gibt die WCET der Vm zurueck.

5.3.3.13 def vm.VM.setcreatedon (*self, newcreatedon*)

Setzt einen Erstellungszeitpunkt fuer die VM.

5.3.3.14 def vm.VM.setimgpath (*self*, *newimgpath*)

Setzt einen neuen Pfad fuer die image der VM.

5.3.3.15 def vm.VM.setip (*self*, *newip*)

Setzt eine neue IP fuer die VM.

5.3.3.16 def vm.VM.setmac (*self*, *newmac*)

Setzt eine neue Macadresse fuer die VM.

5.3.3.17 def vm.VM.setos (*self*, *newos*)

Setzt den Betriebssystemtyp.

5.3.3.18 def vm.VM.setstate (*self*, *newstate*)

Gibt der VM einen neuen Status.

Parameter

newstate	Der neue Status (Master, Backup oder Node)

5.3.4 Dokumentation der Datenelemente

5.3.4.1 vm.VM.createdon

5.3.4.2 vm.VM.ident

5.3.4.3 vm.VM.imgpath

5.3.4.4 vm.VM.ip

5.3.4.5 vm.VM.mac

5.3.4.6 vm.VM.operating

5.3.4.7 vm.VM.period

5.3.4.8 vm.VM.redundancy

5.3.4.9 vm.VM.state

5.3.4.10 vm.VM.taskname

5.3.4.11 vm.VM.wcet

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Datei:

- vm.py

Kapitel 6

Datei-Dokumentation

6.1 createxml.py-Dateireferenz

Namensbereiche

- createxml

Constant Groups

- createxml

Funktionen

- def createxml.create_heap_dump

  Erzeugt einen Heapdump.

- def createxml.createreply

  Erstellen einer reply.xml.

6.2 host.py-Dateireferenz

Klassen

- class host.Host

  Host Objekte entahlten alle Informationen ueber die Hosts und eine Liste von VMs.

Namensbereiche

- host

Constant Groups

- host

6.3 scanadapter.py-Dateireferenz

Namensbereiche

- scanadapter

Constant Groups

- scanadapter

Funktionen

- def scanadapter.stop_host
  Funktion sucht zu das passende Host Objekt und uebergibt es dem scheduler.
- def scanadapter.stop_all_hosts
  Dummy Funktion, macht nichts.
- def scanadapter.move_task_xml
- def scanadapter.start_task
  Staret einen Task und verschiebt die task.xml nach refused oder running.
- def scanadapter.stop_task
  Beendet einen Task.
- def scanadapter.saveobject
  Speichert ein Objekt.
- def scanadapter.loadobject
  Laedt ein Objekt.
- def scanadapter.addnewhost
  Fuegt dem Heap einen Host hinzu.
- def scanadapter.startup
  Initialisieren vom Heap, Replay und logging.

Variablen

- string scanadapter.NEUEZEILE = '\n '
  Trennzeichen von Zeilen fuer die reply.xml.
- int scanadapter.MASTER = 0
  State von dem Task.
- int scanadapter.BACKUP = 1
- int scanadapter.NODE = 2
- int scanadapter.MBN = 3
  Redundanz mit der die VM gestartet werden soll.
- int scanadapter.MB = 2
- int scanadapter.M = 1
- int scanadapter.RUNNING = 0
- int scanadapter.PAUSED = 1
- string scanadapter.TRENNZEICHEN = ';'

6.4 scheduler.py-Dateireferenz

Namensbereiche

- scheduler

Constant Groups

- scheduler

Funktionen

- def scheduler.printd
  Erzeugt ein feingranulares Logging.
- def scheduler.test_addfaultnumber
  Im Testmodus werden anstatt Strings Ints gespeichert.
- def scheduler.test_addfault
  Falls ein Fehler aufgetreten ist, wird dieser hier gespeichert.
- def scheduler.addmessage
  Die Nachricht wird nach dem Starten als reply.xml ausgegeben.
- def scheduler.addfault
  Falls ein Fehler aufgetreten ist, wird dieser hier gespeichert.
- def scheduler.gethosttovm
  Sucht zu einem VM Objekt das passendene Hostobjekt aus dem Heap.
- def scheduler.gethost
  Sucht zu einem String den Host.
- def scheduler.removehost
  Loescht ein Host aus dem heap.
- def scheduler.getglobalvmlist
  Gibt eine Liste mit allen VM Objekten auf allen Hosts zurueck.
- def scheduler.isthereamaster
  Gibt true zurueck, falls eine VM mit dem gleichen Namen existiert.
- def scheduler.isthereabackup
  Gibt true zurueck, falls eine VM mit dem gleichen Namen existiert.
- def scheduler.isthereanode
  Gibt true zurueck, falls eine VM mit dem gleichen Namen existiert.
- def scheduler.isvmwithstateonhost
  Gibt zurueck, ob sich eine VM mit dem State und dem Namen auf dem Host befindet.
- def scheduler.isvmonhost
  Gibt zurueck, ob sich eine VM mit dem Namen auf dem Host befindet.
- def scheduler.getmasterhosttovm
  Es wird der Host zurueckgegeben, auf dem sich die MasterVM zu der uebergegeben VM befindet.
- def scheduler.getvmnumberonhost
  Gibt den Index einer VM auf einem Host zurueck.
- def scheduler.getcorenumbertovm
  Gibt den Kern zurueck, auf dem sich die VM befindet.
- def scheduler.delvmfromhost
  Loescht eine VM aus der internen Datenstruktur.
- def scheduler.delvm
  Loescht eine VM vom Host und aus der internen Datenstruktur.
- def scheduler.delnodevm
  Loescht ein VM Objekt mit dem State NODE.
- def scheduler.checkstate
  In dieser Funktion werden fehlende Redundanzen hergestellt.
- def scheduler.migratevm
  Mirgiert eine VM.
- def scheduler.preparemigratevm

Diese Funktion bereitet das mirgieren von einer VM vor.
- def scheduler.startnewmastervm

  Bereitet das Starten einer neuen Mastervm vor.
- def scheduler.transmitstartmaster

  Startet eine VM auf einem Host.
- def scheduler.startnewnodevm

  Node Objekt wird der Datenstruktur hinzugefuegt.
- def scheduler.startnewbackupvm

  Bereitet das Starten einer Backup VM vor.
- def scheduler.transmitstartbackup

  Uebertreagt das Kommando zum Starten einer Backup VM.
- def scheduler.schedulenewvm

  Einsprungprunkt zum Starten einer neuen VM.
- def scheduler.transmitstopvm

  Uebertraegt das Kommando zum Beenden einer VM.
- def scheduler.stopvm

  Beendet eine VM.
- def scheduler.preparehostdown

  Wenn ein Host nicht mehr erreichbar ist, wird hier das Hostobjekt gesucht und weitergeleitet.
- def scheduler.prepearevmtorestart

  Bereitet das Neustarten einer VM vor.
- def scheduler.preparevmchangedstate

  Falls eine VM ihren Status wechselt, sucht diese Funktion die Objekte und beendet die Master VM.
- def scheduler.vmdown

  Diese Funktion startet eine abgestuerzte VM neu.
- def scheduler.hostdown

  Diese Funktion startet alle VMs eines Hosts neu.
- def scheduler.shutdown

  Diese Funktion bereitet einen Host zum Herunterfahren vor.
- def scheduler.checkeverything

  Es wird regelmaessig ueberprueft, ob der Heap mit der Realitaet uebereinstimmt.
- def scheduler.print_heap
- def scheduler.copy_heap

  Gibt eine Kopie des aktuellen Heaps zurueck.
- def scheduler.heap_tripel

  Gibt eine, um Hostnamen und Coreindex erweiterte, Liste des aktuellen Heaps zurueck.
- def scheduler.heap_quadrupel

  Gibt eine Liste des aktuellen Heaps zurueck.
- def scheduler.sort_heap_quadrupel

  Gibt eine nach Core-Auslastung sortierte Heap-Liste zurueck.
- def scheduler.all_list_print
- def scheduler.list_print
- def scheduler.mkr

  Das MaKeRoom wird vor dem Starten einer neuen VM ausgefuehrt.
- def scheduler.find_place

  Wird von mkr aufgerufen.
- def scheduler.place_for_newvm

  Prueft auf einem bestimmten Core, ob die neue VM draufpasst.
- def scheduler.coreload_refresh
- def scheduler.vm_to_move

  Sucht eine Stelle , wohin die angegebene VM verschoben werden kann.

- def scheduler.vm_move

  In der angegebenen Heap-Liste wird die VM vom Quell-Host/Core zum Ziel-Host/Core verschoben.
- def scheduler.new_vm_redundancy_check

  Prueft ob auf diesem Host bereits Platz fuer die neue VM reserviert/freigemacht wurde.
- def scheduler.found_check

  Prueft, ob Core bereits reserviert wurde.
- def scheduler.redundancy_check

  Prueft, ob Redundanz der zu verschiebenden VM bereits auf diesem Host ist.
- def scheduler.load_check

  Prueft, ob die neue VM auf Core passt.
- def scheduler.coreload_update

  Berechnen die Coreauslastung neu.
- def scheduler.vm_migrate

  Migriert Schritt fuer Schritt die Migrate-Liste ab.
- def scheduler.heap_format

  Bringt eine lange-erweiterte Heap-Liste in die normale Heapform.
- def scheduler.heap_update

  Erhaelt eine erweiterte-lange-Heapliste und speichert sie als Heap.
- def scheduler.heap_save

  Speichert eine Heap-Liste als Heap ab.
- def scheduler.moveable_check

  Prueft, ob diese VM migrierbar ist.
- def scheduler.reschedule

  Rebalanciert den Heap, in dem die Core-Auslastungen ausgeglichen werden.

Variablen

- string scheduler.NEUEZEILE = '\n '

  Trennzeichen von Zeilen fuer die reply.xml.
- string scheduler.TRENNZEICHEN = ';'

  Trennzeichen fuer die verschiedenen Fehlercodes.
- int scheduler.MASTER = 0

  State von dem Task.
- int scheduler.BACKUP = 1
- int scheduler.NODE = 2
- int scheduler.MBN = 3

  Redundanz mit der die VM gestartet werden soll.
- int scheduler.MB = 2
- int scheduler.M = 1
- scheduler.globtesting = False
- scheduler.globlogging = True
- int scheduler.maxload_of_core = 1
- tuple scheduler.all_list = list()
- tuple scheduler.migrate_list = list()
- tuple scheduler.found_places_list = list()
- tuple scheduler.temp_migrate_list = list()

6.5 tempreply.py-Dateireferenz

Klassen

- class tempreply.Tempreply

  Wird zum generieren der reply.xml benoetigt.

Namensbereiche

- tempreply

Constant Groups

- tempreply

6.6 testen.py-Dateireferenz

Namensbereiche

- testen

Constant Groups

- testen

Funktionen

- def testen.start_task_test

  Normaler Funktionstest der LMU von verschiedenen Diensten.
- def testen.start_pause_task_test

  Funktionstest der LMU.
- def testen.start_stop_task_test

  Test der LMU, bei dem der Dienst zuerst gestartet wird, dann gestoppt wird.
- def testen.start2times_task_test

  Es wird zwei mal der gleiche Task gestartet.
- def testen.wcet_longer_period_test

  Bei den Tasks ist WCET immer groesser als Periode.
- def testen.configuration_test

  Die Host werden unterschiedlich Konfiguriert.
- def testen.task_xml_error_test

  Es wird eine Fehlerhafte task.xml generiert.
- def testen.action_xml_error_test

  Es wird eine fehlerhafte action.xml generiert.
- def testen.belastung_test

  beliebig viele Tasks werden in einem Zeitraum hintereinander gestartet.
- def testen.create_directory

  Hilfsfunktionen #.
- def testen.move_to_directory

  Am Ende des Tests werden alle task.xml, action.xml, reply.xml und hosts.xml, die im Test generiert wurden, in den Testordner verschoben.

- def testen.move_actionxml_and_replyxml

  Zwischen dem Test werden action.xml und reply.xml zum Testordner verschoben.
- def testen.create_action_xml

  Erstellung von einer action.xml.
- def testen.create_action_xml_1
- def testen.create_action_xml_2
- def testen.create_task_xml

  Erstellung von einer richtigen task.xml.
- def testen.create_task_xml_0

  Erstellung von einer fehlerhaten task.xml.
- def testen.create_task_xml_1

  Erstellung von einer fehlerhaten task.xml.
- def testen.create_task_xml_2

  Erstellung von einer fehlerhaten task.xml.
- def testen.create_random_hosts_xml

  Erstellung von einer hosts.xml, dass beliebig viele Hosts konfiguriert werden.
- def testen.create_one_host_xml

  Erstellung von einer hosts.xml, dass nur ein Host verfuegbar ist.
- def testen.create_two_hosts_xml

  Erstellung von einer hosts.xml, dass zwei Hosts verfuegbar sind.
- def testen.create_three_hosts_xml

  Erstellung von einer hosts.xml, dass drei Hosts verfuegbar sind.
- def testen.create_four_hosts_xml

  Erstellung von einer hosts.xml, dass vier Hosts verfuegbar sind.
- def testen.get_reply

  reply.xml Abruf 5 mal, jeweils 5 Sekunden lang
- def testen.check_reply

  Checkt ob reply.xml in lmu_outbox existiert.
- def testen.get_reply_message

  Fehlercode auffangen.
- def testen.message_list

  Fehlerliste erstellen.
- def testen.failure_check

  Fehlervergleich.
- def testen.error_check

  Ergebnis des Tests wird vergleichen, Es wird ueberprueft, ob es unsere Erwartungen erfuellen konnte.

Variablen

- string testen.pfad = "/home/nas/lmu-test/"

  Testfaelle #.

6.7 transmitcommand.py-Dateireferenz

Namensbereiche

- transmitcommand

Constant Groups

- transmitcommand

Funktionen

- def transmitcommand.connect_to_socket
  Attempts to establish a connection to a remote TCP-socket.
- def transmitcommand.transmit_message
  Pickles the passed command list and attempts to transmit it to the remote LMU interface.
- def transmitcommand.transmit_state
  Pickles the passed command list and attempts to transmit it to the remote LMU interface.
- def transmitcommand.close_socket
  Attempts to close the global TCP-Socket mySocket.
- def transmitcommand.dom_create_cfg
  Creates the configuration file for the new domain.
- def transmitcommand.dom_create
  Creates a new domain on the remote host.
- def transmitcommand.dom_destroy
  Destroys a running domain on the remote host.
- def transmitcommand.dom_migrate
  Migrates a running domain to another host.
- def transmitcommand.dom_remus
  Creates a remus replication of the specified domain with the specified replication interval (default: 500).
- def transmitcommand.dom_set_period_slice
  (SEDF only) Adapts period and slice for the specified domain name.
- def transmitcommand.dom_vcpu_pin
  Pins the specified domains VCPU to the PCPU.
- def transmitcommand.dom_notify_incoming_migration
  Notifies the remote host about an incoming VM migration.
- def transmitcommand.dom_notify_incoming_remus
  Notifies the remote host about an incoming remus replication.
- def transmitcommand.dom_notify_intended_failover
  Notifies the remote host about an intended remus failover.
- def transmitcommand.host_check_sanity
  Check sanity of remote host.
- def transmitcommand.host_get_state
  Check sanity of remote host.

Variablen

- transmitcommand.mySocket = None
- int transmitcommand.socketTimeout = 5
- int transmitcommand.extendedSocketTimeout = 20
- tuple transmitcommand.parser = OptionParser()

6.8 verzeichnisscanner.py-Dateireferenz

Namensbereiche

- verzeichnisscanner

Constant Groups

- verzeichnisscanner

Funktionen

- def verzeichnisscanner.check_xml_file
  Es wird ueberprueft, ob eine action_∗.xml vorhanden ist.
- def verzeichnisscanner.load_taskxml
  Es wird eine task.xml geladen und ein VM Objekt zurueckgegeben.
- def verzeichnisscanner.load_xml_file
  Diese Funktion liest eine action.xml aus und startet die jeweilige Aktion.
- def verzeichnisscanner.ishostinheap
  Ueberprueft, ob sich der Host bereits im Heap befindet.
- def verzeichnisscanner.is_host_in_xml
  Ueberprueft, ob sich ein Hostobjekt in der host.xml befindet.
- def verzeichnisscanner.load_hosts_xml_file_for_check
  Gib den Inhalt der host.xml als Liste mit Hostobjekten zurueck.
- def verzeichnisscanner.load_hosts_xml_file
  Laedt die hosts.xml und ueberprueft ob die Hosts bereits im Heap sind.
- def verzeichnisscanner.getIP
  Liefert die aktuelle IP zurueck.
- def verzeichnisscanner.execCommand
  Fuehrt ein Kommando als Subprozess aus.
- def verzeichnisscanner.start_webserver
  Startet den Webserver.
- def verzeichnisscanner.start_rep_lmu
  Repliziert die LMU.
- def verzeichnisscanner.initPipe
  Initialisiert die Pipe.
- def verzeichnisscanner.readpipe
  Liest aus der Pipe.
- def verzeichnisscanner.checkrunningfiles
  Verschiebt task.xml und action.xml von runnig nach pending.
- def verzeichnisscanner.getsysteminfo
  Empfängt von allen hosts im Heap die aktuell laufenden VMs und überprüft, ob die aktuelle Konfiguration mit der aus dem Heap übereinstimmt.
- def verzeichnisscanner.runme
  Endlosschleife.

Variablen

- string verzeichnisscanner.NEUEZEILE = '\n '
  Trennzeichen von den Zeilen fuer die reply.xml.
- int verzeichnisscanner.MASTER = 0
  Status eines Tasks.
- int verzeichnisscanner.BACKUP = 1
- int verzeichnisscanner.NODE = 2
- int verzeichnisscanner.MBN = 3
  Redundanz mit der die VM gestartet werden soll.
- int verzeichnisscanner.MB = 2

- int verzeichnisscanner.M = 1
- verzeichnisscanner.testing = False
- string verzeichnisscanner.globimgpath = ”
- string verzeichnisscanner.pipe_name = ’/tmp/pipe_globmon-lmu’
- verzeichnisscanner.pipein = None
- string verzeichnisscanner.lokalerpfad = ’/home/nas/’
- int verzeichnisscanner.lastchange = 0

6.9 vm.py-Dateireferenz

Klassen

- class vm.VM

    VM Objekte enthalten alle Informationen ueber die VM.

Namensbereiche

- vm

Constant Groups

- vm

Index

__init__
    host::Host, 48
    tempreply::Tempreply, 52
    vm::VM, 54
__lt__
    host::Host, 48

action_xml_error_test
    testen, 31
addcoreload
    host::Host, 48
addfault
    scheduler, 13
addmessage
    scheduler, 13
addnewhost
    scanadapter, 8
addvm
    host::Host, 48
all_list
    scheduler, 29
all_list_print
    scheduler, 13

BACKUP
    scanadapter, 10
    scheduler, 29
    verzeichnisscanner, 44
belastung_test
    testen, 31

check_reply
    testen, 31
check_xml_file
    verzeichnisscanner, 42
checkeverything
    scheduler, 13
checkrunningfiles
    verzeichnisscanner, 42
checkstate
    scheduler, 14
close_socket
    transmitcommand, 38
configuration_test
    testen, 31
connect_to_socket
    transmitcommand, 38
copy_heap
    scheduler, 14
coreload
    host::Host, 51
coreload_refresh
    scheduler, 14
coreload_update
    scheduler, 14
create_action_xml
    testen, 31
create_action_xml_1
    testen, 32
create_action_xml_2
    testen, 32
create_directory
    testen, 32
create_four_hosts_xml
    testen, 32
create_heap_dump
    createxml, 7
create_one_host_xml
    testen, 32
create_random_hosts_xml
    testen, 32
create_task_xml
    testen, 32
create_task_xml_0
    testen, 33
create_task_xml_1
    testen, 33
create_task_xml_2
    testen, 33
create_three_hosts_xml
    testen, 33
create_two_hosts_xml
    testen, 33
createdon
    vm::VM, 56
createreply
    createxml, 7
createxml, 7
    create_heap_dump, 7
    createreply, 7
createxml.py, 57

delnodevm
    scheduler, 14
delvm
    scheduler, 14
delvmfromhost
    scheduler, 15
dom_create
    transmitcommand, 38

dom_create_cfg
transmitcommand, 38
dom_destroy
transmitcommand, 38
dom_migrate
transmitcommand, 39
dom_notify_incoming_migration
transmitcommand, 39
dom_notify_incoming_remus
transmitcommand, 39
dom_notify_intended_failover
transmitcommand, 39
dom_remus
transmitcommand, 39
dom_set_period_slice
transmitcommand, 40
dom_vcpu_pin
transmitcommand, 40

error_check
testen, 33
execCommand
verzeichnisscanner, 42
extendedSocketTimeout
transmitcommand, 41

failure_check
testen, 35
fault
tempreply::Tempreply, 53
find_place
scheduler, 15
found_check
scheduler, 15
found_places_list
scheduler, 29

get_reply
testen, 35
get_reply_message
testen, 35
getIP
verzeichnisscanner, 42
getallvms
host::Host, 49
getcoreload
host::Host, 49
getcorenumbertovm
scheduler, 15
getcreatedon
vm::VM, 54
getfault
tempreply::Tempreply, 52
getglobalvmlist
scheduler, 15
gethost
scheduler, 17
gethosttovm
scheduler, 17
getid
vm::VM, 54
getimgpath
vm::VM, 55
getip
host::Host, 49
vm::VM, 55
getload
vm::VM, 55
getmac
vm::VM, 55
getmasterhosttovm
scheduler, 17
getmaxcoreload
host::Host, 49
getmaxload
host::Host, 49
getmessage
tempreply::Tempreply, 52
getmincoreload
host::Host, 49
getminloadofcore
host::Host, 49
getname
host::Host, 49
getnumberofcores
host::Host, 49
getos
vm::VM, 55
getperiod
vm::VM, 55
getposmaxcoreload
host::Host, 49
getposmincoreload
host::Host, 50
getredundancy
vm::VM, 55
getstate
vm::VM, 55
getsysteminfo
verzeichnisscanner, 42
gettaskname
vm::VM, 55
getvm
host::Host, 50
getvmlist
host::Host, 50
getvmlistcore
host::Host, 50
getvmlistofcore
host::Host, 50
getvmlistsort
host::Host, 50
getvmnumberonhost
scheduler, 17
getwcet
vm::VM, 55
globimgpath

verzeichnisscanner, 44
globlogging
scheduler, 29
globtesting
scheduler, 29

heap_format
scheduler, 18
heap_quadrupel
scheduler, 18
heap_save
scheduler, 18
heap_tripel
scheduler, 18
heap_update
scheduler, 18
host, 7
host.Host, 47
host.py, 57
host::Host
__init__, 48
__lt__, 48
addcoreload, 48
addvm, 48
coreload, 51
getallvms, 49
getcoreload, 49
getip, 49
getmaxcoreload, 49
getmaxload, 49
getmincoreload, 49
getminloadofcore, 49
getname, 49
getnumberofcores, 49
getposmaxcoreload, 49
getposmincoreload, 50
getvm, 50
getvmlist, 50
getvmlistcore, 50
getvmlistofcore, 50
getvmlistsort, 50
ip, 51
maxload, 51
name, 51
numberofcores, 51
removecoreload, 51
removevm, 51
removevmlist, 51
setcoreload, 51
setvmstate, 51
vmlist, 51
host_check_sanity
transmitcommand, 40
host_get_state
transmitcommand, 40
hostdown
scheduler, 19

ident
vm::VM, 56
imgpath
vm::VM, 56
initPipe
verzeichnisscanner, 43
ip
host::Host, 51
vm::VM, 56
is_host_in_xml
verzeichnisscanner, 43
ishostinheap
verzeichnisscanner, 43
isthereabackup
scheduler, 19
isthereamaster
scheduler, 19
isthereanode
scheduler, 19
isvmonhost
scheduler, 19
isvmwithstateonhost
scheduler, 20

lastchange
verzeichnisscanner, 44
list_print
scheduler, 20
load_check
scheduler, 20
load_hosts_xml_file
verzeichnisscanner, 43
load_hosts_xml_file_for_check
verzeichnisscanner, 43
load_taskxml
verzeichnisscanner, 43
load_xml_file
verzeichnisscanner, 44
loadobject
scanadapter, 8
lokalerpfad
verzeichnisscanner, 44

M
scanadapter, 10
scheduler, 29
verzeichnisscanner, 44
MASTER
scanadapter, 10
scheduler, 29
verzeichnisscanner, 44
MB
scanadapter, 10
scheduler, 29
verzeichnisscanner, 44
MBN
scanadapter, 10
scheduler, 29
verzeichnisscanner, 45
mac

vm::VM, 56
maxload
host::Host, 51
maxload_of_core
scheduler, 29
message
tempreply::Tempreply, 53
message_list
testen, 35
migrate_list
scheduler, 29
migratevm
scheduler, 20
mkr
scheduler, 20
move_actionxml_and_replyxml
testen, 36
move_task_xml
scanadapter, 9
move_to_directory
testen, 36
moveable_check
scheduler, 22
mySocket
transmitcommand, 41

NEUEZEILE
scanadapter, 10
scheduler, 29
verzeichnisscanner, 45
NODE
scanadapter, 10
scheduler, 29
verzeichnisscanner, 45
name
host::Host, 51
new_vm_redundancy_check
scheduler, 22
numberofcores
host::Host, 51

operating
vm::VM, 56

PAUSED
scanadapter, 10
parser
transmitcommand, 41
period
vm::VM, 56
pfad
testen, 37
pipe_name
verzeichnisscanner, 45
pipein
verzeichnisscanner, 45
place_for_newvm
scheduler, 22
preparehostdown
scheduler, 22
preparemigratevm
scheduler, 24
preparevmchangedstate
scheduler, 24
prepearevmtorestart
scheduler, 24
print_heap
scheduler, 24
printd
scheduler, 24

RUNNING
scanadapter, 10
readpipe
verzeichnisscanner, 44
redundancy
vm::VM, 56
redundancy_check
scheduler, 24
removecoreload
host::Host, 51
removehost
scheduler, 25
removevm
host::Host, 51
removevmlist
host::Host, 51
reschedule
scheduler, 25
runme
verzeichnisscanner, 44

saveobject
scanadapter, 9
scanadapter, 8
addnewhost, 8
BACKUP, 10
loadobject, 8
M, 10
MASTER, 10
MB, 10
MBN, 10
move_task_xml, 9
NEUEZEILE, 10
NODE, 10
PAUSED, 10
RUNNING, 10
saveobject, 9
start_task, 9
startup, 9
stop_all_hosts, 9
stop_host, 9
stop_task, 9
TRENNZEICHEN, 10
scanadapter.py, 58
schedulenewvm
scheduler, 25
scheduler, 10

addfault, 13
addmessage, 13
all_list, 29
all_list_print, 13
BACKUP, 29
checkeverything, 13
checkstate, 14
copy_heap, 14
coreload_refresh, 14
coreload_update, 14
delnodevm, 14
delvm, 14
delvmfromhost, 15
find_place, 15
found_check, 15
found_places_list, 29
getcorenumbertovm, 15
getglobalvmlist, 15
gethost, 17
gethosttovm, 17
getmasterhosttovm, 17
getvmnumberonhost, 17
globlogging, 29
globtesting, 29
heap_format, 18
heap_quadrupel, 18
heap_save, 18
heap_tripel, 18
heap_update, 18
hostdown, 19
isthereabackup, 19
isthereamaster, 19
isthereanode, 19
isvmonhost, 19
isvmwithstateonhost, 20
list_print, 20
load_check, 20
M, 29
MASTER, 29
MB, 29
MBN, 29
maxload_of_core, 29
migrate_list, 29
migratevm, 20
mkr, 20
moveable_check, 22
NEUEZEILE, 29
NODE, 29
new_vm_redundancy_check, 22
place_for_newvm, 22
preparehostdown, 22
preparemigratevm, 24
preparevmchangedstate, 24
prepearevmtorestart, 24
print_heap, 24
printd, 24
redundancy_check, 24
removehost, 25
reschedule, 25
schedulenewvm, 25
shutdown, 25
sort_heap_quadrupel, 25
startnewbackupvm, 26
startnewmastervm, 26
startnewnodevm, 26
stopvm, 26
TRENNZEICHEN, 29
temp_migrate_list, 29
test_addfault, 26
test_addfaultnumber, 27
transmitstartbackup, 27
transmitstartmaster, 27
transmitstopvm, 27
vm_migrate, 27
vm_move, 28
vm_to_move, 28
vmdown, 28
scheduler.py, 58
setcoreload
host::Host, 51
setcreatedon
vm::VM, 55
setfault
tempreply::Tempreply, 52
setimgpath
vm::VM, 55
setip
vm::VM, 56
setmac
vm::VM, 56
setmessage
tempreply::Tempreply, 52
setos
vm::VM, 56
setstate
vm::VM, 56
setvmstate
host::Host, 51
shutdown
scheduler, 25
socketTimeout
transmitcommand, 41
sort_heap_quadrupel
scheduler, 25
start2times_task_test
testen, 36
start_pause_task_test
testen, 36
start_rep_lmu
verzeichnisscanner, 44
start_stop_task_test
testen, 36
start_task
scanadapter, 9
start_task_test
testen, 36

start_webserver
verzeichnisscanner, 44
startnewbackupvm
scheduler, 26
startnewmastervm
scheduler, 26
startnewnodevm
scheduler, 26
startup
scanadapter, 9
state
vm::VM, 56
stop_all_hosts
scanadapter, 9
stop_host
scanadapter, 9
stop_task
scanadapter, 9
stopvm
scheduler, 26

