

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

Diplomarbeit Nr. 2795

# **Adaptation of the Time Dilation Factor in a Time Virtualized Emulation Environment**

Frederik Pakai

<b>Studiengang:</b>	Softwaretechnik
<b>Prüfer:</b>	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
<b>Betreuer:</b>	Dipl.-Inf. Andreas Grau
<b>begonnen am:</b>	01. Juli 2008
<b>beendet am:</b>	31. Dezember 2008
<b>CR-Klassifikation:</b>	C2.4, C4, I6.8



## Abstract

Um neu entwickelte Netzwerkprotokolle zu testen, muss man mit realistischen Szenarien arbeiten. Diese benötigen viele Knoten, auf denen das zu testende Protokoll läuft. Da diese vielen Knoten aus Kostengründen meist