TRENNZEICHEN
scanadapter, 10
scheduler, 29
task_xml_error_test
testen, 36
taskname
vm::VM, 56
temp_migrate_list
scheduler, 29
tempreply, 29
tempreply.py, 62
tempreply.Tempreply, 52
tempreply::Tempreply
__init__, 52
fault, 53
getfault, 52
getmessage, 52
message, 53
setfault, 52
setmessage, 52
test_addfault
scheduler, 26
test_addfaultnumber
scheduler, 27
testen, 30
action_xml_error_test, 31
belastung_test, 31
check_reply, 31
configuration_test, 31
create_action_xml, 31
create_action_xml_1, 32
create_action_xml_2, 32
create_directory, 32
create_four_hosts_xml, 32
create_one_host_xml, 32
create_random_hosts_xml, 32
create_task_xml, 32
create_task_xml_0, 33
create_task_xml_1, 33
create_task_xml_2, 33
create_three_hosts_xml, 33
create_two_hosts_xml, 33
error_check, 33
failure_check, 35
get_reply, 35
get_reply_message, 35
message_list, 35
move_actionxml_and_replyxml, 36
move_to_directory, 36
pfad, 37
start2times_task_test, 36
start_pause_task_test, 36
start_stop_task_test, 36
start_task_test, 36
task_xml_error_test, 36
wcet_longer_period_test, 36
testen.py, 62
testing
verzeichnisscanner, 45
transmit_message
transmitcommand, 40
transmit_state
transmitcommand, 41
transmitcommand, 37
close_socket, 38
connect_to_socket, 38
dom_create, 38
dom_create_cfg, 38
dom_destroy, 38
dom_migrate, 39
dom_notify_incoming_migration, 39
dom_notify_incoming_remus, 39
dom_notify_intended_failover, 39
dom_remus, 39
dom_set_period_slice, 40
dom_vcpu_pin, 40
extendedSocketTimeout, 41
host_check_sanity, 40
host_get_state, 40
mySocket, 41
parser, 41
socketTimeout, 41
transmit_message, 40
transmit_state, 41
transmitcommand.py, 63
transmitstartbackup
scheduler, 27
transmitstartmaster
scheduler, 27
transmitstopvm
scheduler, 27

verzeichnisscanner, 41
BACKUP, 44
check_xml_file, 42
checkrunningfiles, 42
execCommand, 42

getIP, 42
getsysteminfo, 42
globimgpath, 44
initPipe, 43
is_host_in_xml, 43
ishostinheap, 43
lastchange, 44
load_hosts_xml_file, 43
load_hosts_xml_file_for_check, 43
load_taskxml, 43
load_xml_file, 44
lokalerpfad, 44
M, 44
MASTER, 44
MB, 44
MBN, 45
NEUEZEILE, 45
NODE, 45
pipe_name, 45
pipein, 45
readpipe, 44
runme, 44
start_rep_lmu, 44
start_webserver, 44
testing, 45
verzeichnisscanner.py, 64
vm, 45
vm.py, 66
vm.VM, 53
vm::VM
__init__, 54
createdon, 56
getcreatedon, 54
getid, 54
getimgpath, 55
getip, 55
getload, 55
getmac, 55
getos, 55
getperiod, 55
getredundancy, 55
getstate, 55
gettaskname, 55
getwcet, 55
ident, 56
imgpath, 56
ip, 56
mac, 56
operating, 56
period, 56
redundancy, 56
setcreatedon, 55
setimgpath, 55
setip, 56
setmac, 56
setos, 56
setstate, 56
state, 56
taskname, 56
wcet, 56
vm_migrate
scheduler, 27
vm_move
scheduler, 28
vm_to_move
scheduler, 28
vmdown
scheduler, 28
vmlist
host::Host, 51

wcet
vm::VM, 56
wcet_longer_period_test
testen, 36

B. Liste aller Fehlerfälle

Komponente	Fehler	Erwartetes Ergebnis	Test erfolgreich?/Ergebnis
Discovery	Discovery startet nicht	System startet nicht, Fehlermeldung	Konnte nicht nachgestellt werden
Discovery	Fehlende Host-XML	System startet nicht, entsprechende Fehlermeldung	OK
Discovery	Fehlerhafte Host-XML	System startet nicht, entsprechende Fehlermeldung	OK
Discovery	Discovery entdeckt keine Rechner	System startet nicht, entsprechende Fehlermeldung	Konnte nicht nachgestellt werden
LMU	LMU bootet nicht	Discovery hängt sich auf, System läuft nicht	OK
LMU	LMU schlägt bei Erzeugung der eigenen Redundanz fehl	Fehlermeldung im LMU-Log	KEINE Meldung im LMU-Log!
LMU	Nur ein Host verfügbar	LMU versucht eigene Redundanz herzustellen, schlägt fehl und produziert Eintrag im LMU-Log, startet diese sobald neuer Host hinzukommt	Keinerlei Einträge im LMU Log
LMU	(Action.xml oder Task.xml fehlerhaft/nicht vorhanden)	Aktion wird nicht ausgeführt, Reply.xml zurückgegeben	OK
LMU	Nicht genügend Ressourcen für Start eines weiteren Dienstes verfügbar	Dienst wird nicht gestartet, in abgelehnt-Ordner verschoben, entsprechende reply zurückgeben	OK
LMU	(Name des zu startenden Dienstes bereits vergeben)	Das Formular zum Dienst starten kann nicht abgeschickt werden und es zeigt den fehlerhaften Eintrag rot an	OK
LMU	Befehl zum Starten einer VM wurde nicht korrekt übertragen/ausgeführt	Antwort über Fehler wird erhalten, alle schon gestarteten VMs werden beendet, in-abgelehnt-Ordner verschoben, entsprechende reply.xml zurückgegeben	OK
LMU	Interne Datenstruktur enthält VMs, die in der Realität nicht laufen (abgestürzt)	Falls aufgehangen, aber noch in VM-Liste: Keine Reaktion. Sonst Mitteilung des Monitorings über fehlende VM, Löschen aus Datenstruktur und Neustart der VM/Redundanz	Info durch das Monitoring
Webserver	bootet nicht	GUI und sämtliche GUI-Funktionalität nicht erreichbar	GUI und sämtliche GUI-Funktionalität nicht erreichbar
Webserver	Erhält Logs im falschen Format	Logeinträge, die im falschen Format sind, werden nicht	OK

		angezeigt	
Webserver	Erhält Reply im falschen Format	Fehlermeldung in Form einer korrekten Reply, die dem Benutzer angezeigt wird	OK. Fehlermeldung in der GUI wird angezeigt
Webserver	Erhält keine/fehlerhafte Host-XML	Benutzer bekommt Fehlermeldung zu fehlender/fehlerhafter Host.xml und ein leeres Hostformular angezeigt	Leeres Hostformular: OK
Webserver	Falsche/Fehlende Eingaben beim Dienststart	Keine action.xml und task.xml Generierung, Benutzer sieht Fehler	OK
Webserver	Falsche/Fehlende Eingaben bei der Hostkonfiguration	Keine host.xml wird generiert, Benutzer sieht Fehler	OK
Webserver	Timeout beim Upload eines Isos	Fehlermeldung in der GUI	Konnte nicht nachgestellt werden
Alle	Skript fehlt	Absturz des Systems	Konnte nicht nachgestellt werden
Alle	Skript Semantik Fehler	Ablauf des Systems nicht mehr nachvollziehbar	Konnte nicht nachgestellt werden
Alle	Kompletter Stromausfall und Neustart der Rechne	Discovery startet sich automatisch und startet das System neu, wobei vorher laufende Dienst neugestartet werden, allerdings ohne Speicherung der Zustände	OK
Alle	Falsche oder keine IP durch DHCP	Kann nicht nachgestellt werden	Kann nicht nachgestellt werden
Hardware	Ausfall des Sensornetzes	Wird momentan nicht geloggt	Indirekt erkannt
Hardware	Paketverlust/Verzögerung auf dem Sensornetz	Monitoring loggt Paketverzögerungs-Statistiken, Paketverlust wird nicht erkannt	Muss angepasst werden. Paketverlust-Erkennung und Logging
Hardware	Ausfall des Internode Netzes	Wird momentan nicht geloggt	Indirekt erkannt, aber nicht geloggt
Hardware	Paketverlust/Verzögerung auf dem Internode Netz	Monitoring loggt Paketverzögerungs-Statistiken, Paketverlust wird nicht erkannt	Todo: wird nicht geloggt und erkannt
Hardware	Host fällt aus	Hostausfall wird erkannt und dem Benutzer gemeldet, Master, Backups und Nodes werden auf andere Hosts verlagert, falls Platz ist	OK
NAS	NAS nicht erreichbar	LMU und Webserver starten erst wenn NAS erreichbar ist. Die versuchen es solange, bis es wieder erreichbar ist. Muss manuell gefixt werden	ToDo! Keine Daten im Frontend sichtbar. Keine Fehlermeldung
VM	VM konnte nicht als Master/Backup gestartet werden	Auf anderen Hosts probieren und ansonsten muss der Dienst neugestartet werden	OK
VM	Master-VM stürzt ab	M-Dienst: Monitoring erkennt Ausfall, LMU veranlasst Neustart. MB-Dienst: Backup wird zum Master und übernimmt, neues Backup wird gestartet. MBN-Dienst: Backup übernimmt Masterfunktion, Node wird gelöscht, Platz für neuen Backup	OK

		wird gesucht, Backup gestartet, Node neu gestartet	
VM	Backup-VM stürzt ab	MB-Dienst: Neuer Backup wird gestartet. MBN-Dienst: Node wird gelöscht, Platz für neuen Backup wird gesucht, Backup gestartet, Node neu gestartet	OK bis auf Backups auf Mitte-Links
VM	VM ist nach Ablauf seiner erlaubten Rechenzeit nicht fertig	Wird überprüft und Eintrag im Error-log der Applikation erstellt und Darstellung über GUI erfolgt	OK
VM	Überschreitung der Deadline	Wird überprüft und Eintrag im Error_log der Applikation erstellt, Darstellung über GUI	OK
VM	VM überlastet CPU	Die Task XML von von pending auf refused verschoben	OK
VM	Config fehlt	1. Systemdienst(LMU oder Webserver) können nicht gestartet werden bevor nicht manuelle config Datei erstellt worden ist. 2. VMs die über die GUI gestartet werden sollen, können ohne Config Datei nicht gestartet werden (im Normalfall automatische Generierung durch LMU bzw. LMU-Interface)	OK
VM	Config fehlerhaft	Kann nicht geschehen, da sie automatisch generiert werden. Falls doch der Fall auftreten sollte, kann der Dienst nicht gestartet werden	Kann nicht nachgestellt werden
VM	VM konnte nicht korrekt beendet werden	Fehlermeldung der LMU-Interface an LMU. Muss manuell beendet werden.	Wurde nicht getestet
VM	Stromausfall beim Migrieren	Wenn auf dem Ziel Host der Strom ausfällt, dann wird die Migration abgebrochen und die VM läuft auf dem Quellhost weiter. Falls sie auf dem abbricht, dann ist die Migration fehlgeschlagen	Wurde nicht getestet
VM	Timeout beim Migrieren	Auf anderen Hosts probieren und ansonsten muss der Dienst neugestartet werden	OK
Verzeichnisscanner	Startet nicht	Muss manuell neugestartet werden	Konnte nicht nachgestellt werden
Verzeichnisscanner	Stürzt ab	Stürzt nur bei Programmfehler ab. Muss manuell behoben werden.	OK
Monitoring	Lok. Monitoring erreicht glob. Monitoring nicht	Versucht das glob. Monitoring alle 10 Sekunden zu erreichen, bis es gelingt	OK, aber nur, wenn glob Monitoring wieder ansprechbar ist
Monitoring	Keine Logeinträge	Kein Monitoring über den Webserver sichtbar	OK. Keine Einträge sichtbar; keine Fehlermeldung
GrabVirq	Abgestürzt	Redundanz wird innerhalb von 5 Sekunden wieder erstellt	OK

C. Liste aller Host-Logging-Daten

Host-Logging

Kind	Type	Data
Static	HOST_NUMBER_CORES	INT_NUM
Static	HOST_IP_ADDRESS	TXT_HOST_IP_ADDRESS
Static	HOST_MEM_TOTAL	INT_BYTES
Static	HOST_IP_ADDRESS	TXT_HOST_IP_ADDRESS
Static	HOST_MEM_TOTAL	INT_BYTES
Static	HOST_NAME	TXT_HOST_NAME
Cyclic	HOST_MEM_UTILIZATION	INT_CURRENTLY_USED_BYTES|INT_AVAILABLE_BYTES
Cyclic	HOST_CPU_TOTAL_UTILIZATION	FLOAT_PERCENT
Cyclic	HOST_CPU_CORE_0_UTILIZATION	FLOAT_PERCENT
Cyclic	HOST_CPU_CORE_1_UTILIZATION	FLOAT_PERCENT
Cyclic	………	FLOAT_PERCENT
Cyclic	HOST_CPU_CORE_N-1_UTILIZATION	FLOAT_PERCENT
Cyclic	HOST_SYS_LOAD	TXT
Cyclic	HOST_NET_INTERNODE_THROUGHPUT	FLOAT_MIN_BYTESPSEC|FLOAT_AVG_BYTESPSEC|FLOAT_MAX_BYTESPSEC
Cyclic	HOST_NET_SENSORS_THROUGHPUT	FLOAT_MIN_BYTESPSEC|FLOAT_AVG_BYTESPSEC|FLOAT_MAX_BYTESPSEC
Cyclic	HOST_NET_INTERNODE_LATENCY	FLOAT_MIN_MS|FLOAT_AVG_MS|FLOAT_MAX_MS
Cyclic	HOST_NET_SENSORS_LATENCY	FLOAT_MIN_MS|FLOAT_AVG_MS|FLOAT_MAX_MS
Cyclic	HOST_VMs_PAUSED	TXT_VM_NAMES
Event	HOST_EVT_VM_START	TXT_VM_NAME|TXT_VM_ROLE ∈ {"MASTER", "BACKUP"}
Event	HOST_EVT_VM_STOP	TXT_VM_NAME|TXT_VM_ROLE ∈ {"MASTER", "BACKUP"}
Event	HOST_EVT_VM_SUSPEND	TXT_VM_NAME|TXT_VM_ROLE ∈ {"MASTER", "BACKUP"}
Event	HOST_EVT_VM_MIGRATE	TXT_VM_NAME|TXT_VM_ROLE ∈ {"MASTER", "BACKUP"}
Event	HOST_EVT_HOST_MAINTENANCE	TXT_HOST_IP|TXT_HOST_NAME

D. Liste aller Application-Logging-Daten

Application-Logging

Kind	Type	Data
Static	APP_NAME	TXT_APP_NAME
Static	APP_SLICE	INT_MS
Static	APP_PERIOD	INT_MS
Cyclic	APP_RUNTIME	FLOAT_MIN_MS|FLOAT_AVG_MS|FLOAT_MAX_MS
Cyclic	APP_VM_MASTER_LOCATION	TXT_HOST_IP
Cyclic	APP_VM_BACKUP_LOCATION	TXT_HOST_IP
Event	APP_EVT_VM_START	TXT_VM_NAME|TXT_VM_ROLE ∈ {"MASTER", "BACKUP", "NODE"}
Event	APP_EVT_VM_STOP	TXT_VM_NAME|TXT_VM_ROLE ∈ {"MASTER", "BACKUP", "NODE"}
Event	APP_EVT_OUTGOING_MIGRATION	SRC_HOST|DST_HOST
Event	APP_EVT_OUTGOING_REMUS	SRC_HOST|DST_HOST
Event	APP_EVT_HOST_MAINTENANCE	TXT_HOST_IP|TXT_HOST_NAME

Host- Error

Kind	Type	Data
Event	HOST_MONITORING_NA	TXT_HOST_IP
Event	HOST_IFCE_INTERNODE_DOWN	TXT_HOST_IP
Event	HOST_IFCE_SENSORS_DOWN	TXT_HOST_IP
Event	HOST_DEAD	TXT_HOST_IP

APP/VM-Error

Kind	Type	Data
Event	VM_CRASH	TXT_VM_NAME|TXT_APP_NAME
Event	VM_DEADLINE_MISS	TXT_APP_NAME
Event	VM_WCET_MISS	TXT_APP_NAME

E. Dokumentation der Frontend-Implementierung

(Dominic Wirkner)

E.1. Verzeichnisstruktur

- / Controller der Haupt-Seiten (Aufruf über direkte Links)
- **/cache** Cache-Verzeichnis der Template-Engine Smarty; deaktiviert
- **/configs/global_config.php** globale Konfigurationsdatei der der Seite
- **/css** CSS-Style-Dateien und das Theme von JQueryUI
- **/helpers** Controller für Aufrufe per AJAX
- **/img** genutzte Bilder
- **/js** JavaScript-Implementierungen
- **/lib** genutzte Fremdsoftware und Bibliotheken
- **/models** Modell-Klassen zu XML- und Log-Dateien
- **/templates** Views der Hauptseiten und Hilfstemplates
- **/templates_c** Cache-Verzeichnis für kompilierte Templates (Smarty)

E.2. global_config.php

Listing 27: Auszug der Konfiguration

```
1 /* Global Configuration */
2 define('PAGE_ROOT', '/var/www/pg574_v2/');
3 define('WEB_ROOT', 'http://192.168.56.101/pg574_v2/');
4
5 /* NAS/XML - Configuration */
6 define('NAS_ROOT', '/home/nas/');
7 define('HOST_XML_FILE', NAS_ROOT . 'hosts.xml');
8 define('LMU_INBOX_PATH', NAS_ROOT . 'lmu_inbox/');
9 define('LMU_OUTBOX_PATH', NAS_ROOT . 'lmu_outbox/');
```

```
10 define('ISO_IMAGE_PATH', NAS_ROOT . 'images/');
11 define('TASK_PATH', NAS_ROOT . 'tasks/');
12 define('LOG_FILES_PATH', NAS_ROOT . 'log/');
13 define('XSD_FILES_PATH', PAGE_ROOT . 'models/xsd/');
14
15 /* Initialize Smarty */
16 define('SMARTY_DIR', PAGE_ROOT . 'lib/smarty/');
17 require (SMARTY_DIR . 'Smarty.class.php');
18 $smarty = new Smarty();
19 ...
```

F. Funktionstests für LMU

(Heng Liu)

Test 1 Dienste starten Normaler Funktionstest der LMU von verschiedenen Diensten

Ablauf des Tests:

1. Verzeichnisscanner wird mit DEBUG aufgerufen.
2. Testordner mit start_task_test_timestamp wird angelegt.
3. hosts.xml wird generiert und in den Testordner gelegt.
4. task_timestamp.xml wird generiert und in Pending gelegt.
5. action_timestamp.xml mit dem Befehl „start_task“ wird generiert und in lmu_inbox gelegt.
6. aus lmu_outbox wird 5 x 3 Sekunden lang reply_timestamp aufgerufen.
7. nach erhaltener Reply wird die heap.dump.csv in den Testordner kopiert.
8. Reply-Fehler werden ausgelesen und direkt im Testordner in failure_list_timestamp.csv geschrieben.

9. ausgelesene Fehler werden mit dem erwarteten Fehler verglichen. Bei Match wird der Testordner in OK_timestamp umbenannt.
10. Test wird beendet, alle generierten xml aufgeräumt und in den Test- bzw. OK-Ordner gelegt.
11. Verzeichnisscanner wird beendet.

Konfiguration der Hosts:

Hostname	IP
ubuntu-m-l	129.217.43.30
ubuntu-r	129.217.43.55
ubuntu-l	129.217.43.56

Konfiguration der VMs:

VM-Name	Red.	Prio.	Periode	WCET	IP	MAC	Image	OS
start1	1	1	60	20	xxxx	xxxx	pfad	ciao
start2	2	1	70	30	xxxx	xxxx	pfad	ciao
start3	3	1	50	10	xxxx	xxxx	pfad	ciao

Ausgabe der LMU:

Hostname	Core	Core-Aus.	VM-Name	WCET	Periode	Red.	Status
ubuntu-m-l	1	0.333	start1	20	60	M	M
ubuntu-m-l	2	0.2	start3	10	50	MBN	M
ubuntu-r	1	0.428	start2	30	70	MB	M
ubuntu-r	2	0.2	start3	10	50	MBN	B
ubuntu-l	1	0.428	start2	30	70	MB	B
ubuntu-l	2	0.2	start3	10	50	MBN	N

Resultat des Tests Test verlief erfolgreich. Alle Dienste wurden erfolgreich gestartet, weil es ausreichende Ressourcen zur Verfügung gibt.

Test 2 Dienste starten, dann pausieren Funktionstest der LMU. Der Dienst wird zuerst gestartet, dann pausiert.

Ablauf des Tests:

1. Verzeichnisscanner wird mit DEBUG aufgerufen.
2. Testordner mit start_pause_task_test_timestamp wird angelegt.
3. hosts.xml wird generiert und in den Testordner gelegt.
4. Beliebige task.xml wird generiert und in Pending gelegt.
5. action_timestamp.xml mit dem Befehl„start_task“ wird generiert und in lmu_inbox gelegt.
6. aus lmu_outbox wird 5 x 3 Sekunden lang reply_timestamp aufgerufen.
7. nach dem Starten dieses Task wird eine zweite action.xml generiert, um den zu pausieren.
8. Nach erhaltener Reply wird die heap_dump.csv in den Testordner kopiert.
9. Reply-Fehler werden ausgelesen und direkt im Testordner in failure_list_timestamp.csv geschrieben.
10. Ausgelesene Fehler werden mit dem erwarteten Fehler verglichen. Bei Match wird der Testordner in OK_timestamp umbenannt.
11. Der Test wird beendet,alle generierten XML-Dateien werden aufgeräumt und in den Test- bzw. OK-Ordner gelegt.
12. Der Verzeichnisscanner wird beendet.

Konfiguration der Hosts:

Hostname	IP
ubuntu-m-r	129.217.43.31
ubuntu-m-l	129.217.43.30
ubuntu-r	129.217.43.55

Konfiguration der VMs:

VM-Name	Red.	Prio.	Periode	WCET	IP	MAC	Image	OS
start_pause1	1	1	60	20	xxxx	xxxx	pfad	ciao
start_pause2	2	1	50	10	xxxx	xxxx	pfad	ciao
start_pause3	3	1	40	10	xxxx	xxxx	pfad	ciao

Ausgabe der LMU:

Hostname	Core	Core-Aus.	VM-Name	WCET	Periode	Red.	Status

Resultat des Tests In der Testphase wurden alle drei Dienste zuerst gestartet, danach wieder erfolgreich pausiert. Die Ergebnisse haben die Erwartungen des Tests erfüllt.

Test 3 Dienste starten, dann stoppen/löschen Test der LMU, bei dem der Dienst zuerst gestartet und dann gestoppt wird.

Ablauf des Tests:

1. Verzeichnisscanner wird mit DEBUG aufgerufen.
2. Testordner mit start_delete_task_test_timestamp wird angelegt.
3. hosts.xml wird generiert und in den Testordner gelegt.
4. beliebige task.xml wird generiert und in Pending gelegt.
5. action_timestamp.xml mit dem Befehl „start_task“ wird generiert und in lmu_inbox gelegt.
6. aus lmu_outbox wird 5 x 3 Sekunden lang reply_timestamp aufgerufen.
7. nach erfolgreichem Starten dieses Task wird eine zweite action.xml generiert, um den zu stoppen.
8. nach erhaltener Reply wird die heap_dump.csv in den Testordner kopiert.
9. Reply-Fehler werden ausgelesen und direkt im Testordner in failure_list_timestamp.csv geschrieben.

10. ausgelesene Fehler werden mit dem erwarteten Fehler verglichen. Bei Match wird der Testordner in OK_timestamp umbenannt.
11. Test wird beendet,alle generierten xml aufgeräumt und in den Test- bzw. OK-Ordner gelegt.
12. Verzeichnisscanner wird beendet.

Konfiguration der Hosts:

Hostname	IP
ubuntu-m-l	129.217.43.30
ubuntu-l	129.217.43.56
ubuntu-r	129.217.43.55

Konfiguration der VMs:

VM-Name	Red.	Prio.	Periode	WCET	IP	MAC	Image	OS
start_stop1	1	1	60	20	xxxx	xxxx	pfad	ciao
start_stop2	2	1	70	20	xxxx	xxxx	pfad	ciao
start_stop3	3	1	50	20	xxxx	xxxx	pfad	ciao

Ausgabe der LMU:

Hostname	Core	Core-Aus.	VM-Name	WCET	Periode	Red.	Status

Resultat des Tests Hierzu wurden die Dienste am Anfang richtig gestartet und anschließend gestoppt. Das Resultat ist wie erwartet.

Test 4 Gleicher Dienst wird zwei mal gestartet Es wird zwei mal der gleiche Dienst gestartet.

Ablauf des Tests:

1. Verzeichnisscanner wird mit DEBUG aufgerufen.

2. Testordner mit start2times_task_test_timestamp wird angelegt.
3. hosts.xml wird generiert und in den Testordner gelegt.
4. task_timestamp.xml wird generiert und in Pending gelegt.
5. action_timestamp.xml wird generiert und in lmu_inbox gelegt.
6. wartet auf reply_timestamp.xml.
7. die gleiche task_timestamp.xml wird nochmal generiert.
8. Fehlerchecken und wartet auf reply_timestamp.xml.
9. nach erhaltener Reply wird die heap.dump.csv in den Testordner kopiert.
10. Reply-Fehler werden ausgelesen und direkt im Testordner in failure_list_timestamp.csv geschrieben.
11. ausgelesene Fehler werden mit dem erwarteten Fehler verglichen. Bei Match wird der Testordner in OK_timestamp umbenannt.
12. Test wird beendet, alle generierten xml aufgeräumt und in den Test- bzw. OK-Ordner gelegt.
13. Verzeichnisscanner wird beendet.

Konfiguration der Hosts:

Hostname	IP
ubuntu-m-l	129.217.43.30
ubuntu-l	129.217.43.56
ubuntu-r	129.217.43.55

Konfiguration der VMs:

VM-Name	Red.	Prio.	Periode	WCET	IP	MAC	Image	OS
2times1	1	1	60	30	xxxx	xxxx	pfad	ciao
2times2	2	1	60	30	xxxx	xxxx	pfad	ciao
2times3	3	1	60	30	xxxx	xxxx	pfad	ciao

Ausgabe der LMU:

Hostname	Core	Core-Aus.	VM-Name	WCET	Periode	Red.	Status

Resultat des Tests Bei diesem Test wurde der gleiche Dienst mehrmals hintereinander gestartet. Nachdem der erste Dienst erfolgreich gestartet wurde, wurden die gleichen Task.xml und action.xml generiert, um den gleichen Dienst wiederum zu starten. In der Regel wäre es unmöglich, weil die LMU nicht erlaubt, die Dienste mit gleichem Name auszuführen. Es wird Fehlercode zurückbekommen.

Test 5 WCET größer als Periode Bei dem Dienst ist WCET immer größer als Periode.

Ablauf des Tests:

1. Verzeichnisscanner wird mit DEBUG aufgerufen.
2. Testordner mit wcet_longer_period_test_timestamp wird angelegt.
3. hosts.xml wird generiert und in den Testordner gelegt.
4. task_timestamp.xml wird generiert, der Wert von WCET ist größer als Periode.
5. action_timestamp.xml wird generiert und in lmu_inbox gelegt.
6. Fehlerchecken und wartet auf reply_timestamp.xml.
7. nach erhaltener Reply wird die heap.dump.csv in den Testordner kopiert.
8. Reply-Fehler werden ausgelesen und direkt im Testordner in failure_list_timestamp.csv geschrieben.
9. ausgelesene Fehler werden mit dem erwarteten Fehler verglichen. Bei Match wird der Testordner in OK_timestamp umbenannt.
10. Test wird beendet, alle generierten xml aufgeräumt und in den Test- bzw. OK-Ordner gelegt.
11. Verzeichnisscanner wird beendet.

Konfiguration der Hosts:

Hostname	IP
ubuntu-m-l	129.217.43.30
ubuntu-l	129.217.43.56
ubuntu-r	129.217.43.55

Konfiguration der VMs:

VM-Name	Red.	Prio.	Periode	WCET	IP	MAC	Image	OS
wcet1	1	5	40	60	xxxx	xxxx	pfad	ciao
wcet2	2	6	30	50	xxxx	xxxx	pfad	ciao
wcet3	3	1	10	90	xxxx	xxxx	pfad	ciao

Ausgabe der LMU:

Hostname	Core	Core-Aus.	VM-Name	WCET	Periode	Red.	Status

Resultat des Tests Wegen der Definition darf die Länge der WCET nicht Periode überschritten. Nach Ablauf dieses Tests kommt eine Fehlermeldung, weshalb der Test erfolgreich ist.

Test 6 Belastungstest Beliebig viele Dienste werden in einem Zeitraum hintereinander gestartet. Ob die LMU überbelastet ist, hängt von den verfügbaren Hosts, VMs und Cores ab. Dieser Belastungstest ist am wichtigsten bei der Testmodule für LMU.

Ablauf des Tests:

1. Verzeichnisscanner wird mit DEBUG aufgerufen.
2. Testordner mit belastung_test_timestamp wird angelegt.
3. hosts.xml wird generiert und in den Testordner gelegt.
4. in einem Zeitraum werden beliebig viele Tasks hintereinander gestartet sollten.
5. task_timestamp.xml wird generiert und in Pending angelegt.

6. action_timestamp.xml wird generiert und in lmu_inbox gelegt.
7. aus lmu_outbox wird 5 x 3 Sekunden lang reply_timestamp aufgerufen.
8. nach erhaltener Reply wird die heap.dump.csv in den Testordner kopiert.
9. Reply-Fehler werden ausgelesen und direkt im Testordner in failure_list_timestamp.csv geschrieben.
10. ausgelesene Fehler werden mit dem erwarteten Fehler verglichen. Bei Match wird der Testordner in OK_timestamp umbenannt.
11. Test wird beendet, alle generierten xml aufgeräumt und in den Test- bzw. OK-Ordner gelegt.
12. Verzeichnisscanner wird beendet.

1/2 der Belastungstest

Konfiguration der Hosts:

Hostname	IP
ubuntu-m-l	129.217.43.30
ubuntu-m-r	129.217.43.31
ubuntu-l	129.217.43.56
ubuntu-r	129.217.43.55

Konfiguration der VMs:

VM-Name	Red.	Prio.	Periode	WCET	IP	MAC	Image	OS
belastung1	1	1	60	10	xxxx	xxxx	pfad	ciao
belastung2	1	1	70	10	xxxx	xxxx	pfad	ciao
belastung3	1	1	60	10	xxxx	xxxx	pfad	ciao
belastung4	1	1	70	10	xxxx	xxxx	pfad	ciao
belastung5	2	1	70	20	xxxx	xxxx	pfad	ciao
belastung6	2	1	80	40	xxxx	xxxx	pfad	ciao
belastung7	2	1	60	30	xxxx	xxxx	pfad	ciao
belastung8	2	1	90	10	xxxx	xxxx	pfad	ciao
belastung9	2	1	90	10	xxxx	xxxx	pfad	ciao
belastung10	3	1	50	30	xxxx	xxxx	pfad	ciao
belastung11	3	1	50	20	xxxx	xxxx	pfad	ciao
belastung12	3	1	50	10	xxxx	xxxx	pfad	ciao
belastung13	3	1	80	30	xxxx	xxxx	pfad	ciao
belastung14	3	1	80	10	xxxx	xxxx	pfad	ciao
belastung15	3	1	80	10	xxxx	xxxx	pfad	ciao
belastung16	3	1	80	10	xxxx	xxxx	pfad	ciao
belastung17	2	1	80	10	xxxx	xxxx	pfad	ciao
belastung18	1	1	80	10	xxxx	xxxx	pfad	ciao
belastung19	3	1	80	10	xxxx	xxxx	pfad	ciao
belastung20	2	1	100	10	xxxx	xxxx	pfad	ciao

Ausgabe der LMU:

Hostname	Core	Core-Aus.	VM-Name	WCET	Periode	Red.	Status
ubuntu-m-l	1	0.825	belastung7	30	60	MB	B
ubuntu-m-l	1	0.825	belastung12	10	50	MBN	N
ubuntu-m-l	1	0.825	belastung17	10	80	MB	M
ubuntu-m-l	2	0.836	belastung8	10	90	MB	M
ubuntu-m-l	2	0.836	belastung10	30	50	MBN	M
ubuntu-m-l	2	0.836	belastung18	10	80	M	M
ubuntu-m-l	3	0.886	belastung9	10	90	MB	M
ubuntu-m-l	3	0.886	belastung11	20	50	MBN	M

ubuntu-m-l	3	0.886	belastung14	10	80	MBN	N
ubuntu-m-l	3	0.886	belastung16	10	80	MBN	M
ubuntu-m-l	3	0.886	belastung19	10	80	MBN	N
ubuntu-l	1	0.878	belastung2	10	70	M	M
ubuntu-l	1	0.878	belastung8	10	90	MB	B
ubuntu-l	1	0.878	belastung11	20	50	MBN	N
ubuntu-l	1	0.878	belastung16	10	80	MBN	B
ubuntu-l	1	0.878	belastung20	10	100	MB	M
ubuntu-l	2	0.860	belastung5	20	70	MB	M
ubuntu-l	2	0.860	belastung12	10	50	MBN	B
ubuntu-l	2	0.860	belastung13	30	80	MBN	B
ubuntu-l	3	0.875	belastung6	40	80	MB	B
ubuntu-l	3	0.875	belastung14	10	80	MBN	B
ubuntu-l	3	0.875	belastung15	10	80	MBN	N
ubuntu-l	3	0.875	belastung19	10	80	MBN	M
ubuntu-m-r	1	0.853	belastung3	10	70	M	M
ubuntu-m-r	1	0.853	belastung9	10	90	MB	B
ubuntu-m-r	1	0.853	belastung10	30	50	MBN	N
ubuntu-m-r	2	0.860	belastung5	20	70	MB	B
ubuntu-m-r	2	0.860	belastung12	10	50	MBN	M
ubuntu-m-r	2	0.860	belastung13	30	80	MBN	M
ubuntu-m-r	3	0.875	belastung7	30	60	MB	M
ubuntu-m-r	3	0.875	belastung14	10	80	MBN	M
ubuntu-m-r	3	0.875	belastung15	10	80	MBN	B
ubuntu-m-r	3	0.875	belastung19	10	80	MBN	B
ubuntu-r	1	0.916	belastung1	10	60	M	M
ubuntu-r	1	0.916	belastung11	20	50	MBN	B
ubuntu-r	1	0.916	belastung15	10	80	MBN	M
ubuntu-r	1	0.916	belastung16	10	80	MBN	N
ubuntu-r	1	0.916	belastung20	10	100	MB	B
ubuntu-r	2	0.867	belastung4	10	70	M	M
ubuntu-r	2	0.867	belastung10	30	50	MBN	B

ubuntu-r	2	0.867	belastung17	10	80	MB	B
ubuntu-r	3	0.875	belastung6	40	80	MB	M
ubuntu-r	3	0.875	belastung13	30	80	MBN	N

Resultat des Tests Für diesen Test stehen vier Hosts zur Verfügung. Wie die Ausgabe von LMU anzuzeigen ist, dass alle zwanzig Dienste mit verschiedenen Konfigurationen auf den unterschiedlichen CPU-Cores der Hosts ausgeführt werden können. Mehrere Dienste können noch ausgeführt werden, weil es bei einigen Cores ausreichende Auslastung gibt. Darum ist dieser Test erfolgreich.

2/2 der Belastungstest

Konfiguration der Hosts:

Hostname	IP
ubuntu-m-l	129.217.43.30
ubuntu-l	129.217.43.56
ubuntu-r	129.217.43.55

Konfiguration der VMs:

VM-Name	Red.	Prio.	Periode	WCET	IP	MAC	Image	OS
belastung1	1	1	60	10	xxxx	xxxx	pfad	ciao
belastung2	1	1	70	10	xxxx	xxxx	pfad	ciao
belastung3	1	1	60	10	xxxx	xxxx	pfad	ciao
belastung4	1	1	70	10	xxxx	xxxx	pfad	ciao
belastung5	2	1	70	20	xxxx	xxxx	pfad	ciao
belastung6	2	1	80	40	xxxx	xxxx	pfad	ciao
belastung7	2	1	60	30	xxxx	xxxx	pfad	ciao
belastung8	2	1	90	10	xxxx	xxxx	pfad	ciao
belastung9	2	1	90	10	xxxx	xxxx	pfad	ciao
belastung10	3	1	50	30	xxxx	xxxx	pfad	ciao
belastung11	3	1	50	20	xxxx	xxxx	pfad	ciao
belastung12	3	1	50	10	xxxx	xxxx	pfad	ciao
belastung13	3	1	80	30	xxxx	xxxx	pfad	ciao
belastung14	3	1	80	10	xxxx	xxxx	pfad	ciao
belastung15	3	1	80	10	xxxx	xxxx	pfad	ciao
belastung16	3	1	80	10	xxxx	xxxx	pfad	ciao
belastung17	2	1	80	10	xxxx	xxxx	pfad	ciao
belastung18	1	1	80	10	xxxx	xxxx	pfad	ciao
belastung19	3	1	80	10	xxxx	xxxx	pfad	ciao
belastung20	2	1	100	10	xxxx	xxxx	pfad	ciao

Ausgabe der LMU:

Hostname	Core	Core-Aus.	VM-Name	WCET	Periode	Red.	Status
ubuntu-l	1	0.996	belastung8	10	90	MB	B
ubuntu-l	1	0.996	belastung3	10	70	M	M
ubuntu-l	1	0.996	belastung10	30	50	MBN	M
ubuntu-l	1	0.996	belastung2	10	70	M	M
ubuntu-l	2	0.935	belastung14	10	80	MBN	M
ubuntu-l	2	0.935	belastung5	20	70	MB	B
ubuntu-l	2	0.935	belastung11	20	50	MBN	M
ubuntu-l	2	0.935	belastung16	10	80	MBN	B

ubuntu-l	3	0.95	belastung15	10	80	MBN	M
ubuntu-l	3	0.95	belastung12	10	50	MBN	M
ubuntu-l	3	0.95	belastung7	30	60	MB	M
ubuntu-l	3	0.95	belastung17	10	80	MB	M
ubuntu-m-l	1	0.997	belastung8	10	90	MB	M
ubuntu-m-l	1	0.997	belastung9	10	90	MB	B
ubuntu-m-l	1	0.997	belastung15	10	80	MBN	N
ubuntu-m-l	1	0.997	belastung11	20	50	MBN	B
ubuntu-m-l	1	0.997	belastung16	10	80	MBN	M
ubuntu-m-l	1	0.997	belastung17	10	80	MB	B
ubuntu-m-l	2	0.985	belastung5	20	70	MB	M
ubuntu-m-l	2	0.985	belastung10	30	50	MBN	N
ubuntu-m-l	2	0.985	belastung20	10	100	MB	M
ubuntu-m-l	3	0.991	belastung14	10	80	MBN	B
ubuntu-m-l	3	0.991	belastung1	10	60	M	M
ubuntu-m-l	3	0.991	belastung12	10	50	MBN	B
ubuntu-m-l	3	0.991	belastung6	40	80	MB	B
ubuntu-r	1	0.961	belastung9	10	90	MB	M
ubuntu-r	1	0.961	belastung15	10	80	MBN	B
ubuntu-r	1	0.961	belastung10	30	50	MBN	B
ubuntu-r	1	0.961	belastung16	10	80	MBN	N
ubuntu-r	2	0.967	belastung14	10	80	MBN	N
ubuntu-r	2	0.967	belastung4	10	70	M	M
ubuntu-r	2	0.967	belastung12	10	50	MBN	N
ubuntu-r	2	0.967	belastung7	30	60	MB	B
ubuntu-r	3	1.0	belastung11	20	50	MBN	N
ubuntu-r	3	1.0	belastung6	40	80	MB	M
ubuntu-r	3	1.0	belastung20	10	100	MB	B

Resultat des Tests In diesem Testfall betrachtet man, dass nur drei Hosts im Betrieb. Allerdings ist die Konfiguration von VMs der Dienste ebenfalls. Von den Ausgaben kann man erkennen, dass einige Dienste nicht gestartet wurden. Der Grund liegt an keinen ausreichenden Ressourcen für die Dienste, z.B. belastung13, belastung18 und belastung19. belastung13 braucht die CPU-Auslastung in Höhe von 0.375 und Master-Backup-Node-Red.. Nach der Ausführung der belastung12 wurde die Ressourcen mittels Rescheduling-Algorithmus durch Migration wieder verteilt. Trotzdem können dreifache CPU-Auslastung in Höhe von 0.375 auf verschiedene Hosts und unterschiede Cores nicht gefunden werden. Deswegen ist belastung13 nicht zu starten. belastung18 und belastung19 sind analog. Darum wird die Belastungsfähigkeit von LMU getestet und unsere Erwartungen erfüllt.

Test 7 Konfigurationstest Die Hosts werden unterschiedlich Konfiguriert. Dabei gibt es keine ausreichende Ressourcen für die Dienste

Ablauf des Tests:

1. Verzeichnisscanner wird mit DEBUG aufgerufen.
2. Testordner mit configuration_test_timestamp wird angelegt.
3. hosts.xml wird generiert und in den Testordner gelegt.
4. task_timestamp.xml wird generiert, dass der Task als M+B/M+B+N gezeichnet ist.
5. action_timestamp.xml wird generiert und in lmu_inbox gelegt.
6. Fehlerchecken und wartet auf reply_timestamp.xml.
7. nach erhaltener Reply wird die heap.dump.csv in den Testordner kopiert.
8. Reply-Fehler werden ausgelesen und direkt im Testordner in failure_list_timestamp.csv geschrieben.
9. ausgelesene Fehler werden mit dem erwarteten Fehler verglichen. Bei Match wird der Testordner in OK_timestamp umbenannt.
10. Test wird beendet, alle generierten xml aufgeräumt und in den Test- bzw. OK-Ordner gelegt.

11. Verzeichnisscanner wird beendet.

Konfiguration der Hosts:

Hostname	IP
ubuntu-r	129.217.43.55

Konfiguration der VMs:

VM-Name	Red.	Prio.	Periode	WCET	IP	MAC	Image	OS
konfigu1	2	1	80	20	xxxx	xxxx	pfad	ciao
konfigu2	3	1	90	30	xxxx	xxxx	pfad	ciao

Ausgabe der LMU:

Hostname	Core	Core-Aus.	VM-Name	WCET	Periode	Red.	Status

Resultat des Tests Für diesen Konfigurationstest kann nur ein Host in einem Fall bereitgestellt werden. Wenn die Dienste mit Red. Master und Backup oder Master, Backup und Node gestartet werden sollen, können die bestimmt nicht ausgeführt werden, weil nur ein Host verfügbar ist. Am Ende sollte die LMU den Dienst zurückweisen und entsprechende Fehlercodes abschicken.

Test 8 Fehler bei Task.xml Es wird eine fehlerhafte Task.xml generiert.

Ablauf des Tests:

1. Verzeichnisscanner wird mit DEBUG aufgerufen.
2. Testordner mit task_xml_error_test_timestamp wird angelegt.
3. hosts.xml wird generiert und in den Testordner gelegt.
4. task_timestamp.xml wird generiert und in Pending gelegt.
5. action_timestamp.xml mit dem Befehl „start_task“ wird generiert und in lmu_inbox gelegt.

6. aus lmu_outbox wird 5 x 3 Sekunden lang reply_timestamp aufgerufen.
7. nach erhaltener Reply wird die heap.dump.csv in den Testordner kopiert.
8. Reply-Fehler werden ausgelesen und direkt im Testordner in failure_list_timestamp.csv geschrieben.
9. ausgelesene Fehler werden mit dem erwarteten Fehler verglichen. Bei Match wird der Testordner in OK_timestamp umbenannt.
10. Test wird beendet, alle generierten xml aufgeräumt und in den Test- bzw. OK-Ordner gelegt.
11. Verzeichnisscanner wird beendet.

Konfiguration der Hosts:

Hostname	IP
ubuntu-m-l	129.217.43.30
ubuntu-r	129.217.43.55

Konfiguration der VMs:

VM-Name	Red.	Prio.	Periode	WCET	IP	MAC	Image	OS
errortask1	1	1	60	20	xxxx	xxxx	pfad	ciao
errortask2	2	1	50	50	xxxx	xxxx	pfad	ciao

Ausgabe der LMU:

Hostname	Core	Core-Aus.	VM-Name	WCET	Periode	Red.	Status

Resultat des Tests Nach der Ausführung des Tests wurde immer der gleiche Fehler bekommt, der heißt, „25: Es wurde keine task.xml gefunden. Die action.xml wird nach old verschoben. “ Da keine richtige Task.xml generiert wurde.

Test 9 Fehlerhafte Action.xml Es wird eine fehlerhafte Action.xml generiert.

Ablauf des Tests:

1. Verzeichnisscanner wird mit DEBUG aufgerufen.
2. Testordner mit action_xml_error_test_timestamp wird angelegt.
3. hosts.xml wird generiert und in den Testordner gelegt.
4. task_timestamp.xml wird generiert und in Pending gelegt.
5. action_timestamp.xml mit dem Befehl „start_task“ wird generiert und in lmu_inbox gelegt.
6. aus lmu_outbox wird 5 x 3 Sekunden lang reply_timestamp aufgerufen.
7. nach erhaltener Reply wird die heap.dump.csv in den Testordner kopiert.
8. Reply-Fehler werden ausgelesen und direkt im Testordner in failure_list_timestamp.csv geschrieben.
9. ausgelesene Fehler werden mit dem erwarteten Fehler verglichen. Bei Match wird der Testordner in OK_timestamp umbenannt.
10. Test wird beendet, alle generierten XML-Dateien aufgeräumt und in den Test- bzw. OK-Ordner gelegt.
11. Verzeichnisscanner wird beendet.

Konfiguration der Hosts:

Hostname	IP
ubuntu-l	129.217.43.56
ubuntu-r	129.217.43.55

Konfiguration der VMs:

VM-Name	Red.	Prio.	Periode	WCET	IP	MAC	Image	OS
erroraction1	1	1	60	20	xxxx	xxxx	pfad	ciao
erroraction2	2	3	80	70	xxxx	xxxx	pfad	ciao
erroraction3	1	2	100	10	xxxx	xxxx	pfad	ciao

Ausgabe der LMU:

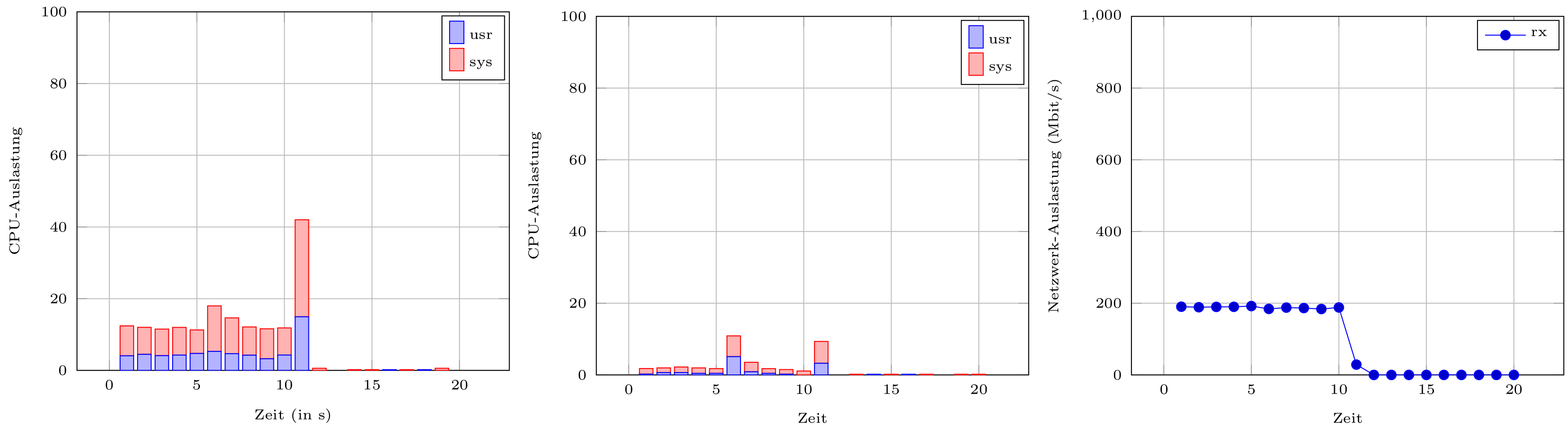
Hostname	Core	Core-Aus.	VM-Name	WCET	Periode	Red.	Status

Resultat des Tests Eine fehlerhafte Action.xml bedeutet, dass der Befehl nicht zur LMU übertragen werden kann, weil der möglichweise in der Action.xml falsch definiert. Es wird den Fehler zurückbekommen, „27: Die action.xml ist Fehlerhaft.“

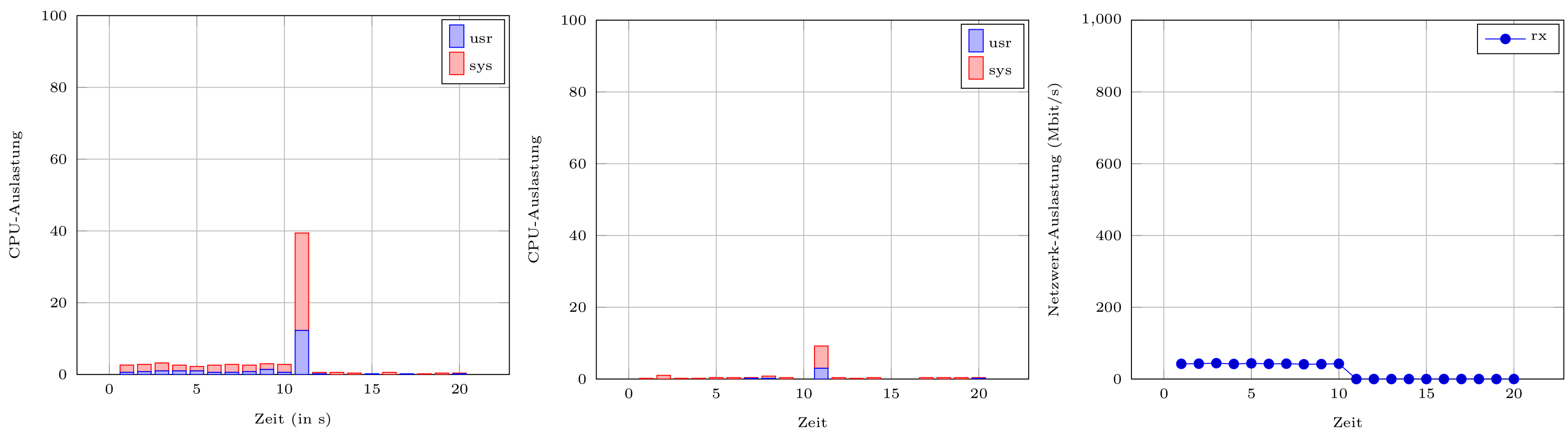
Anmerkung: In den Tests können die IP-Adresse, MAC, Image und OS der Konfiguration der VMs ignoriert werden, da die Daten beim dem lokalen Test nicht benötigt werden.

G. Ergebnisse zur Evaluation der Remus-Replikation

(Tim Harde)

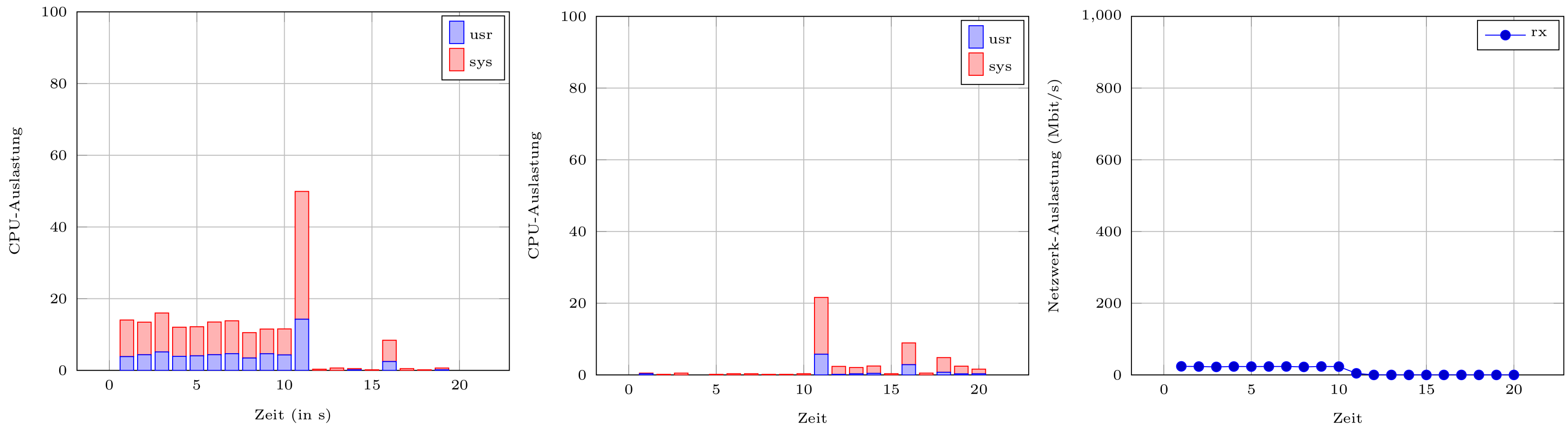


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

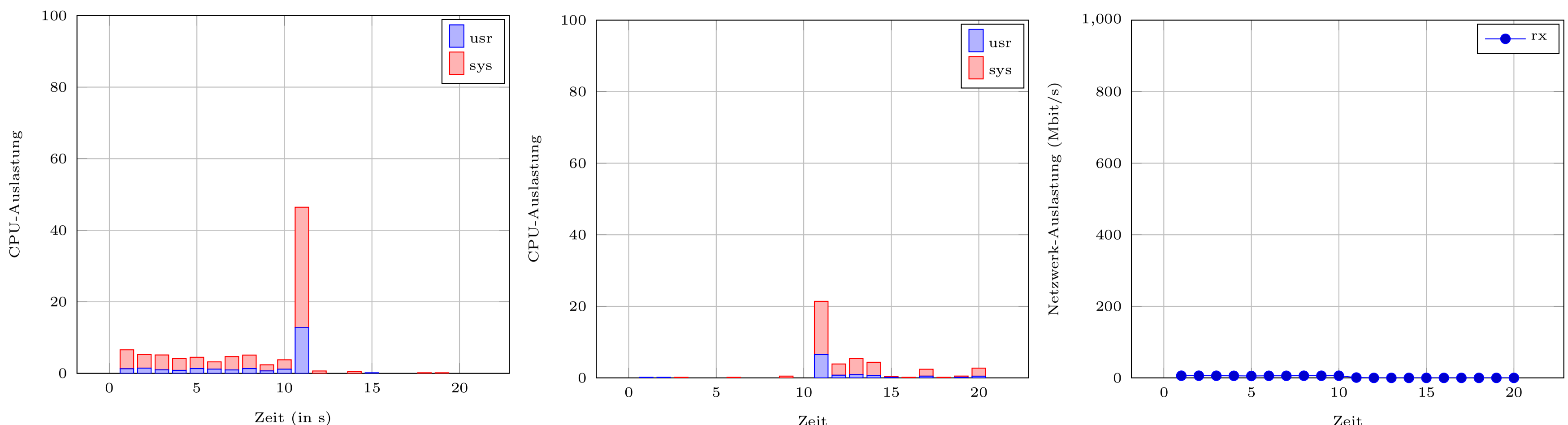


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 48: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *alpine* als Gast-Betriebssystem (Zwischenankunftszeit: 0.01 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

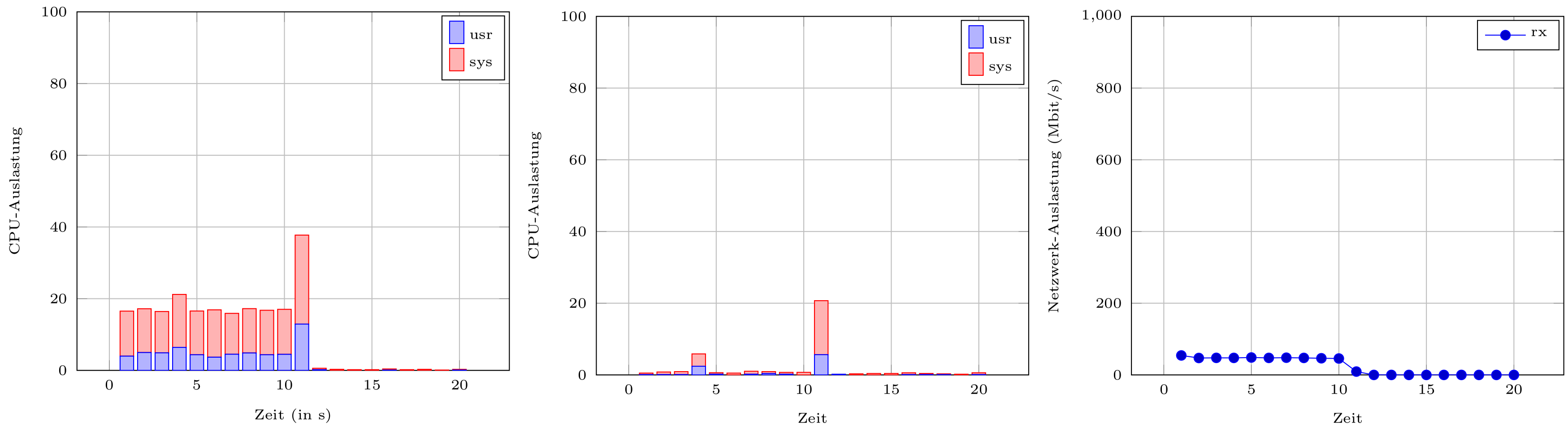


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

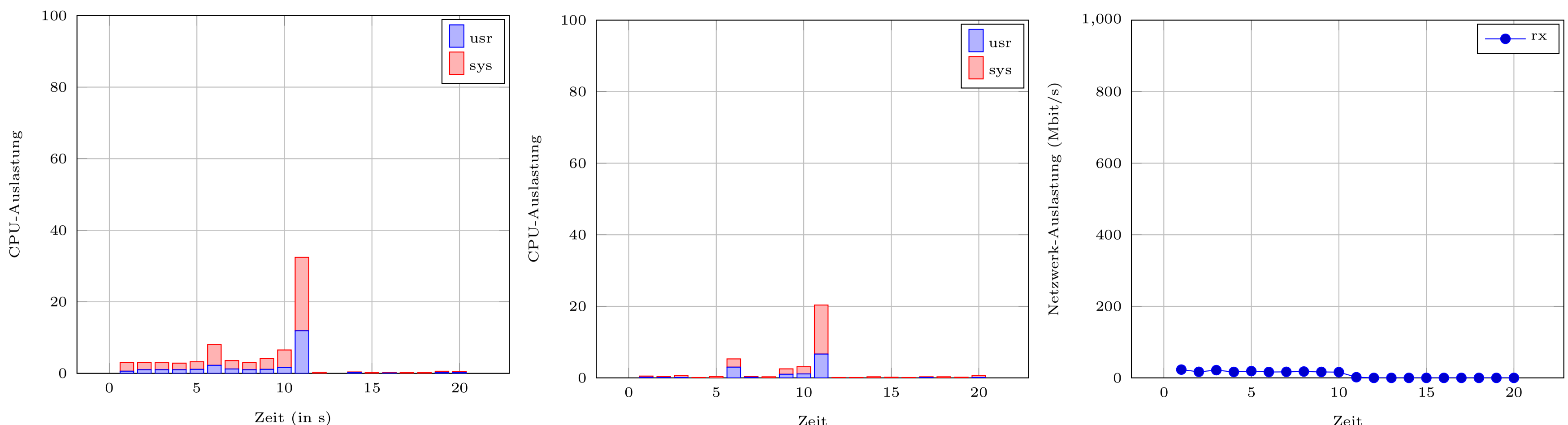


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 49: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ciao* als Gast-Betriebssystem (Zwischenankunftszeit: 0.01 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

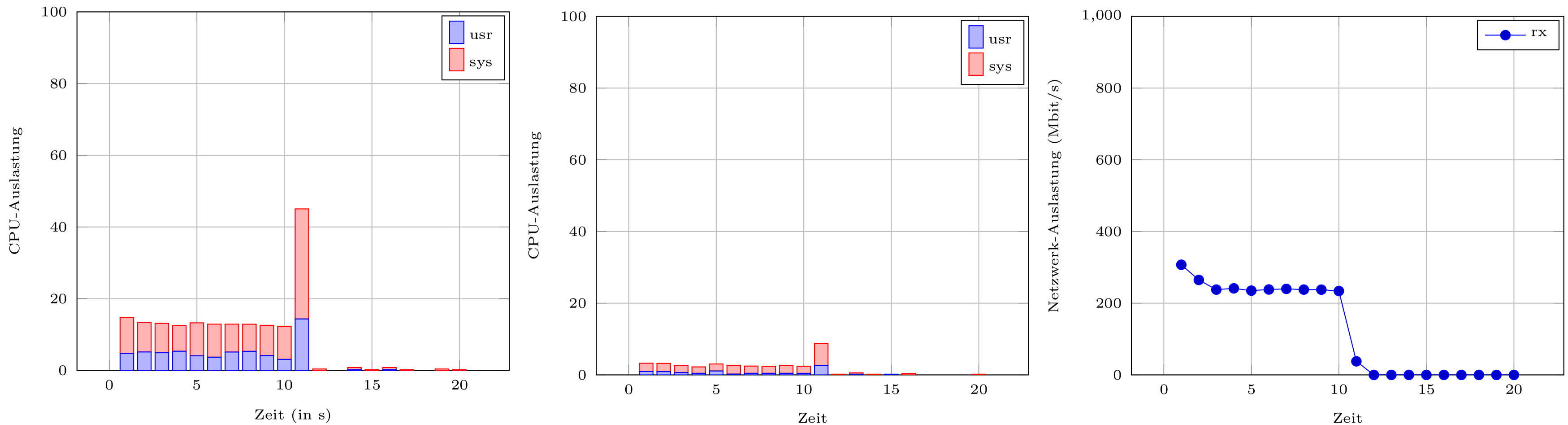


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

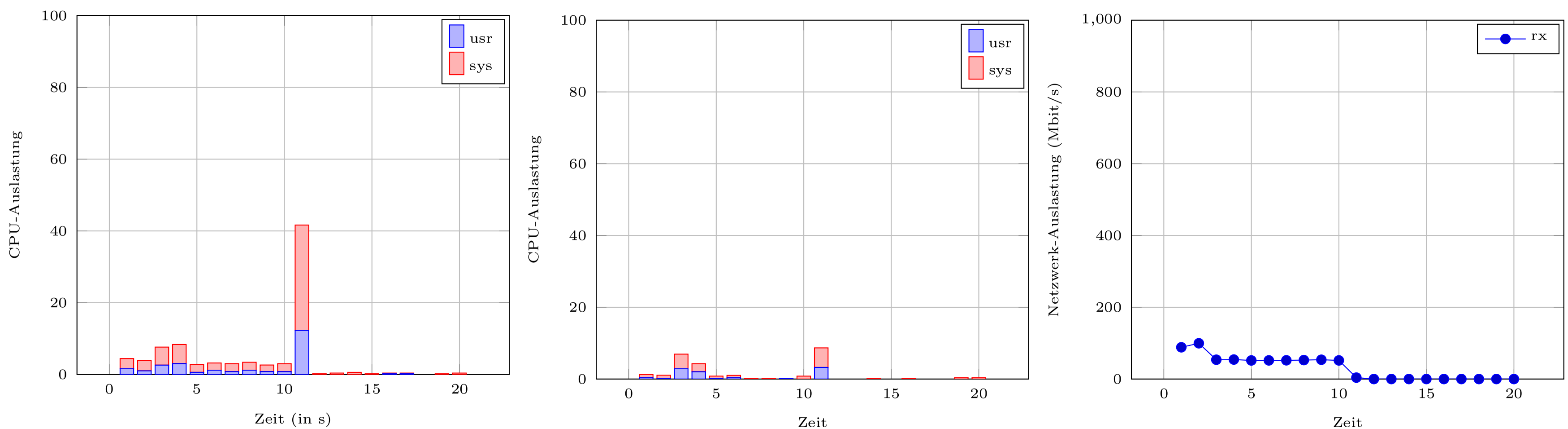


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 50: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *tinycore* als Gast-Betriebssystem (Zwischenankunftszeit: 0.01 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

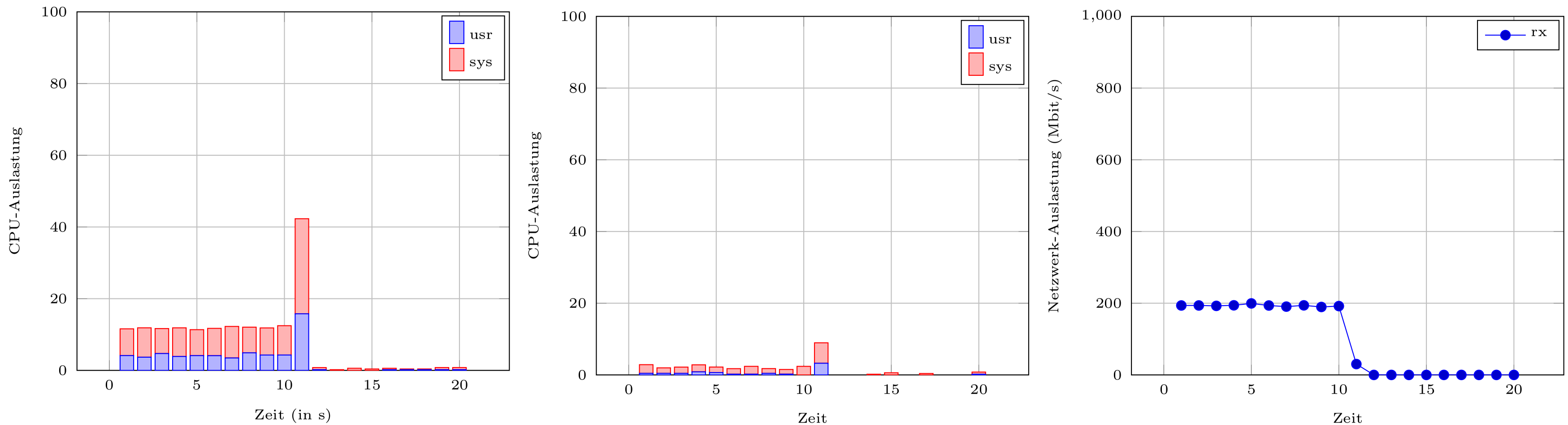


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

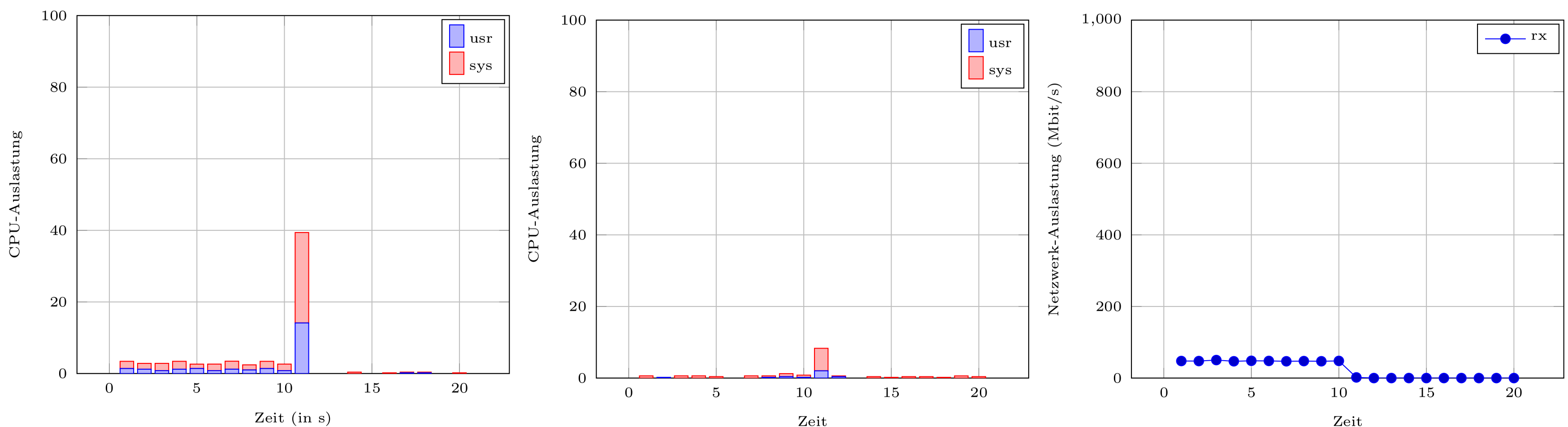


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 51: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ubuntu* als Gast-Betriebssystem (Zwischenankunftszeit: 0.01 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

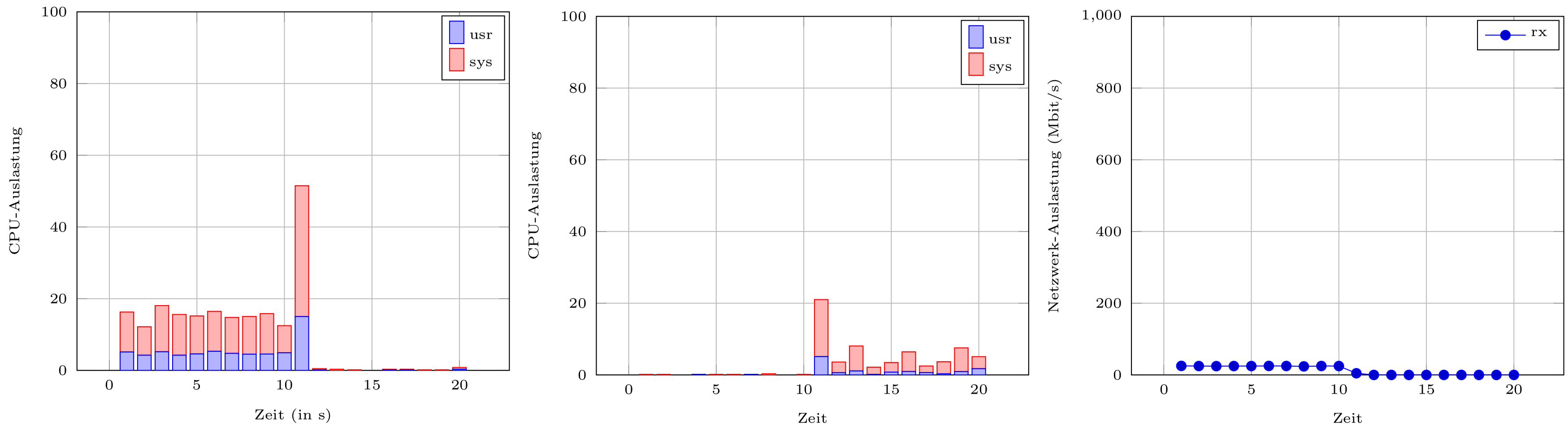


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

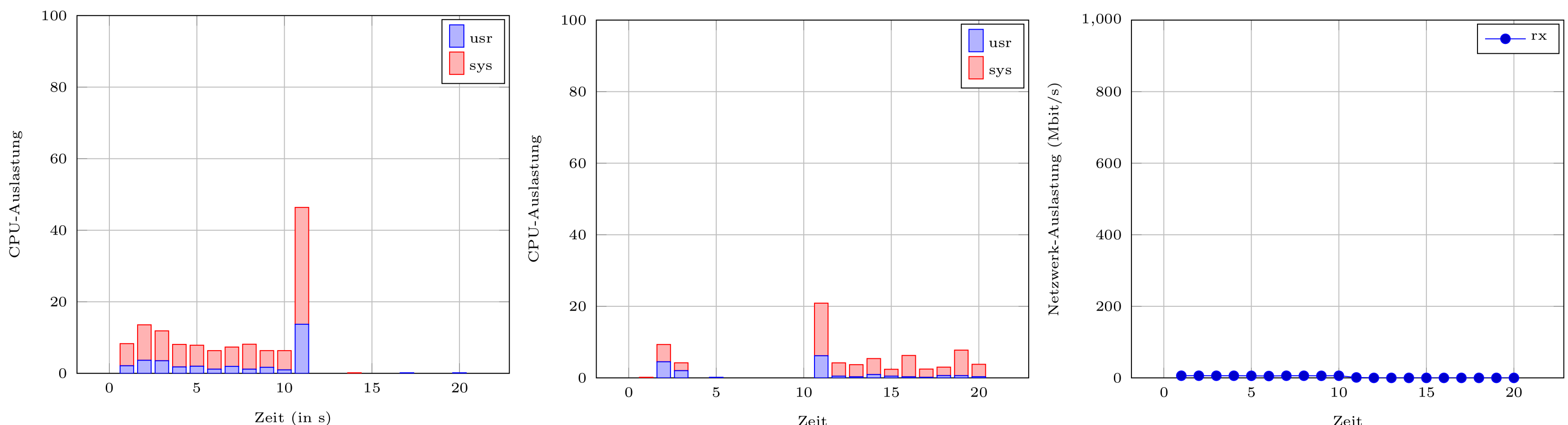


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 52: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *alpine* als Gast-Betriebssystem (Zwischenankunftszeit: 0.005 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

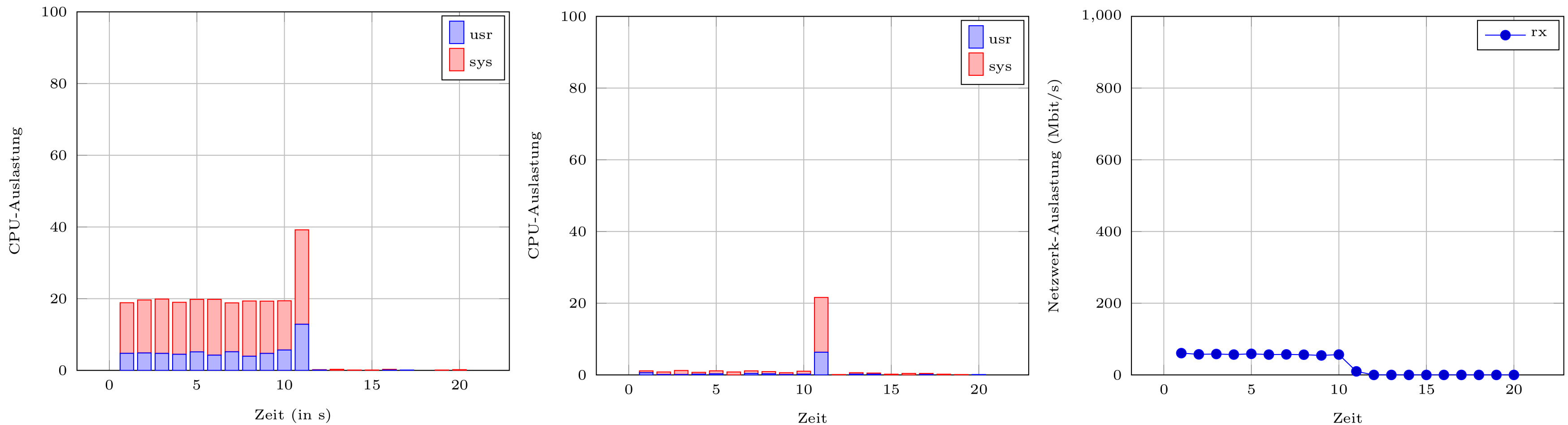


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

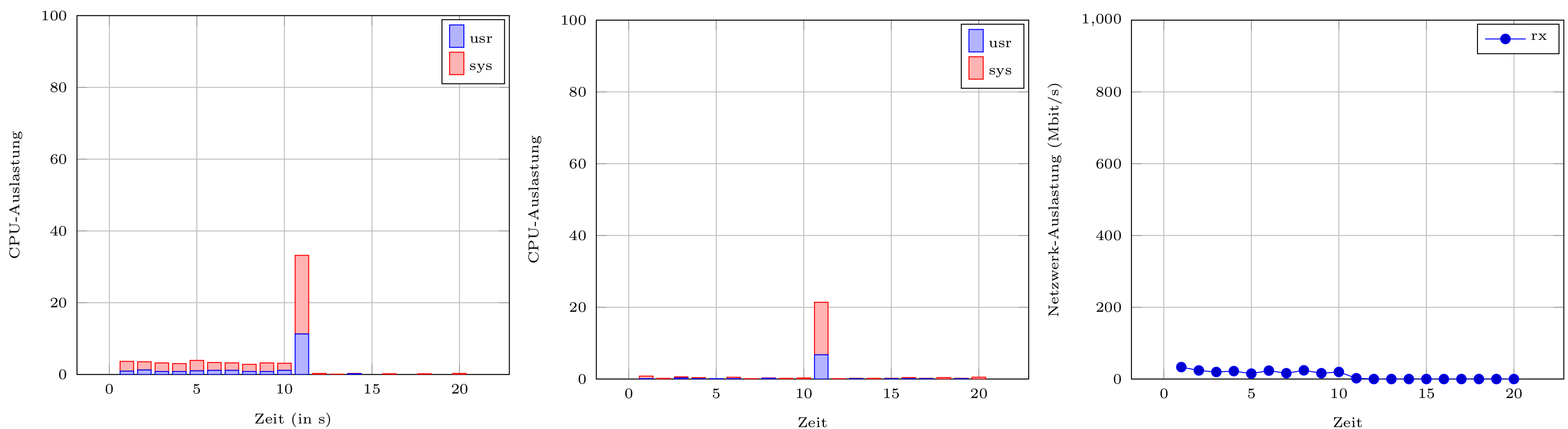


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 53: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ciao* als Gast-Betriebssystem (Zwischenankunftszeit: 0.005 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

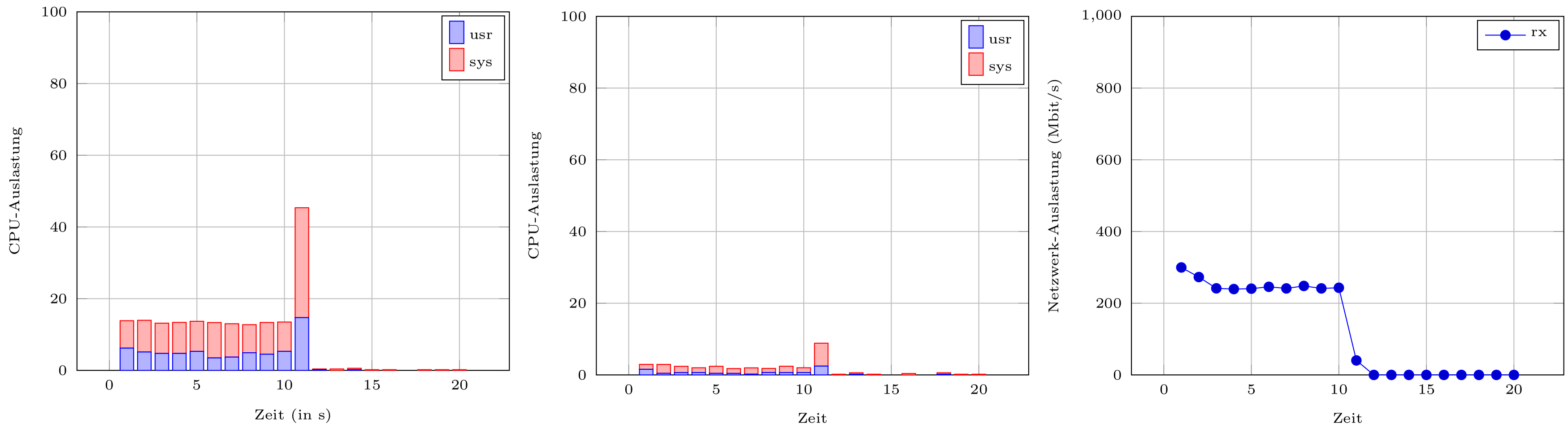


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

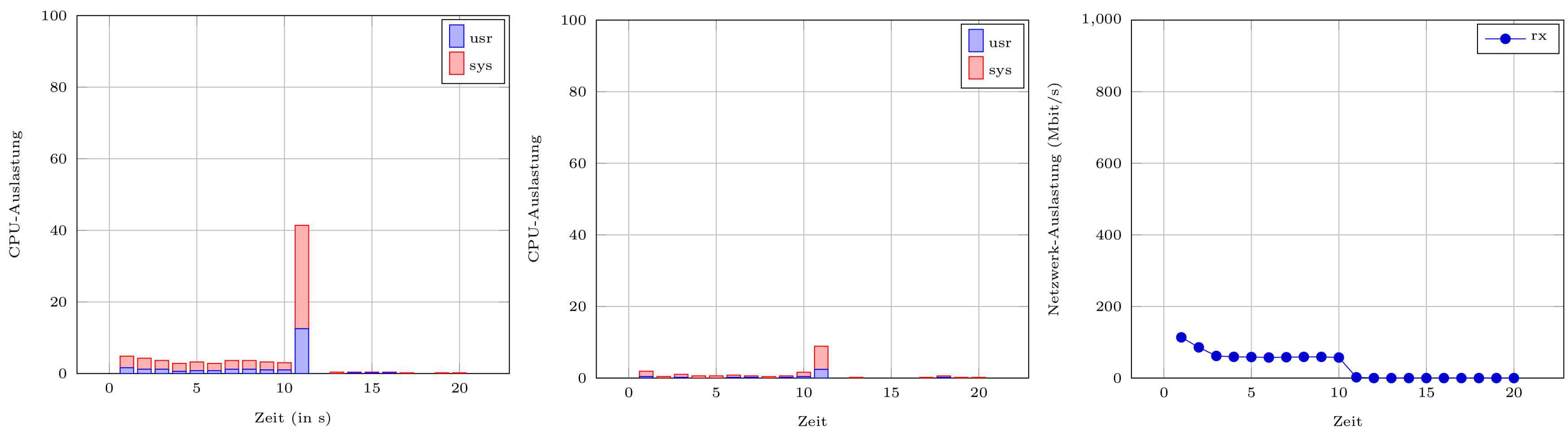


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 54: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *tinycore* als Gast-Betriebssystem (Zwischenankunftszeit: 0.005 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

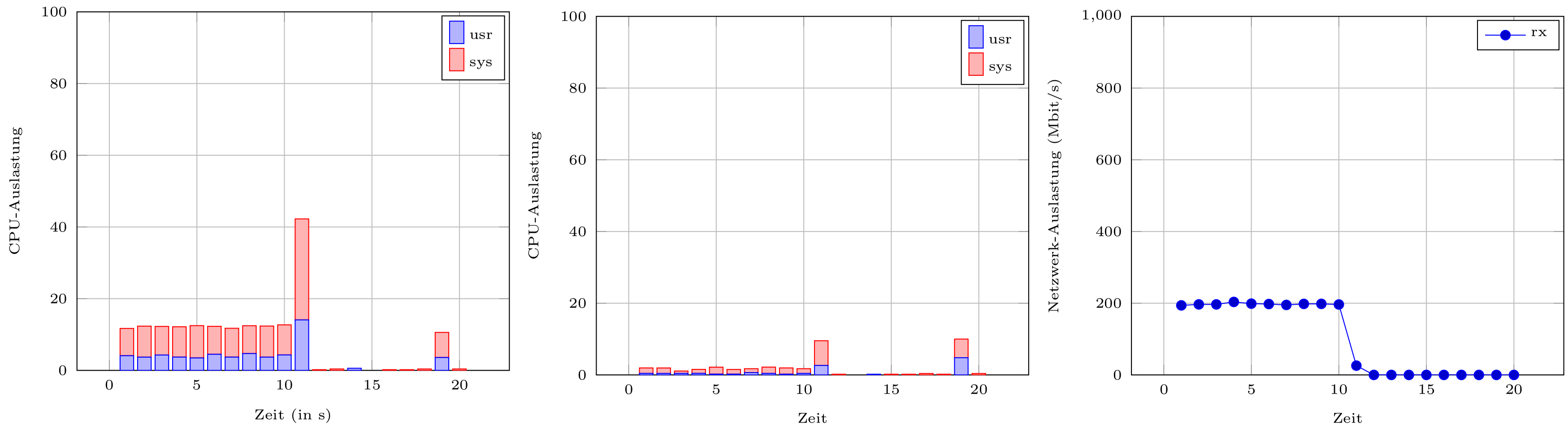


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

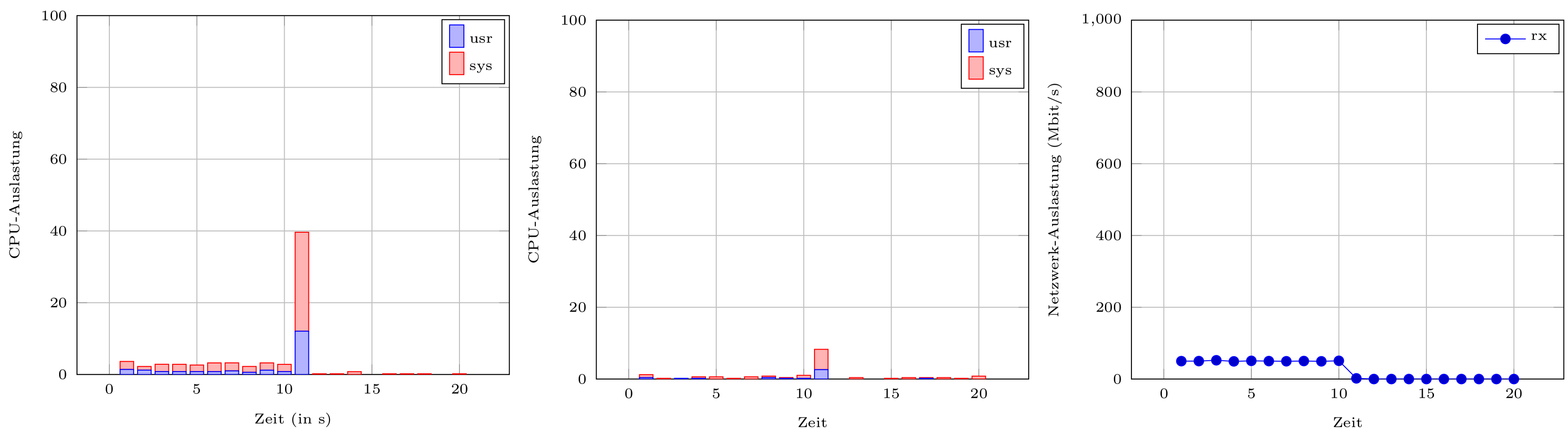


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 55: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ubuntu* als Gast-Betriebssystem (Zwischenankunftszeit: 0.005 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

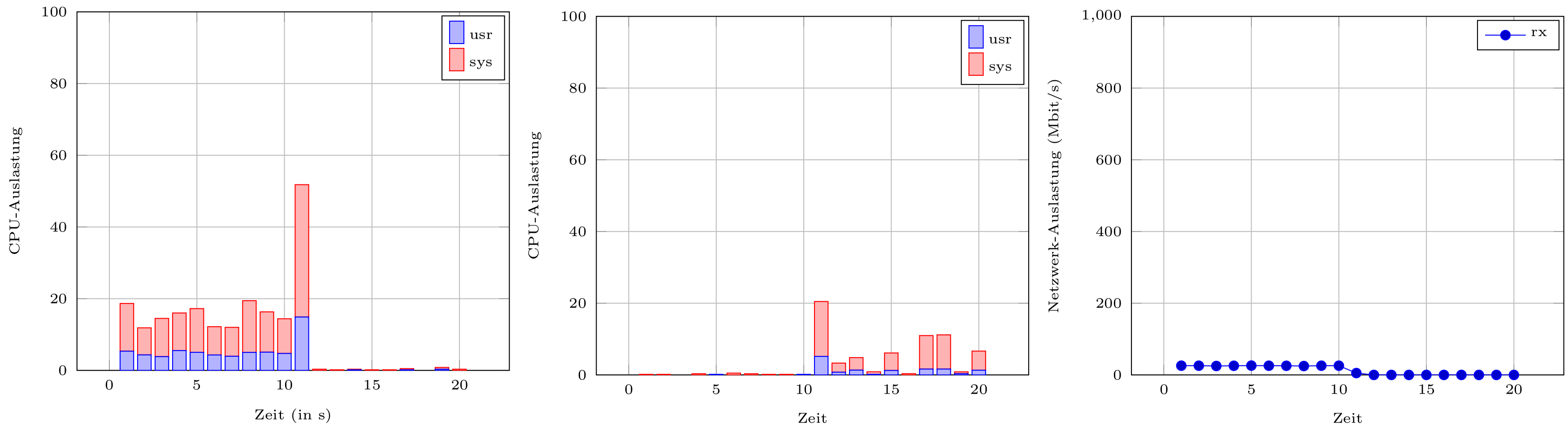


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

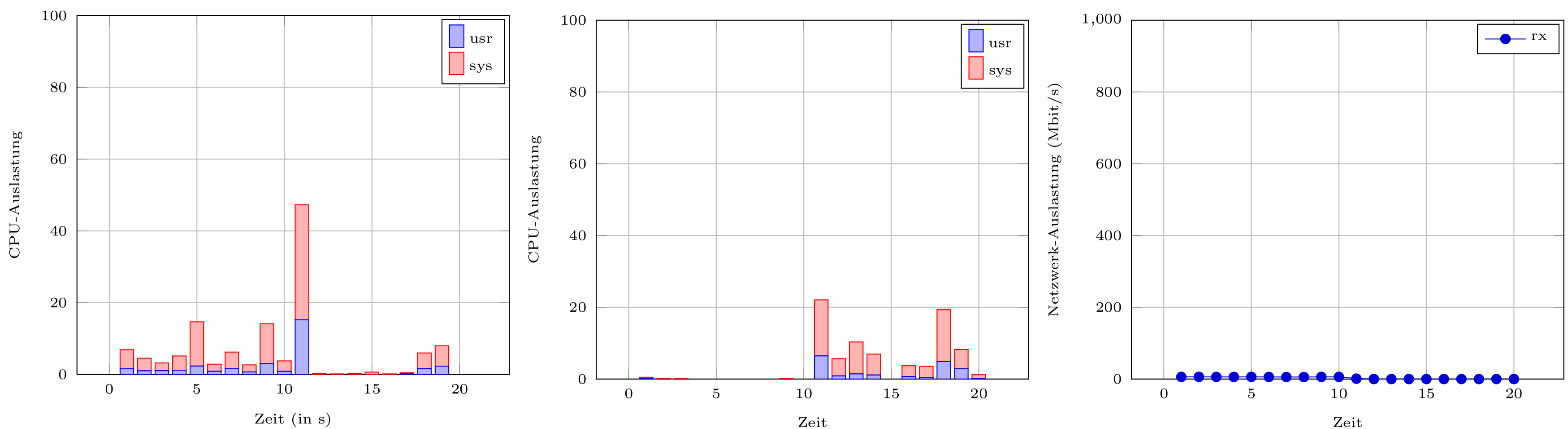


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 56: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *alpine* als Gast-Betriebssystem (Zwischenankunftszeit: 0.004 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

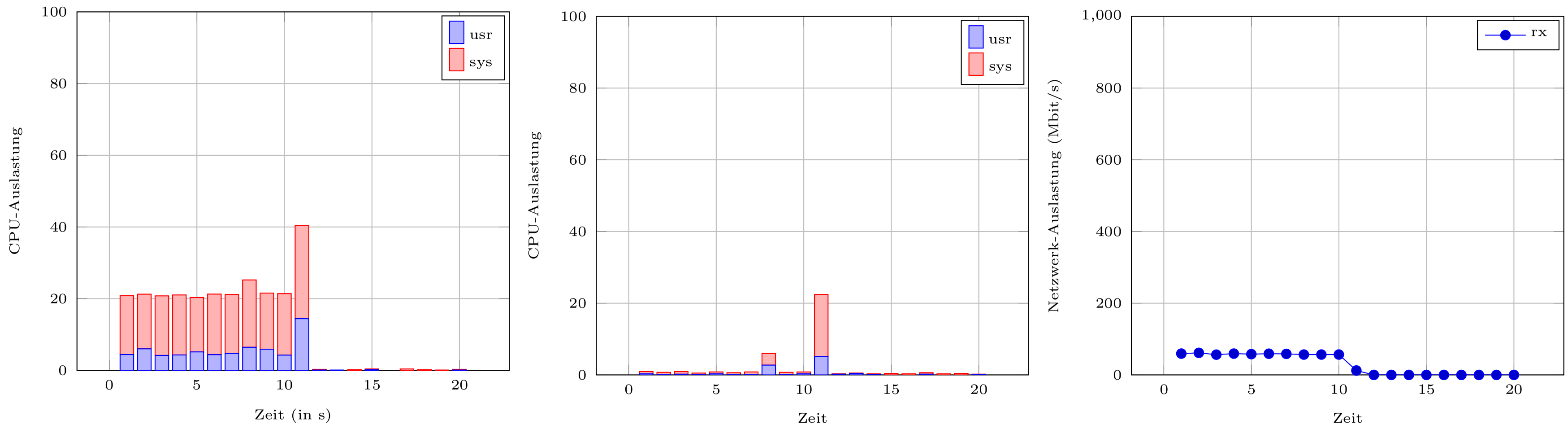


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

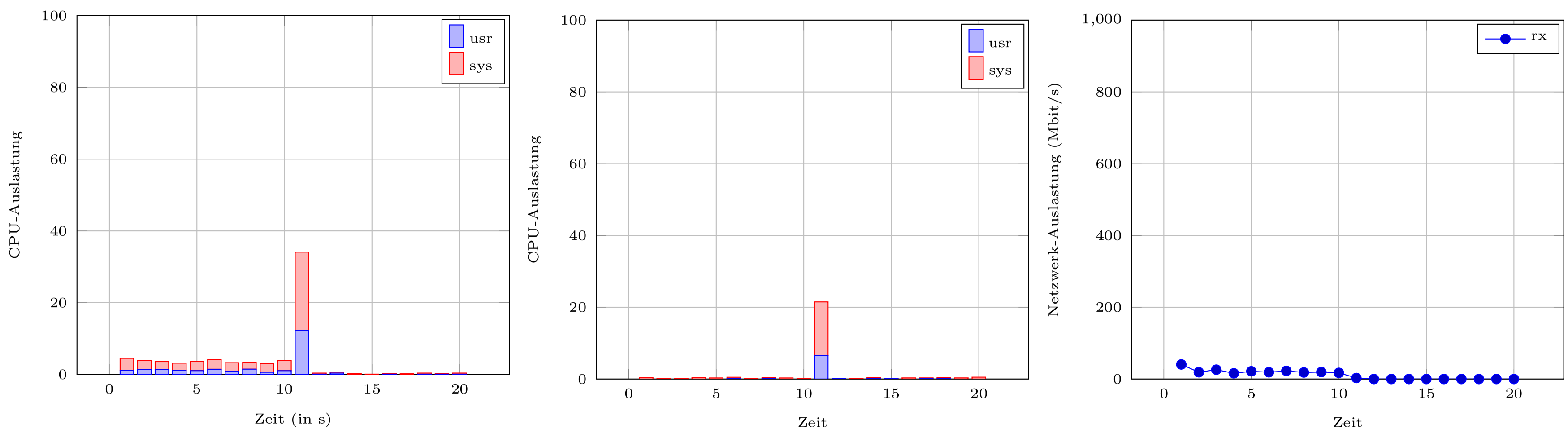


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 57: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ciao* als Gast-Betriebssystem (Zwischenankunftszeit: 0.004 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

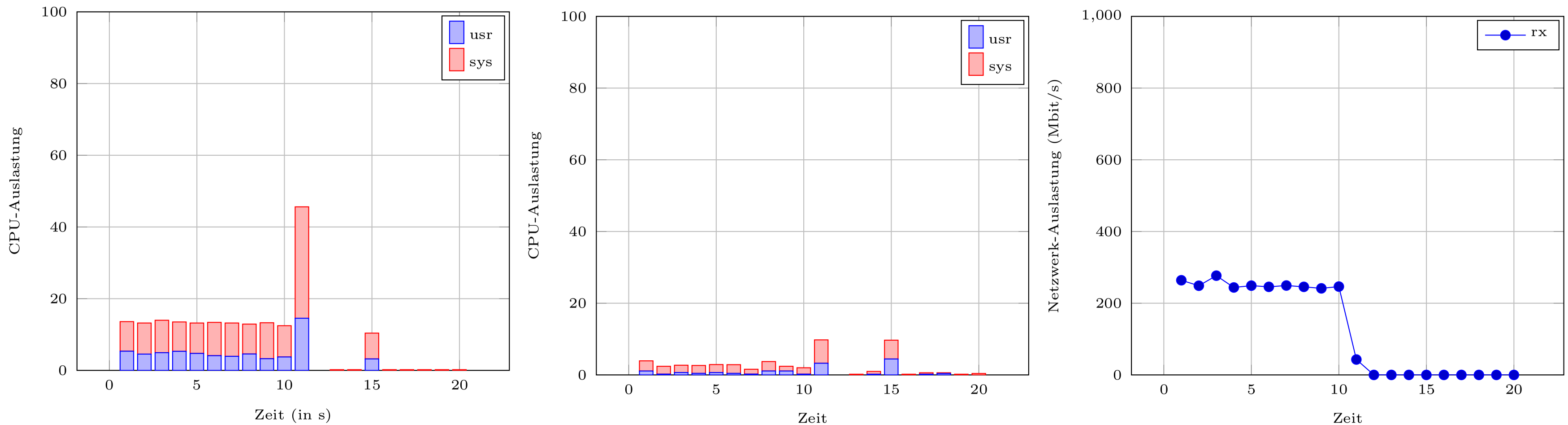


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

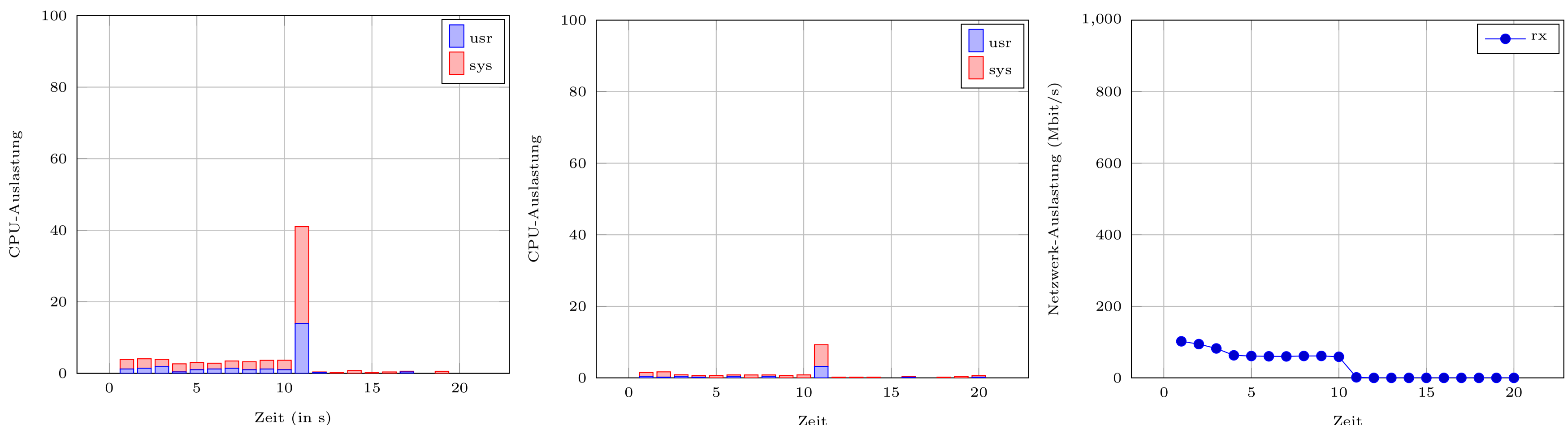


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 58: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *tinycore* als Gast-Betriebssystem (Zwischenankunftszeit: 0.004 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

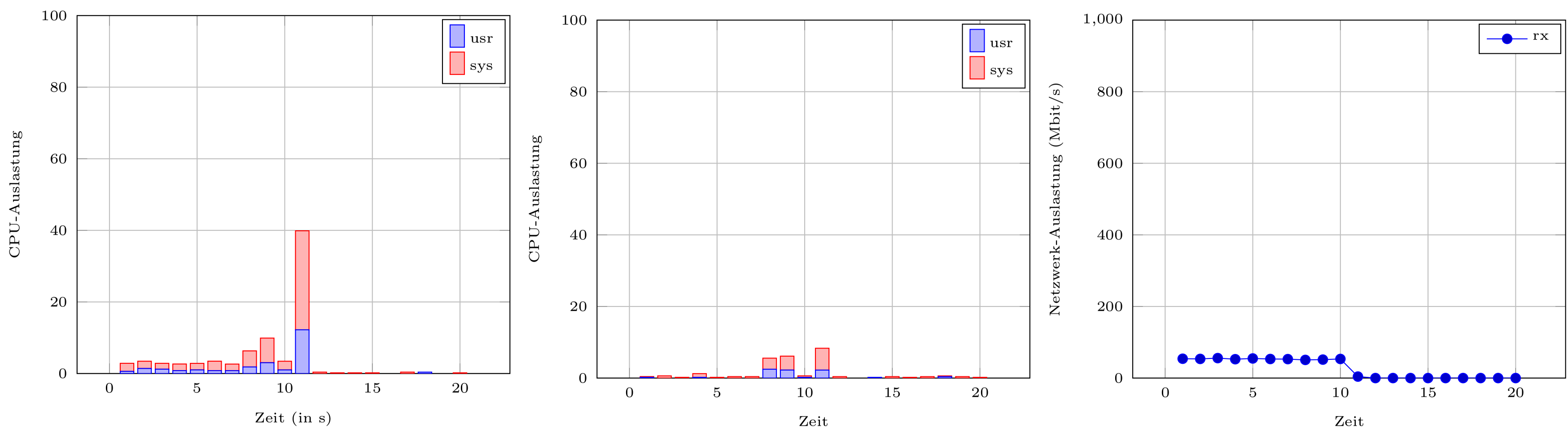


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

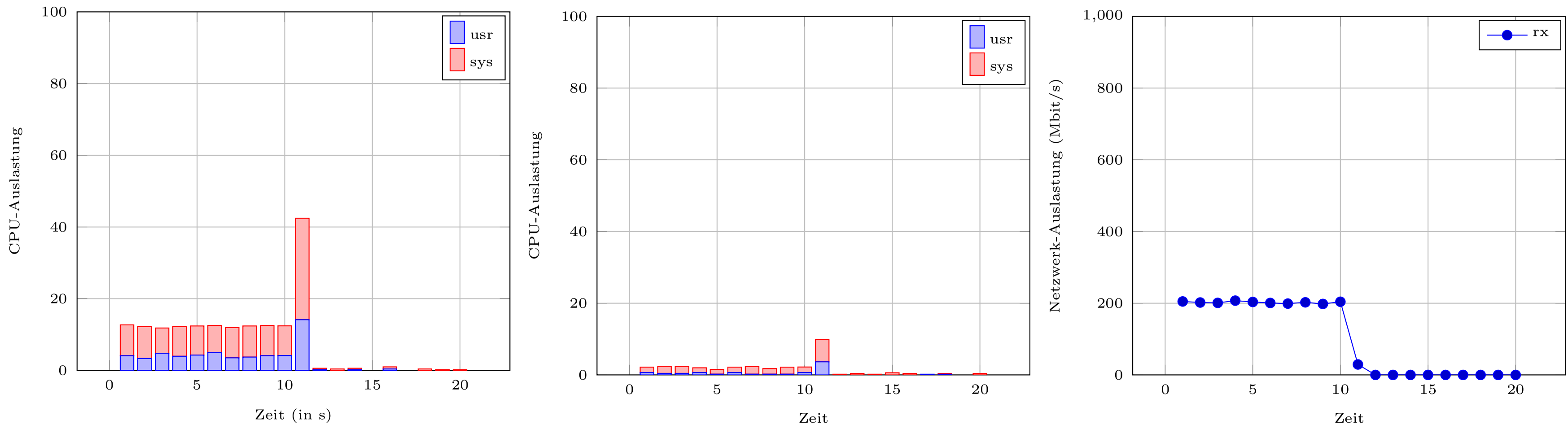


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 59: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ubuntu* als Gast-Betriebssystem (Zwischenankunftszeit: 0.004 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

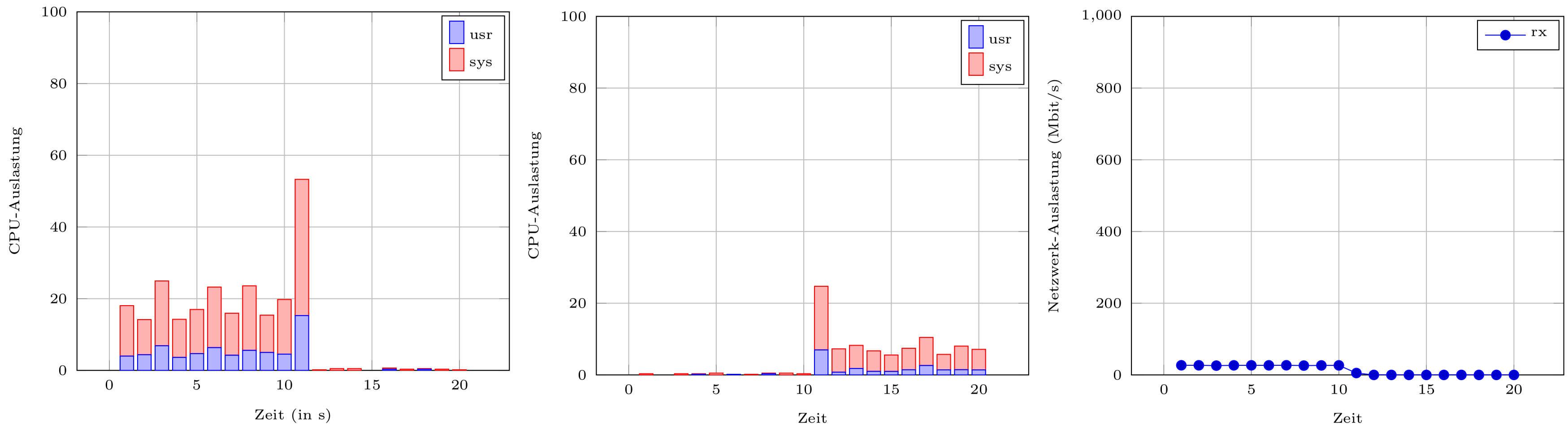


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

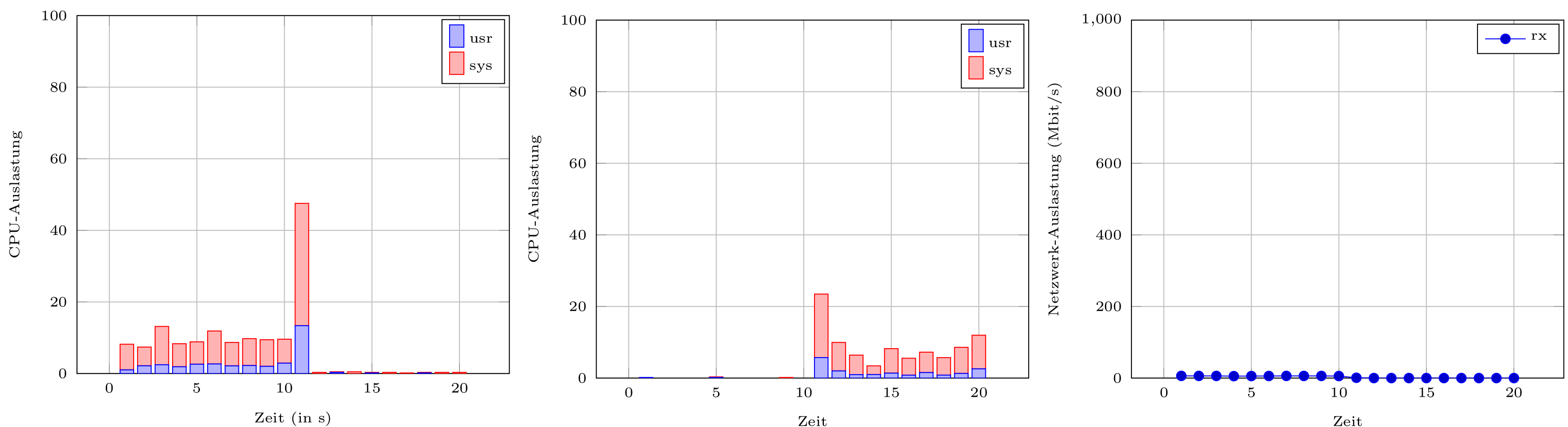


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 60: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *alpine* als Gast-Betriebssystem (Zwischenankunftszeit: 0.003 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

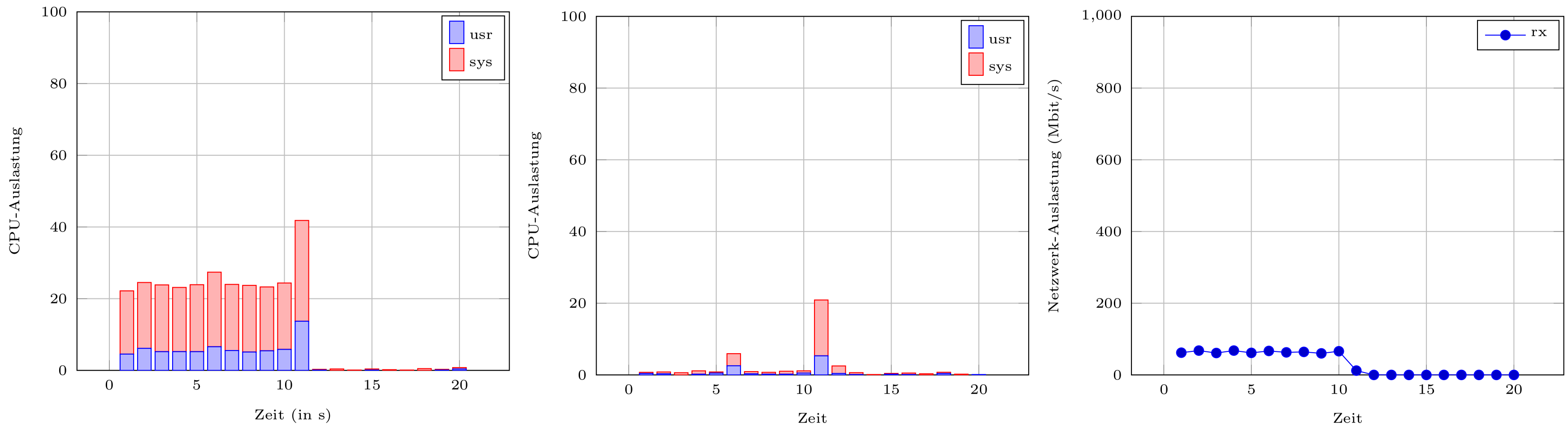


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

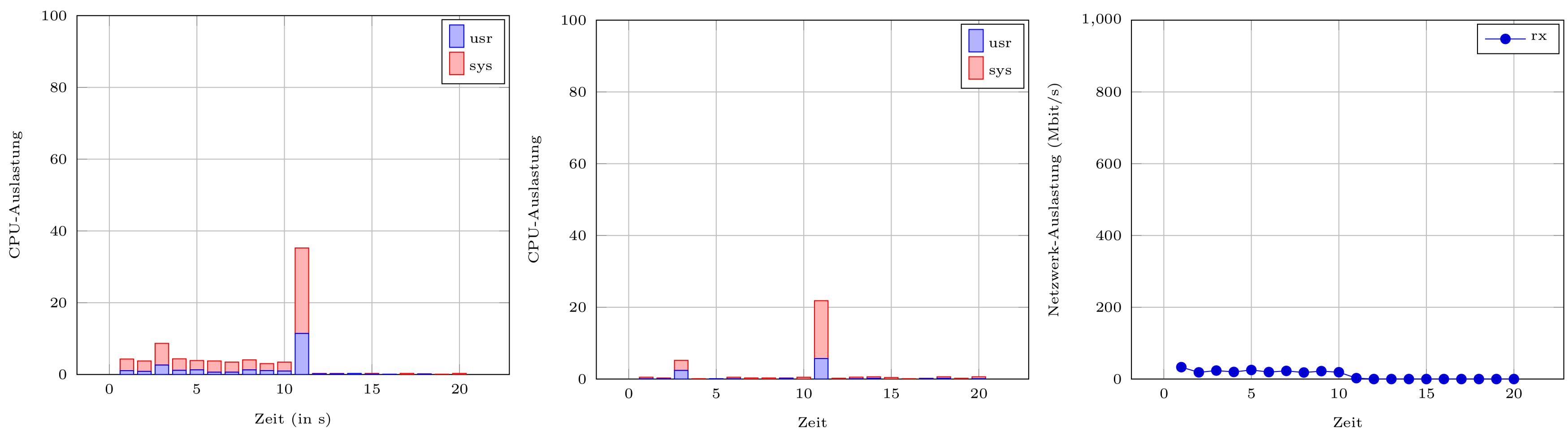


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 61: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ciao* als Gast-Betriebssystem (Zwischenankunftszeit: 0.003 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

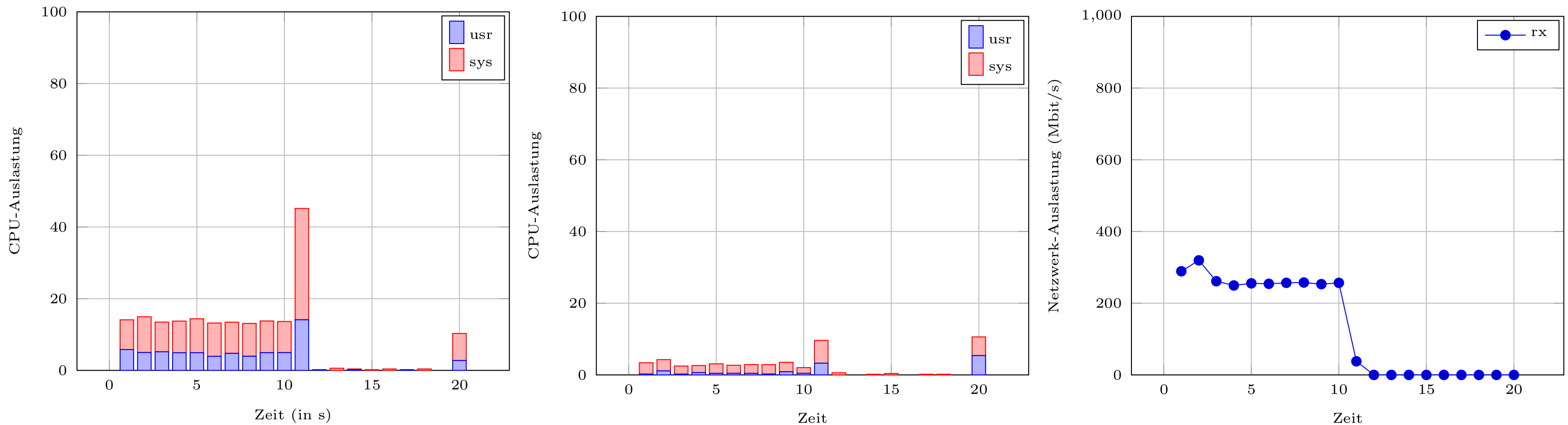


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

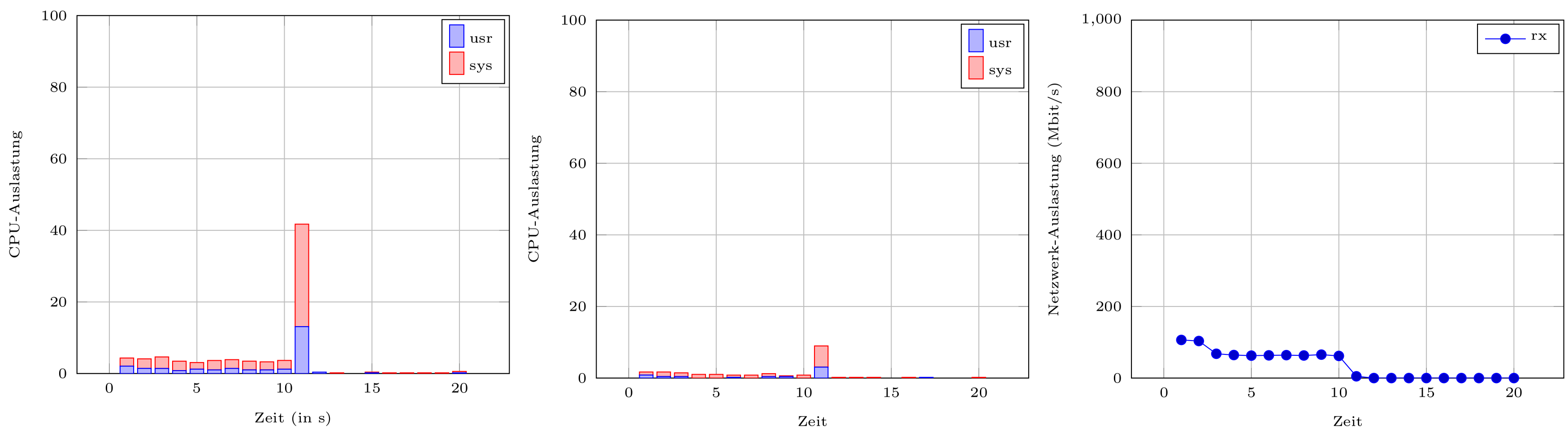


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 62: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *tinycore* als Gast-Betriebssystem (Zwischenankunftszeit: 0.003 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

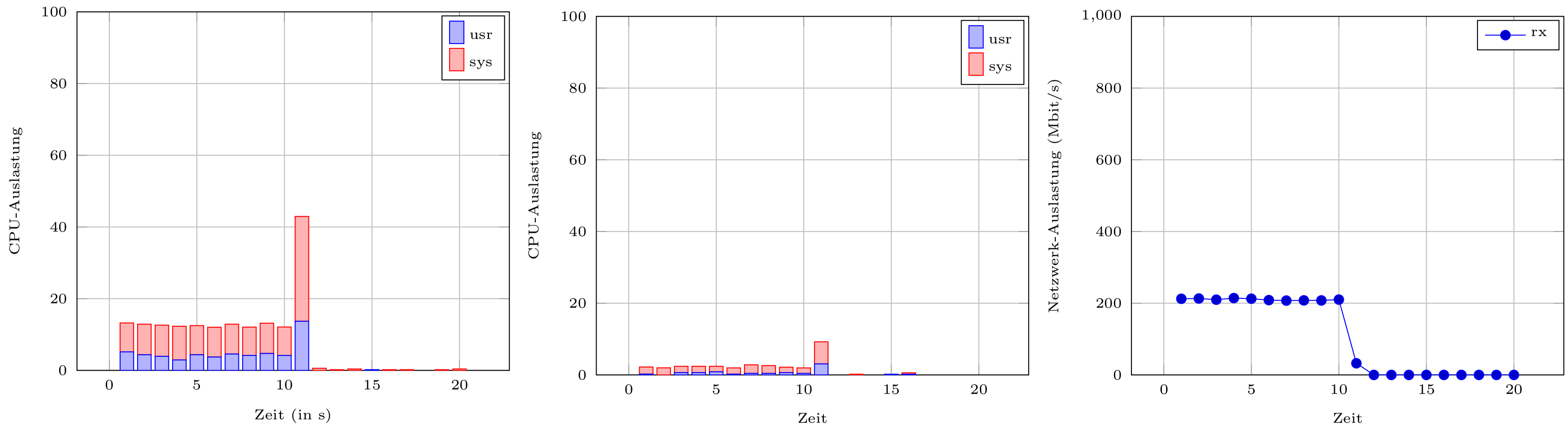


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

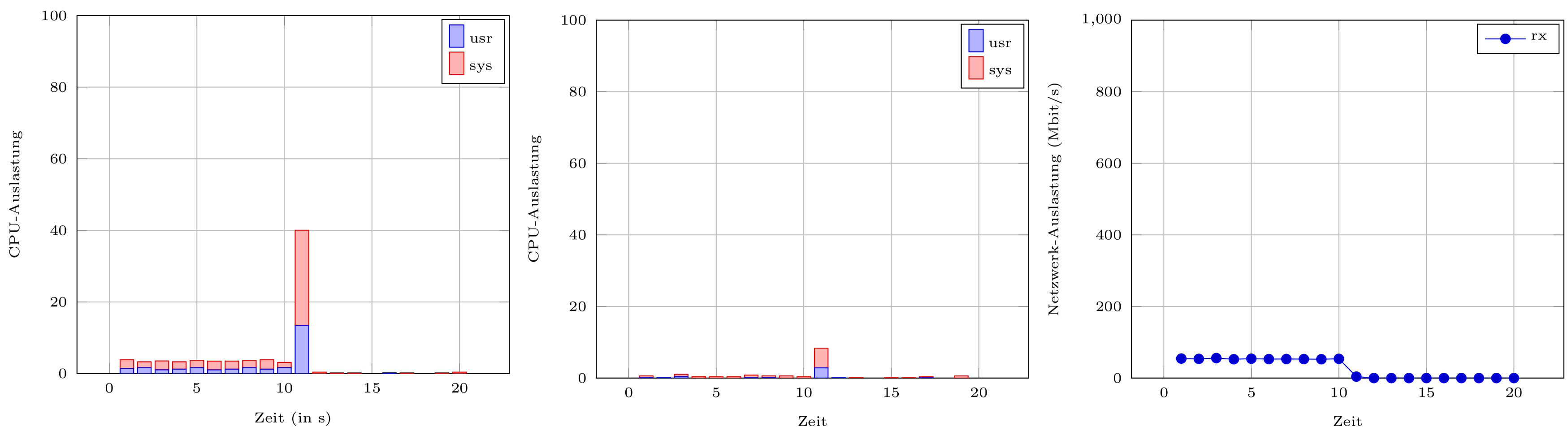


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 63: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ubuntu* als Gast-Betriebssystem (Zwischenankunftszeit: 0.003 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

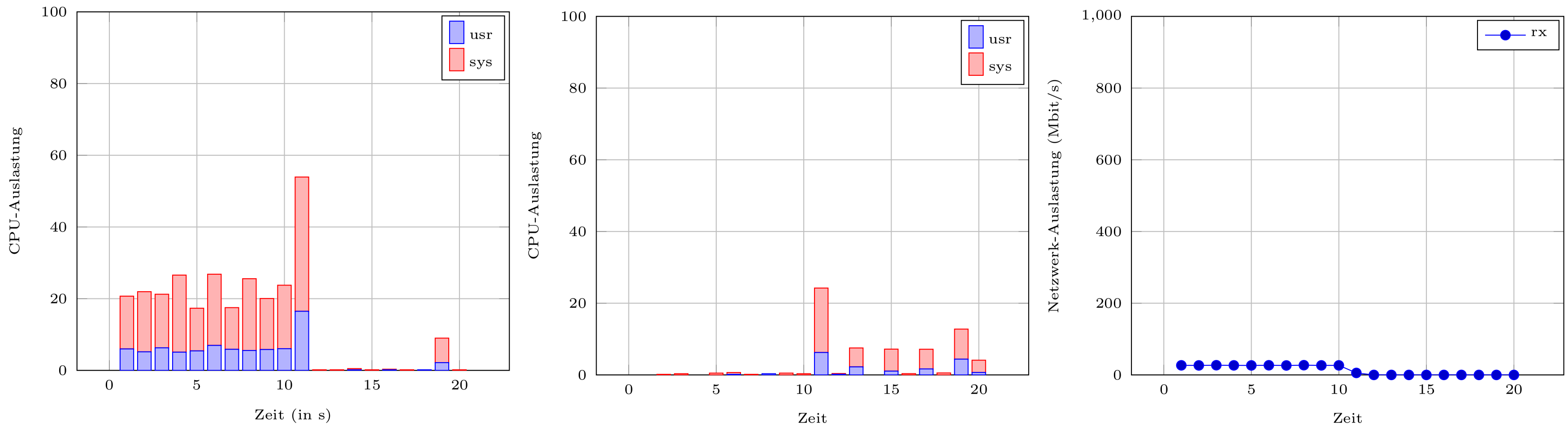


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

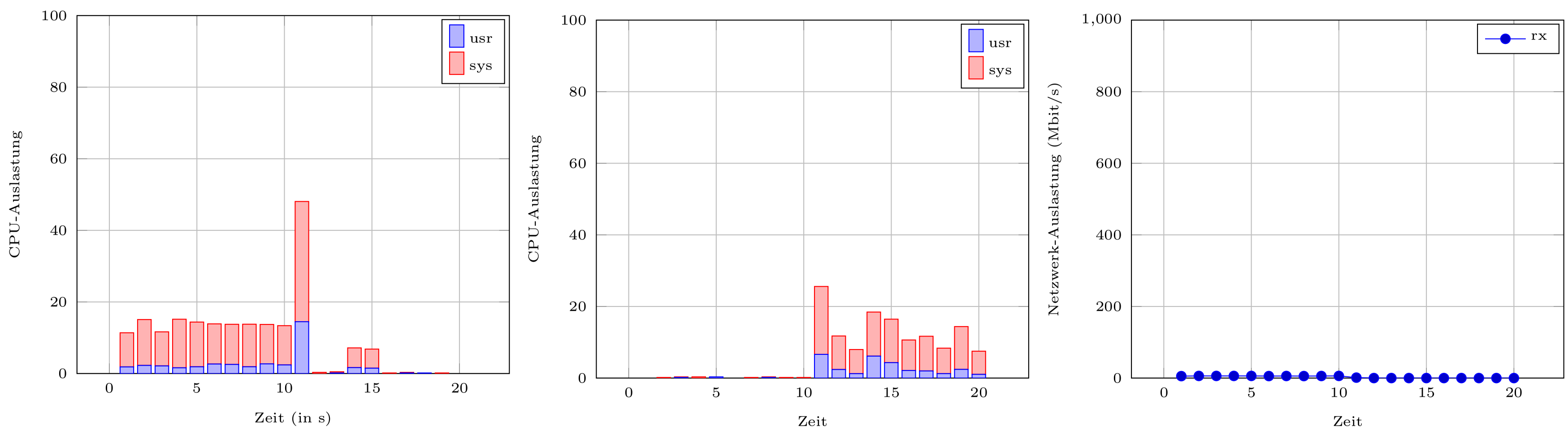


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 64: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *alpine* als Gast-Betriebssystem (Zwischenankunftszeit: 0.002 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

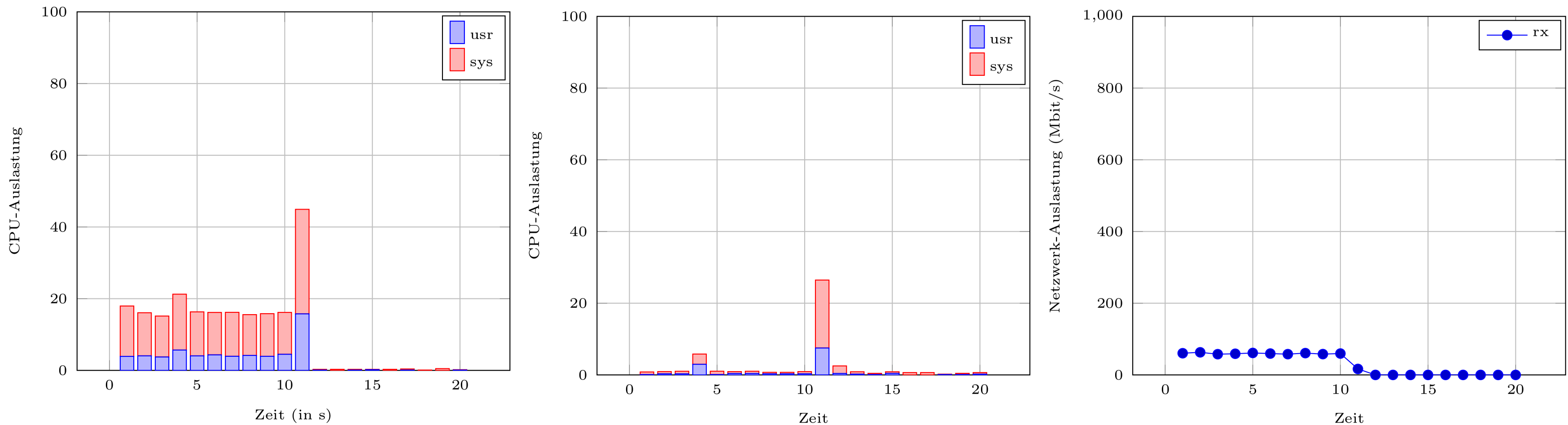


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

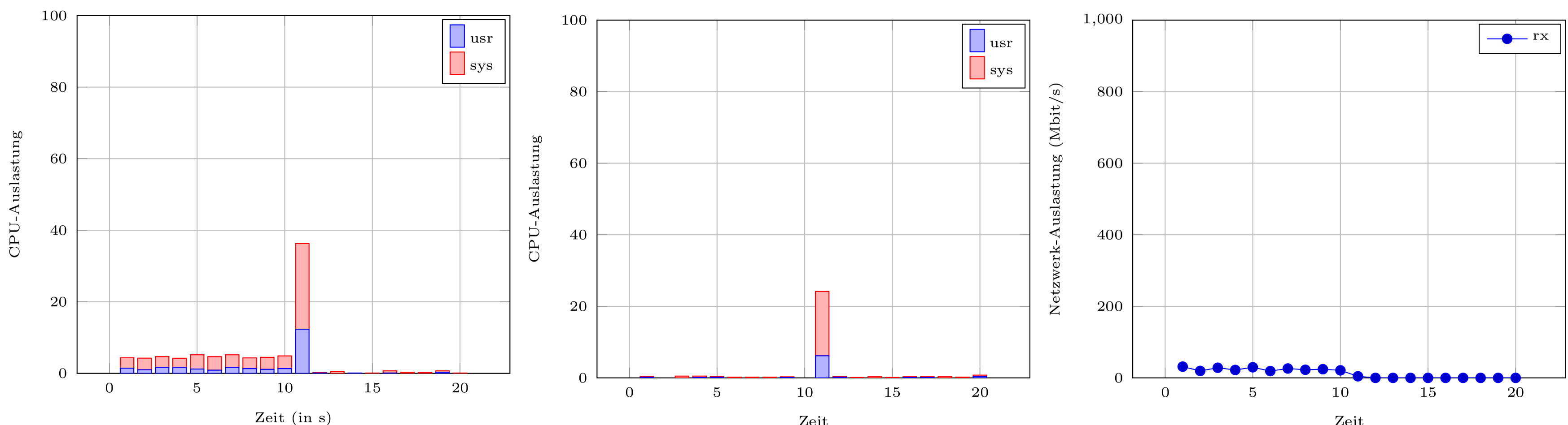


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 65: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ciao* als Gast-Betriebssystem (Zwischenankunftszeit: 0.002 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

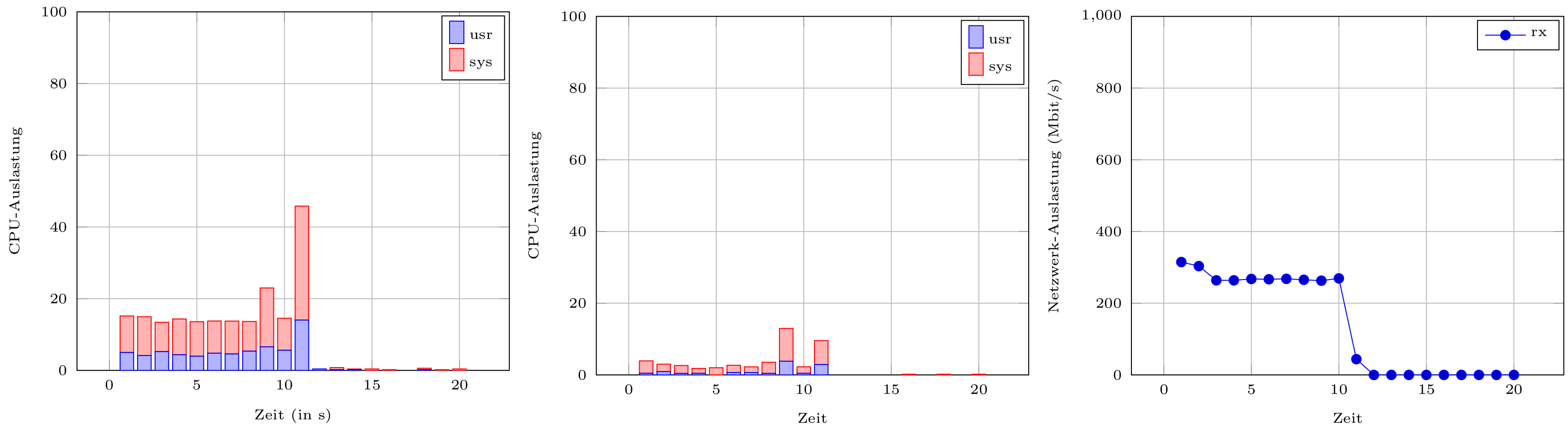


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

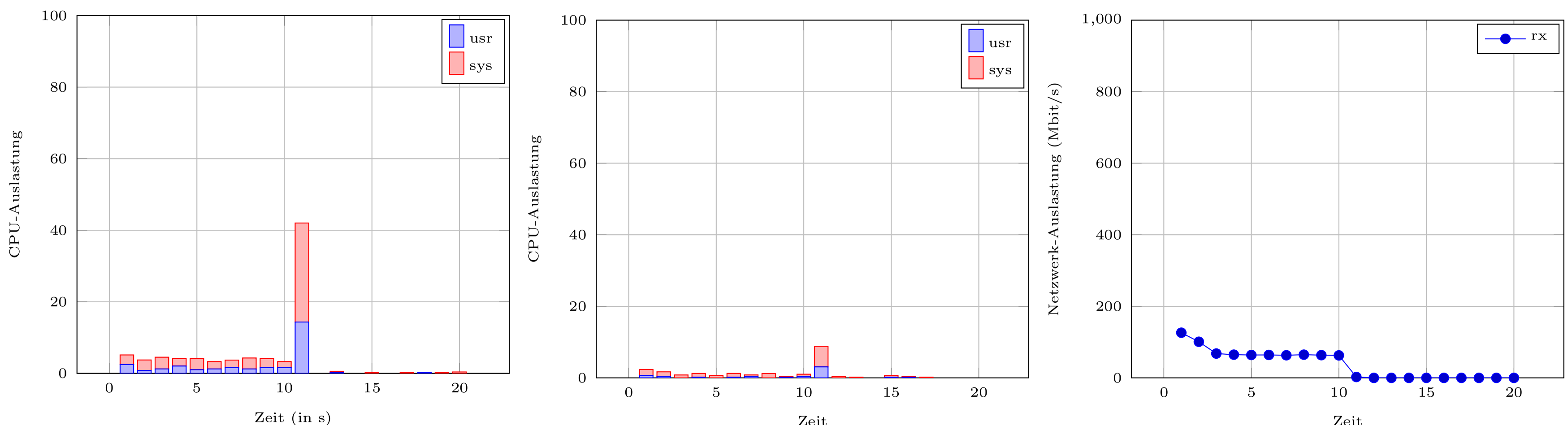


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 66: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *tinycore* als Gast-Betriebssystem (Zwischenankunftszeit: 0.002 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

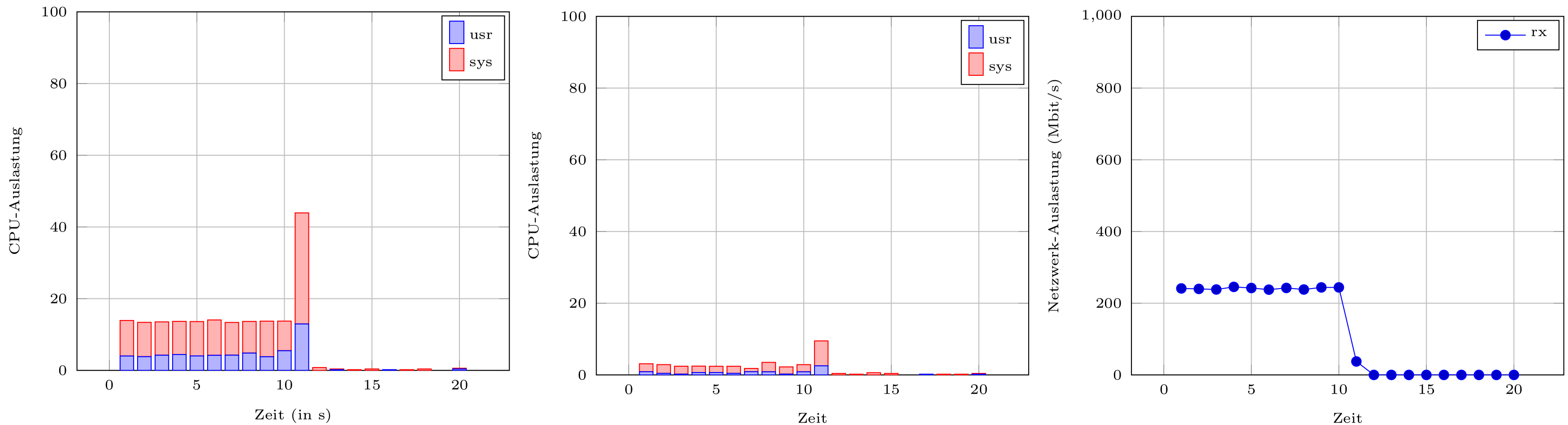


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

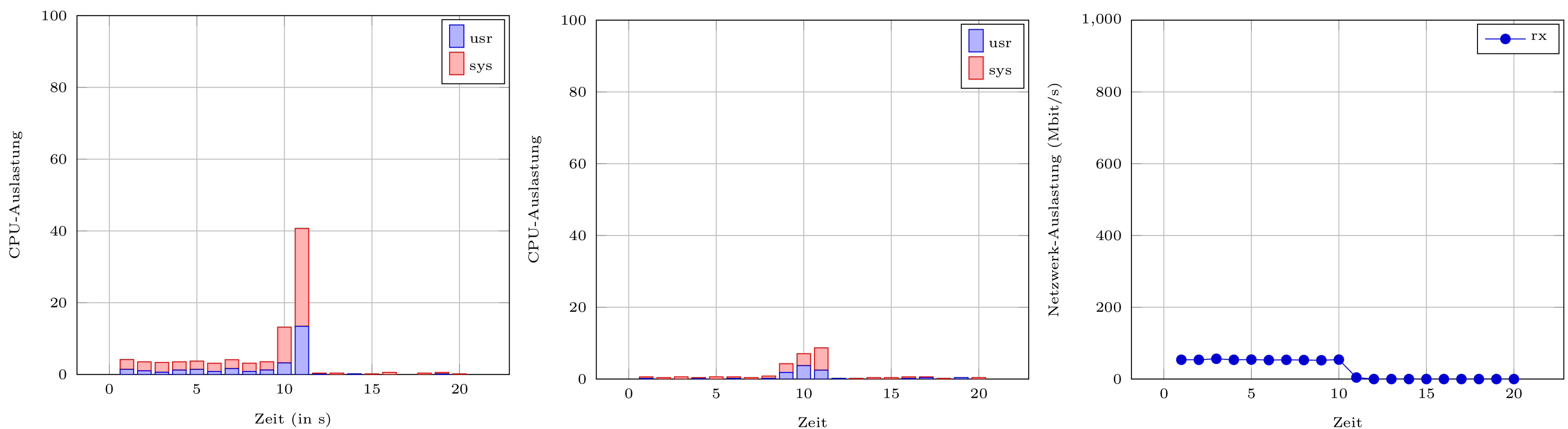


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 67: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ubuntu* als Gast-Betriebssystem (Zwischenankunftszeit: 0.002 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

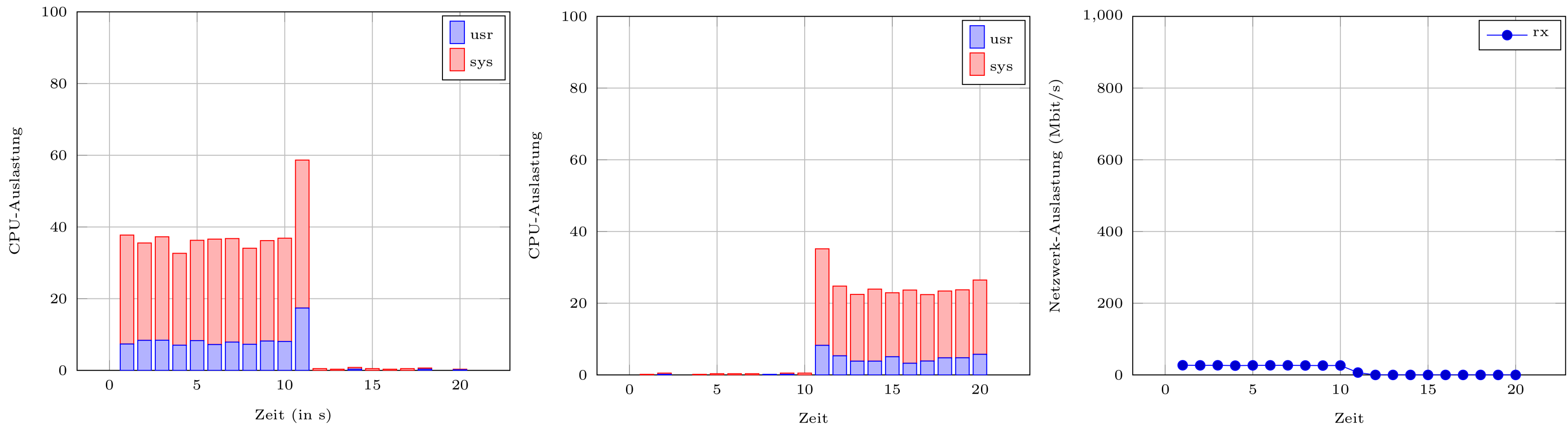


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

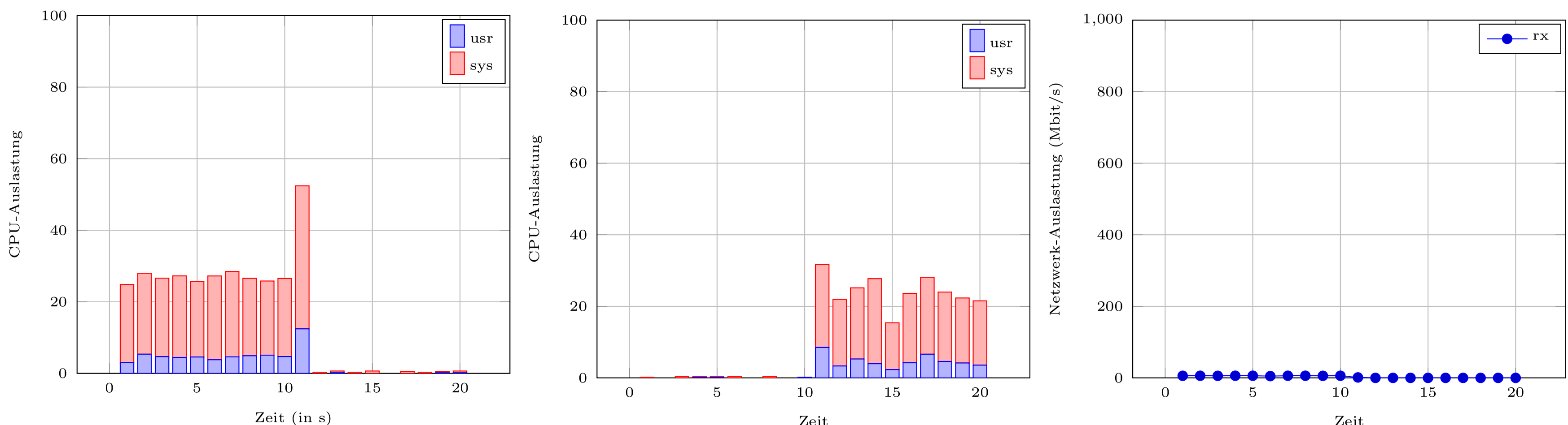


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 68: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *alpine* als Gast-Betriebssystem (Zwischenankunftszeit: 0.001 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

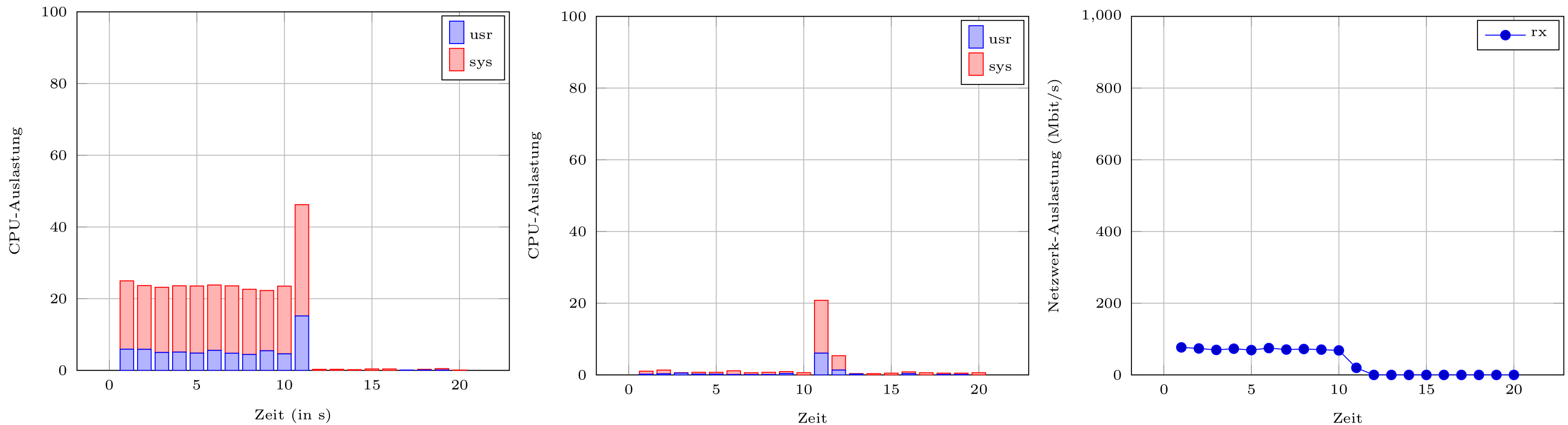


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

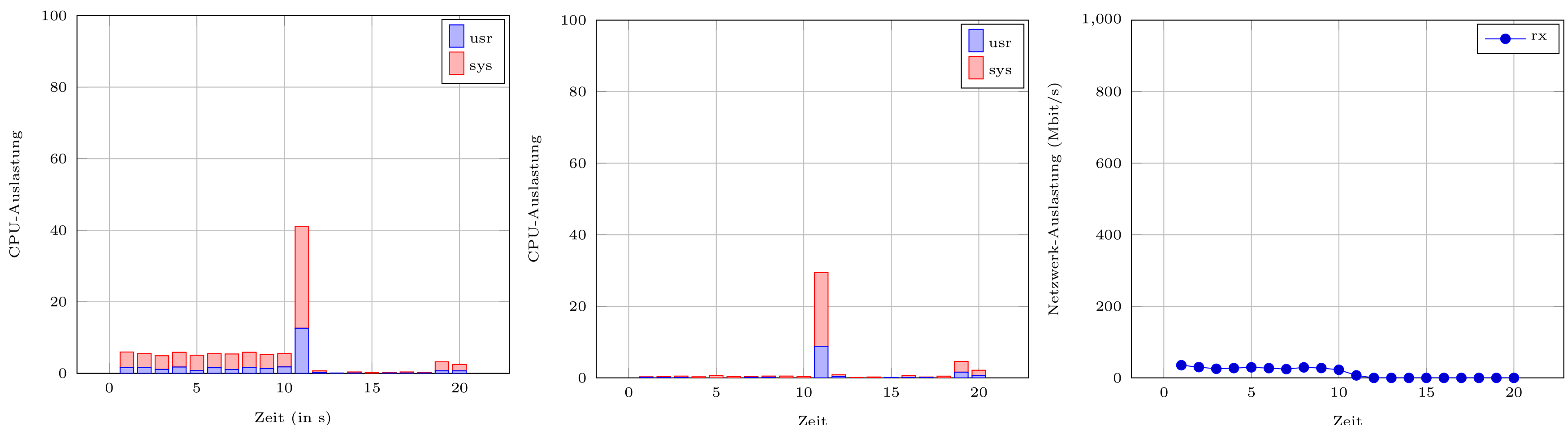


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 69: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ciao* als Gast-Betriebssystem (Zwischenankunftszeit: 0.001 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

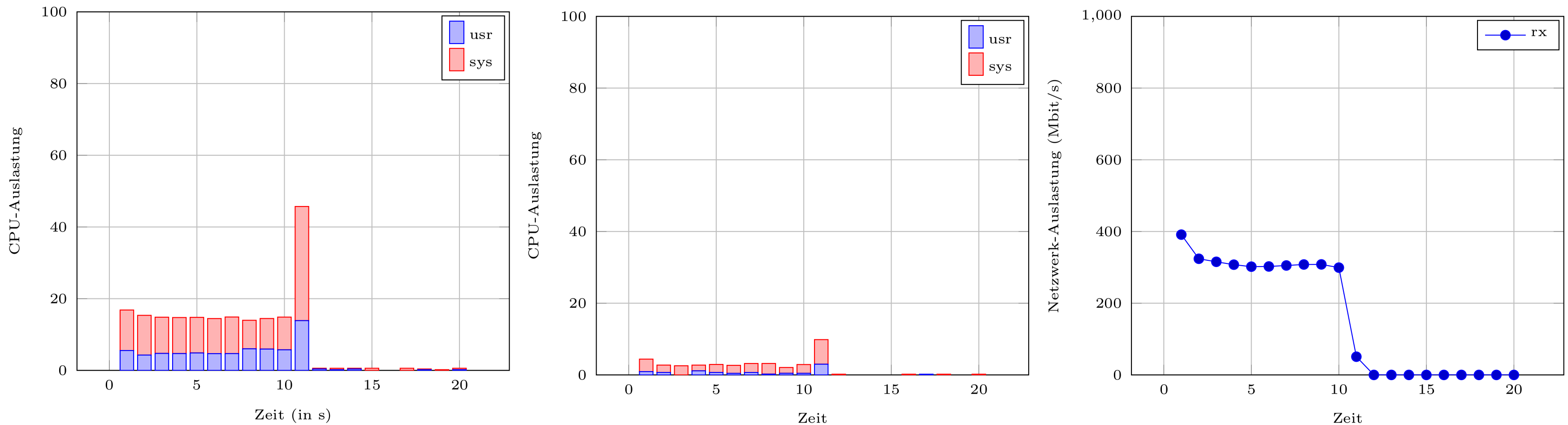


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

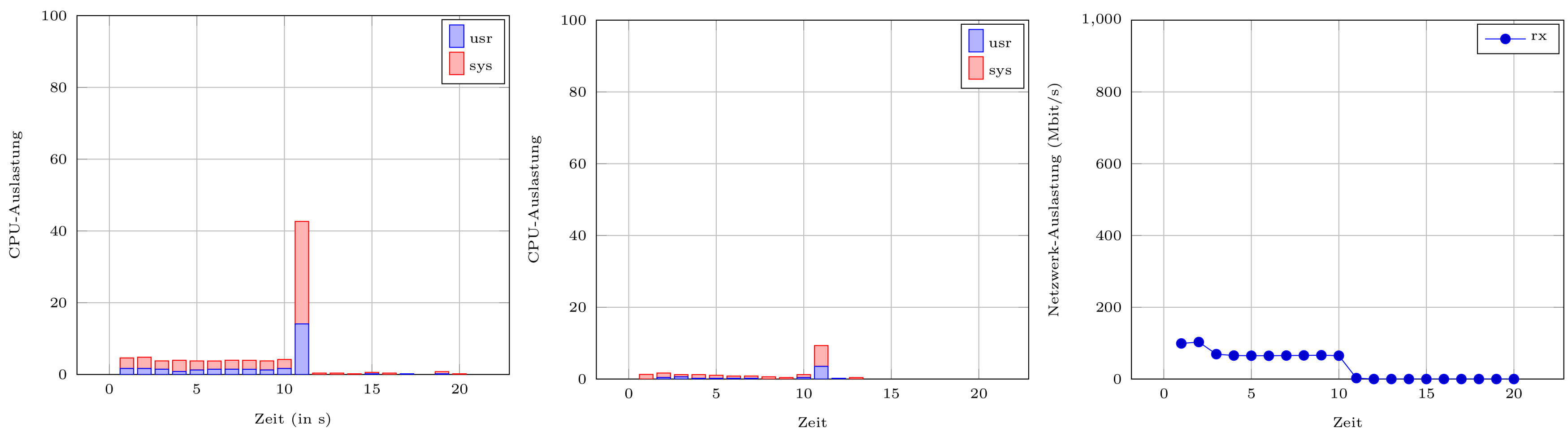


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 70: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *tinycore* als Gast-Betriebssystem (Zwischenankunftszeit: 0.001 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

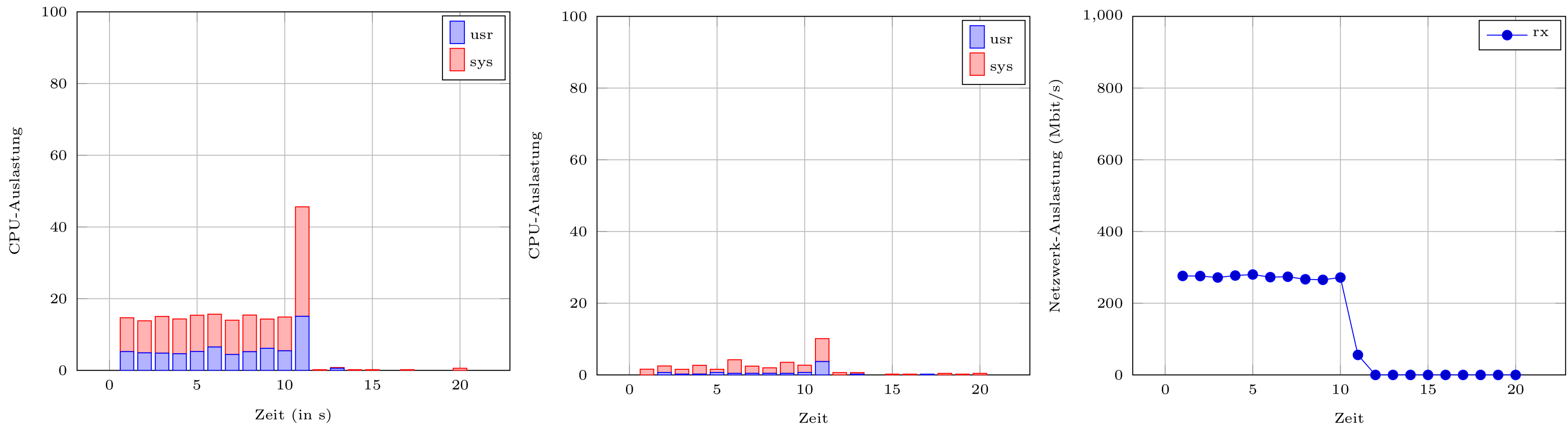


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

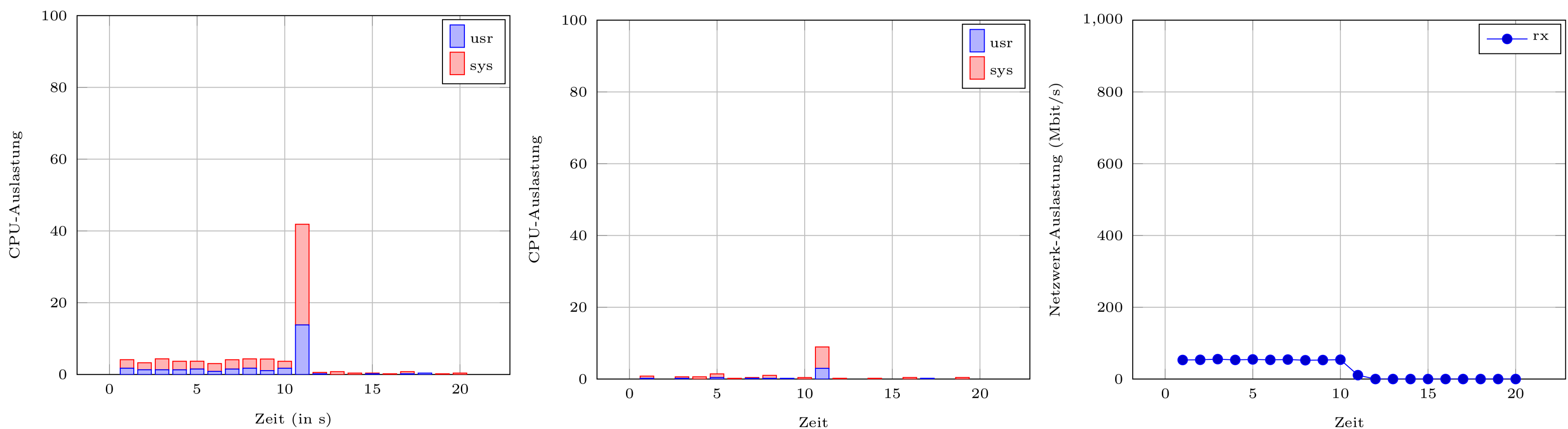


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 71: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ubuntu* als Gast-Betriebssystem (Zwischenankunftszeit: 0.001 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

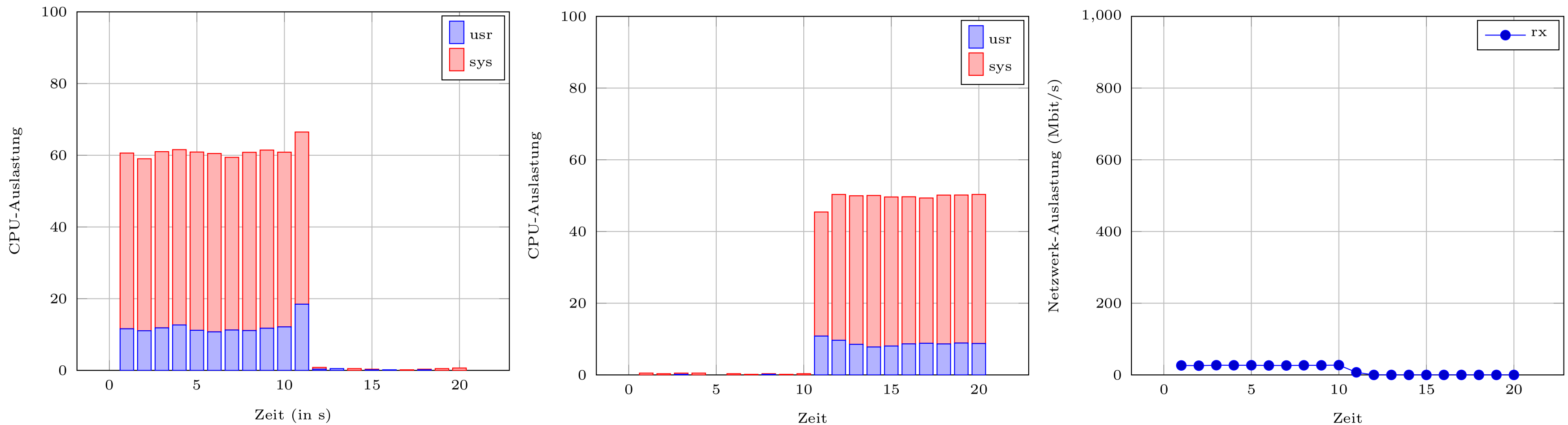


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

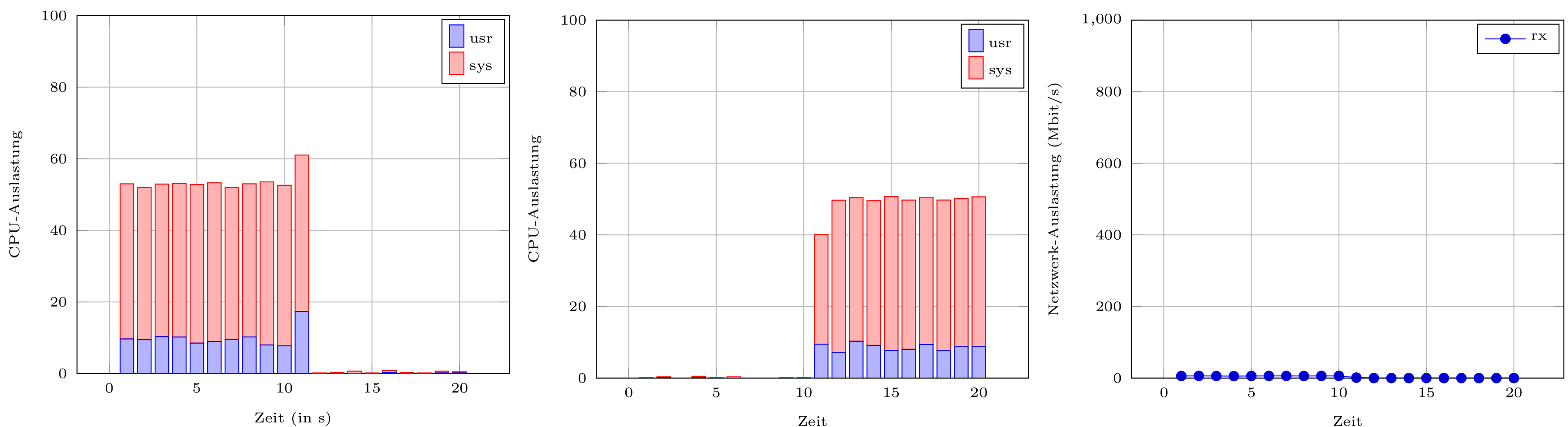


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 72: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *alpine* als Gast-Betriebssystem (Zwischenankunftszeit: 0.0005 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

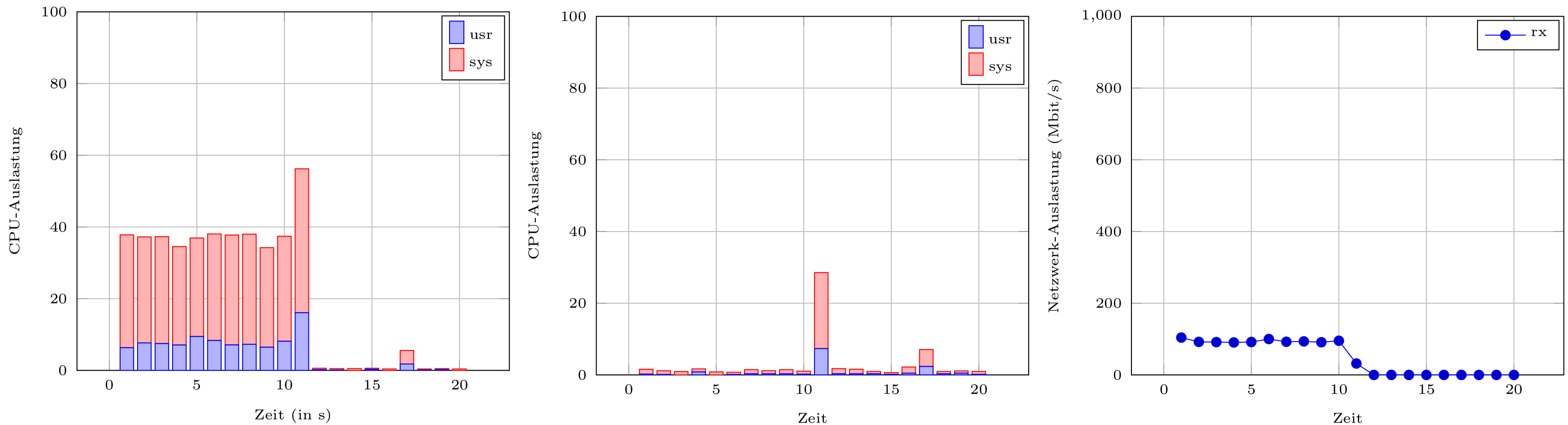


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

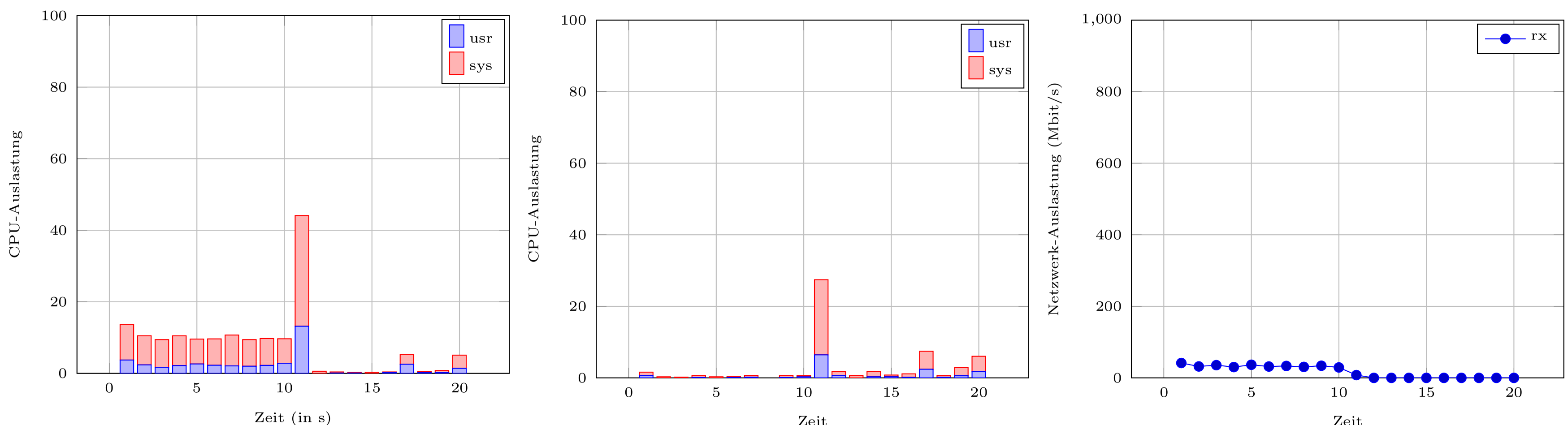


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 73: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ciao* als Gast-Betriebssystem (Zwischenankunftszeit: 0.0005 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

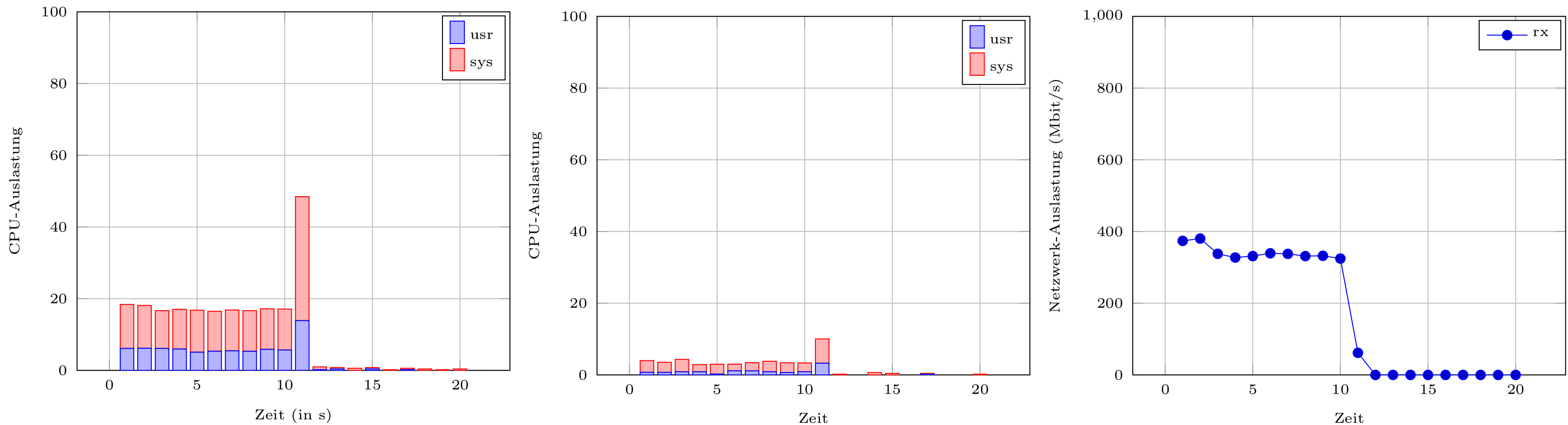


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

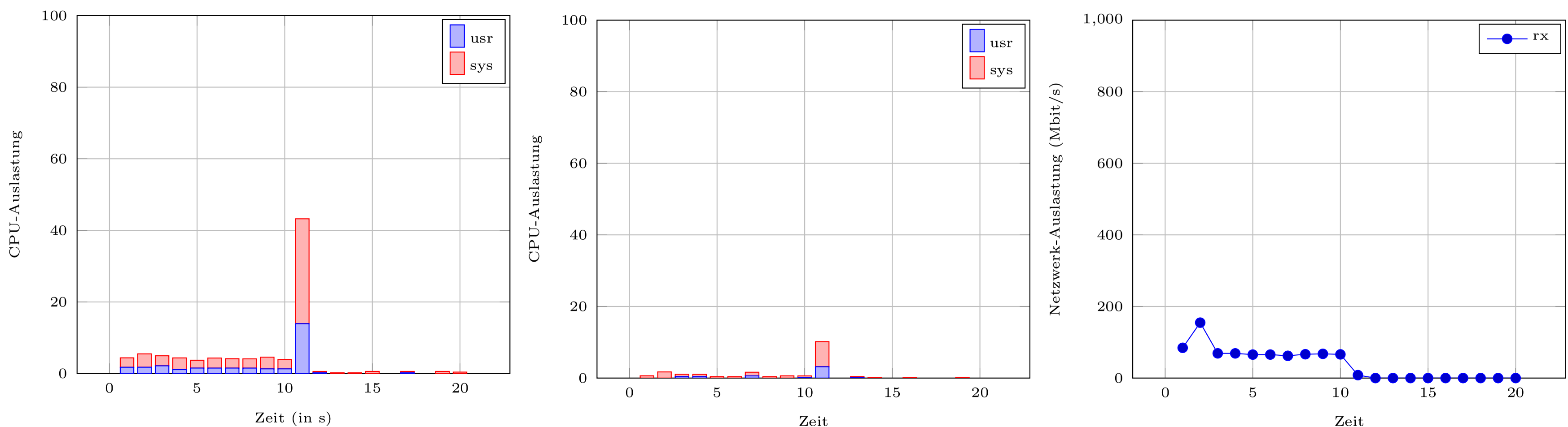


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 74: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *tinycore* als Gast-Betriebssystem (Zwischenankunftszeit: 0.0005 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

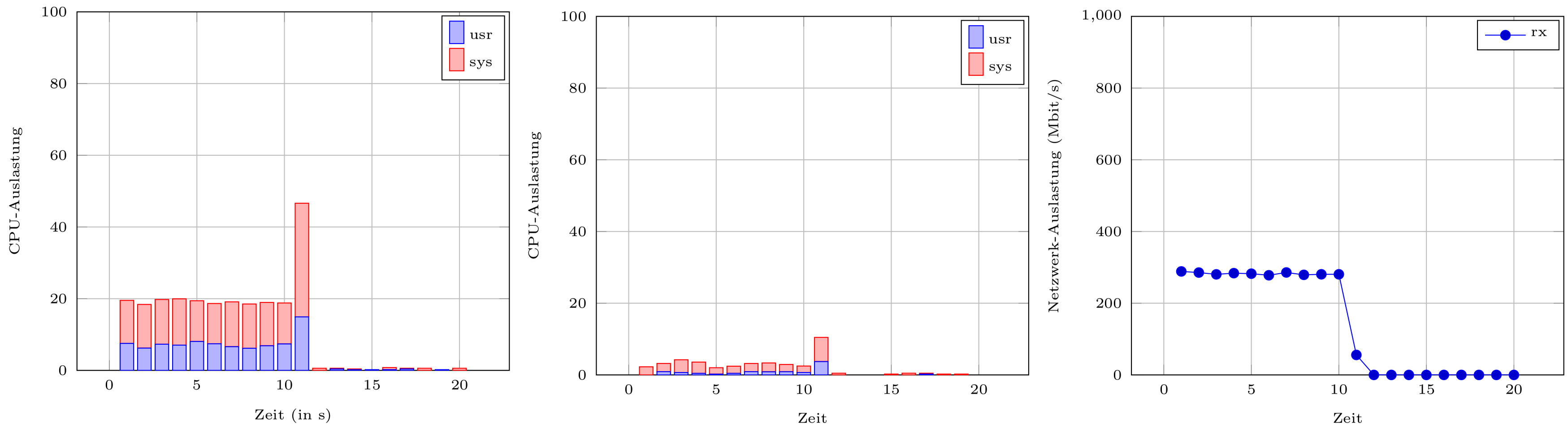


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

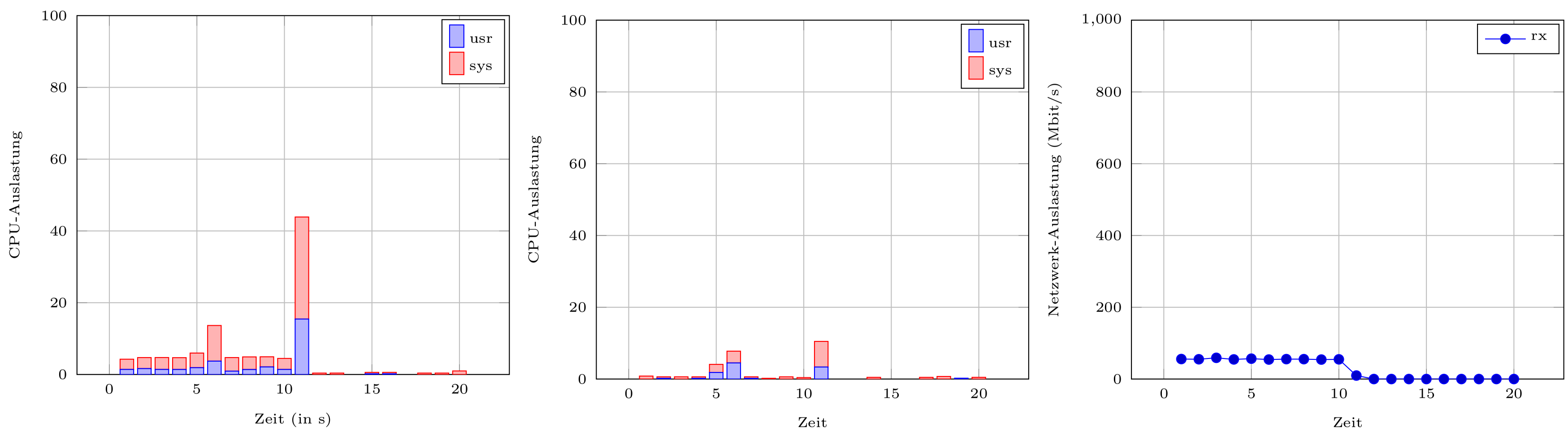


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 75: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ubuntu* als Gast-Betriebssystem (Zwischenankunftszeit: 0.0005 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

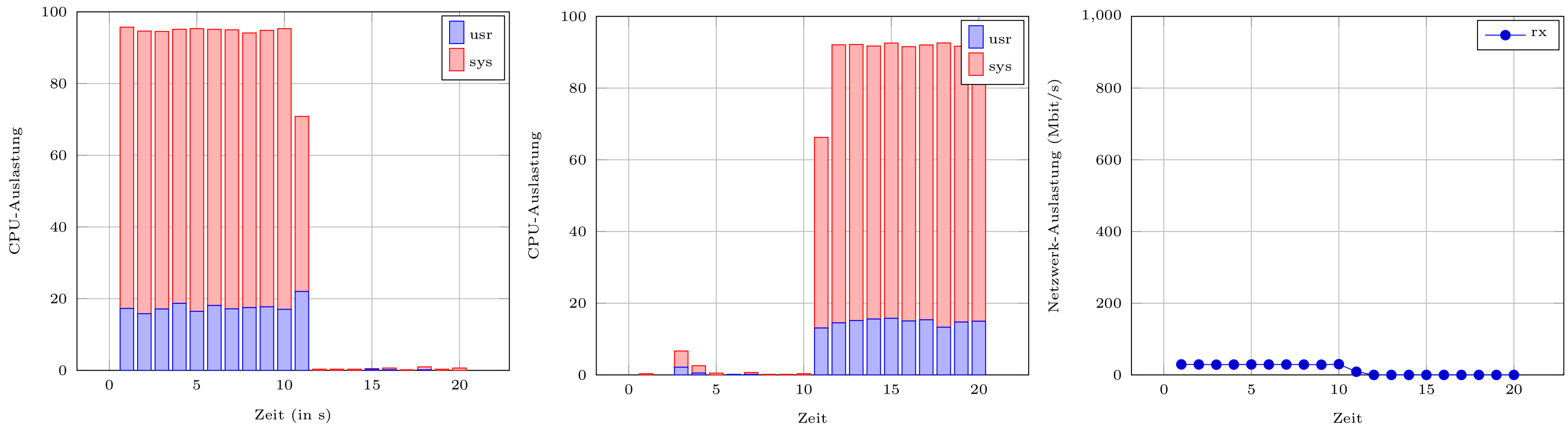


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

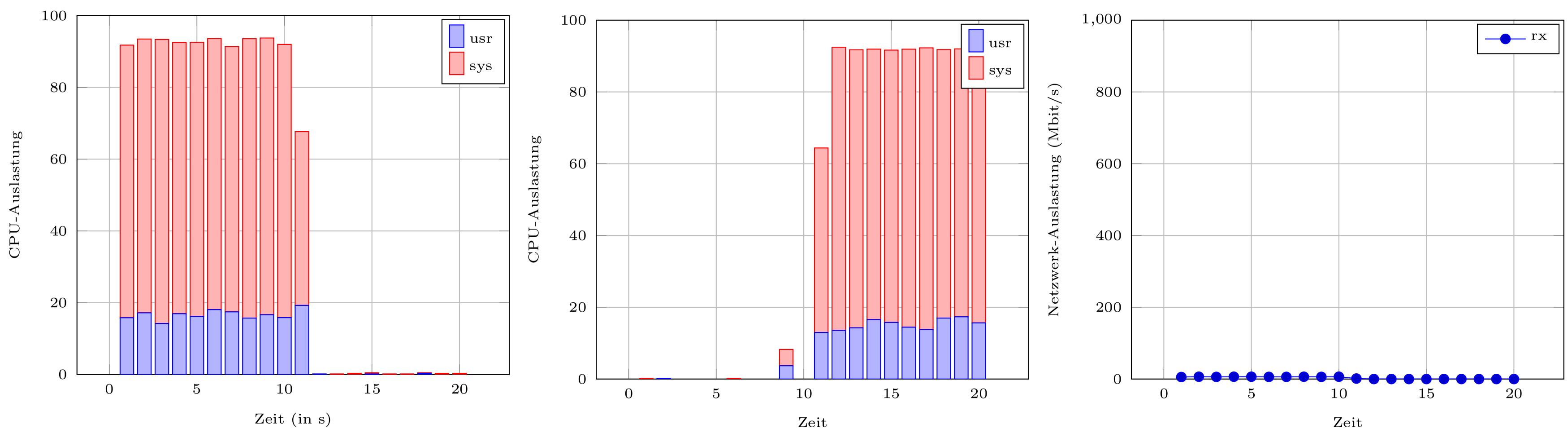


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 76: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *alpine* als Gast-Betriebssystem (Zwischenankunftszeit: 0.00025 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

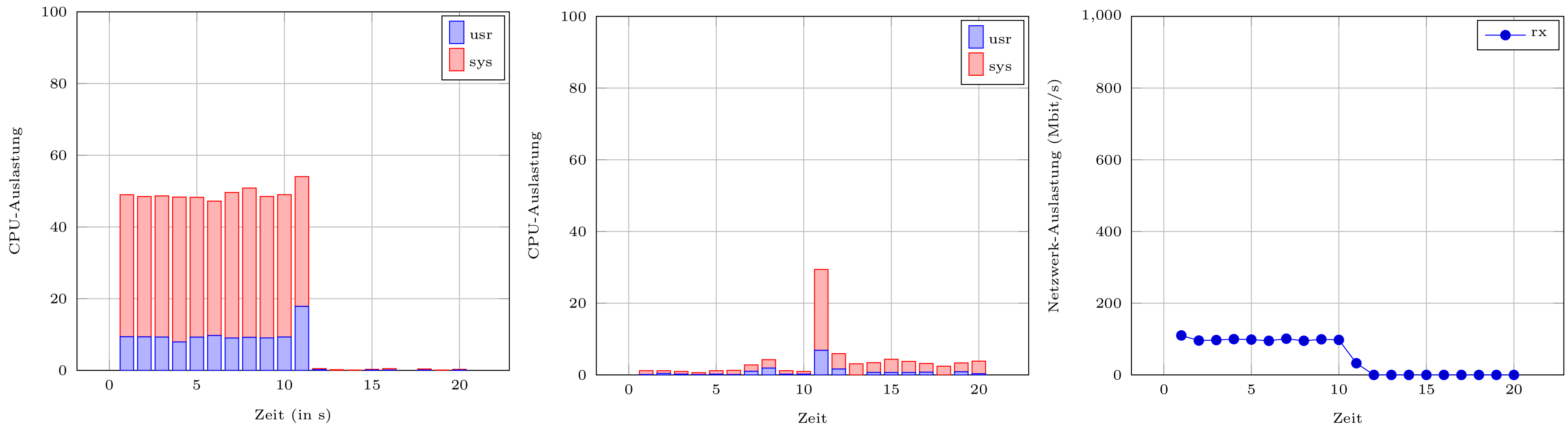


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

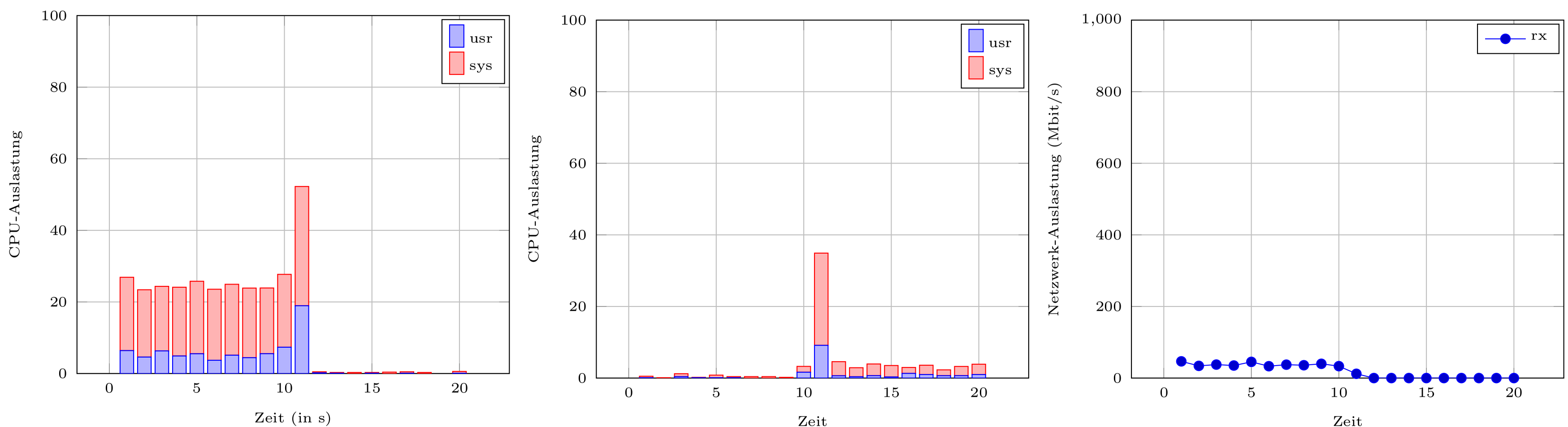


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 77: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ciao* als Gast-Betriebssystem (Zwischenankunftszeit: 0.00025 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

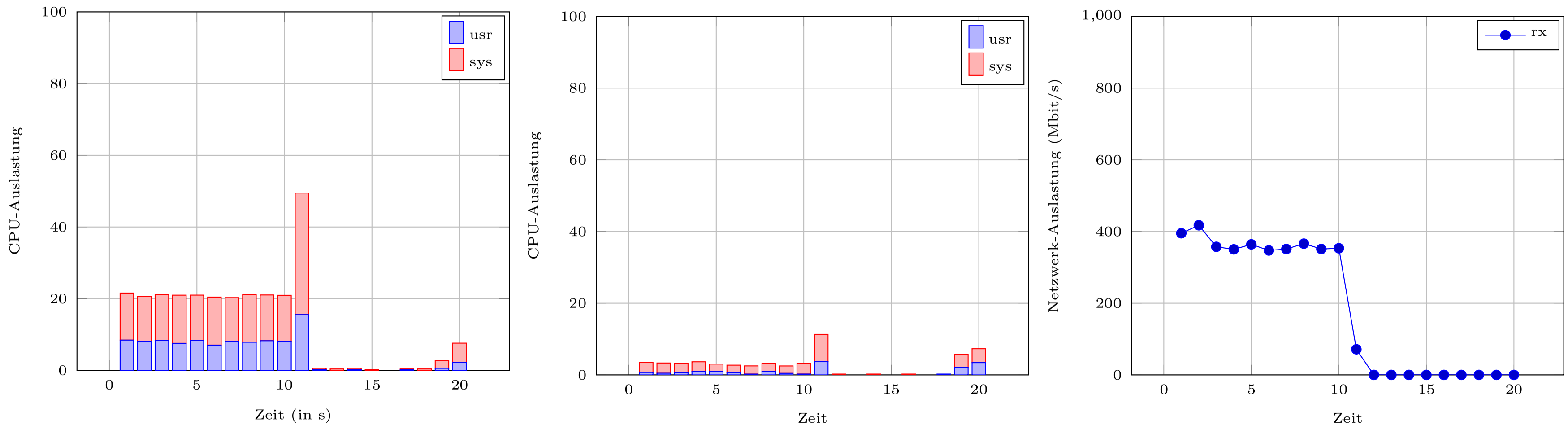


(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.

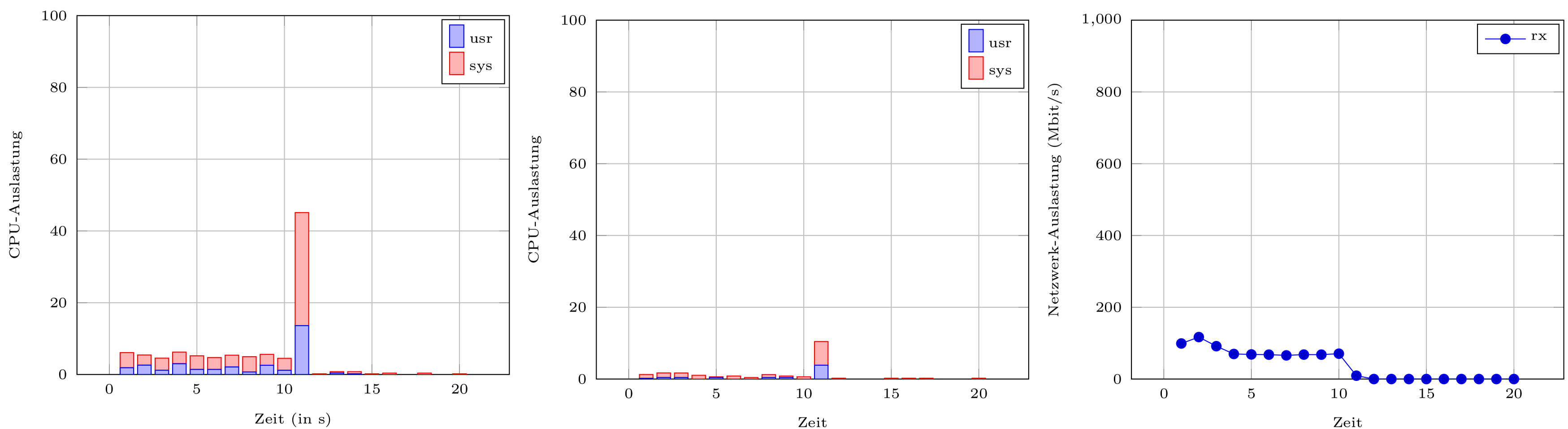


(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 78: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *tinycore* als Gast-Betriebssystem (Zwischenankunftszeit: 0.00025 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.



(a) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 40 ms.



(b) CPU-Auslastung der Dom0 des *Master*- bzw. *Backup*-Hosts (links bzw. mitte) sowie durch die Remus-Replikation benötigte Bandbreite (rechts) bei einem Replikationsintervall von 200 ms.

Abbildung 79: Vergleich der CPU- und Netzwerkauslastung der Dom0 bei der Verwendung von *ubuntu* als Gast-Betriebssystem (Zwischenankunftszeit: 0.00025 s). Der Failover findet zum Zeitpunkt $t = 11$ statt.

H. Literaturverweise

Literatur

[1] Edward A.Lee. *Cyber Physical Systems – Are Computing Foundations Adequate?* NSF Workshop on Cyber Physical Systems: Research Motivation, Techniques and Roadmap, Austin, USA, October 2006.

[2] Proxmox Server Solutions GmbH. http://www.proxmox.com, last consulted on 08.09.2013.

[3] RT-Xen: Real-Time Virtualization Based on Compositional Scheduling. https://sites.google.com/site/realtimexen/, last consulted on 08.03.2014.

[4] Kemari Project. http://www.osrg.net/kemari/, last consulted on 08.09.2013.

[5] Proxmox Server Solutions GmbH, Proxmox Developer Documentation. http://pve.proxmox.com/wiki/Developer_Documentation, last consulted on 08.09.2013.

[6] Citrix Systems, The three levels of high availability – Balancing priorities and cost. http://www.freexenserver.com/documents/3_levels_high_avail_WP_US.pdf, last consulted on 08.09.2013.

[7] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. *System architecture evaluation using modular performance analysis: a case study.* Int. J. Softw. Tools Technol. Transf., 8(6):649–667, October 2006.

[8] Simon Perathoner. *Evaluation and comparison of performance analysis methods for distributed embedded systems.* Master's thesis, Swiss Federal Institute of Technology (ETH) Zurich, Switzerland and Politecnico di Milano, Italy, April 2006.

[9] Ernesto Wandeler. *Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems.* PhD thesis, Swiss

Federal Institute of Technology (ETH) Zurich, Switzerland, September 2006.

[10] Simon Perathoner. *Modular Performance Analysis of Embedded Real-Time Systems: Improving Modeling Scope and Accuracy.* PhD thesis, Swiss Federal Institute of Technology (ETH) Zurich, Switzerland, April 2011.

[11] S. Chakraborty, S. Kunzli, and L. Thiele. *A general framework for analysing system properties in platform-based embedded system designs.* In Design, Automation and Test in Europe Conference and Exhibition, 2003, pages 190–195, 2003.

[12] Boguslaw Jablkowski and Olaf Spinczyk. *Continuous Performance Analysis of Fault-Tolerant Virtual Machines.* In Proceedings of the 1st GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)

[13] Charles David Graziano. *A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project.* Graduate Theses and Dissertations 2011. Paper 12215, Iowa State University, USA.

[14] Arten von Hypervisoren, `http://upload.wikimedia.org/wikipedia/commons/e/e1/Hyperviseur.png`, last consulted on 30.08.2013

[15] David Chisnall. *The Definitive Guide to the Xen Hypervisor.* Prentice Hall Open Source Software Development Series. 2008 Pearson Education, Inc.

[16] Chris Takemura and Luke S. Crawford. *The book of Xen : a practical guide for the system administrator.* 2010 No Starch Press, Inc.

[17] Choice of Toolstacks. `http://wiki.xenproject.org/wiki/Choice_of_Toolstacks`, last consulted on 01.08.2013

[18] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. *Xen and the Art of Virtualization.* In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). ACM, New

York, NY, USA, 164-177.

[19] Lars Kurth - Xen Project Community Manager. *10 Years of Xen and beyond...* LinuxCon Japan 13, May 30, 2013.

[20] REMUS: Transparent high availability for Xen. http://nss.cs.ubc.ca/remus, last consulted on 01.09.2013.

[21] Cully, Brendan and Lefebvre, Geoffrey and Meyer, Dutch and Feeley, Mike and Hutchinson, Norm and Warfield, Andrew. *Remus: high availability via asynchronous virtual machine replication* In Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[22] Udo Steinberg and Bernhard Kauer. *NOVA: a microhypervisor-based secure virtualization architecture.* In Proceedings of the 5th European conference on Computer systems (EuroSys '10). ACM, New York, NY, USA, 209-222.

[23] Gray, J. and Siewiorek, D.P. *High-Availability Computer Systems.* Computer , vol.24, no.9, pp.39,48, Sept. 1991.

[24] Harvard Research Group, Availability Environments (AE). http://www.hrgresearch.com/pdf/AEC%20Defintions.pdf, last consulted on 06.09.2013.

[25] Peter S. Weygant, *Clusters for High Availability: A Primer of HP Solutions, Second Edition.* 2001 Prentice Hall.

[26] Floyd Piedad, Michael Hawkins. *High Availability: Design, Techniques, and Processes.* 2001 Prentice Hall.

[27] *AMD I/O Virtualization Technology (IOMMU) Specification.* Revision: 1.26 http://support.amd.com/us/Embedded_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf Last consulted on 19.09.2013

[28] Hans-Joachim Picht. *XEN Kochbuch: Intelligente Virtualisierungslösungen mit XEN 3.* O'Reilly 2009 .

[29] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. *RT-Xen: Towards Real-time Hypervisor Scheduling in Xen* Department of Computer Science and Engineering Washington University in St. Louis

[30] Joachim Baumann, *Groovy: Grundlagen und fortgeschrittene Techniken, Kapitel 2.3* 2008, dpunkt

[31] Ralph Stever *jQuery: Das JavaScript-Framework für interaktives Design* 2011, Open Source Library

[32] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk; *CiAO: An aspect-oriented operating-system family for resource-constrained embedded system*; In Proceedings of the 2009 USENIX Annual Technical Conference, pages 215–228, Berkeley, CA, USA, June 2009. USENIX Association

[33] KICZALES, G., LAMPING, J., MENDHEKAR, A., M AEDA, C., LOPES, C., LOINGTIER, J.-M., ANDIRWIN, J. *Aspect-oriented programming.* In 11th Eur. Conf. on OOP (ECOOP '97) (June 1997), M. Aksit and S. Matsuoka, Eds., vol. 1241 of LNCS, Springer, pp. 220–242

[34] SPINCZYK, O., AND LOHMANN, D. *The design and implementation of AspectC++.* Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software 20, 7 (2007), 636–651

[35] Brian "beej Jorgensen Hall, *Beej's Guide to Network Programming Using Internet Sockets.* 03-July-2012. Version 3.0.15. `http://www.beej.us/guide/bgnet/output/print/bgnet_A4.pdf`

[36] Satya Popuri. A tour of the Mini-OS kernel. `http://www.cs.uic.edu/~spopuri/minios.html`, last consulted on 07.03.2014.

[37] lwIP - A Lightweight TCP/IP stack. `http://savannah.nongnu.org/projects/lwip/`, last consulted on 07.03.2014

[38] Giorgio C. Buttazzo. *Rate Monotonic vs. EDF: Judgment Day.* Real-

Time Syst. 29, 1 (January 2005), 5-26.

[39] RT-Xen: Real-Time Virtualization in Xen. http://blog.xen.org/index.php/2013/11/27/rt-xen-real-time-virtualization-in-xen/. Xen Project Community Blog, last consulted on 07.03.2014.

[40] Liu, C. L. and Layland, James W. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. Journal of the ACM 20(1), pp. 40-61 (1973).

[41] Neil C. Audsley. *Deadline Monotonic Scheduling*. Department of Computer Science, University of York (1990).

[42] Björn Keune. *Realisierung eines Distanzschutzes mit Methoden der Industrieautomatisierung*. Institut für Energiesysteme, Energiewirtschaft und Energieeffizienz, Technische Universität Dortmund (08/2012).

[43] Python Documentation http://www.python.org/doc/essays/comparisons.html, last consulted on 30.09.2013.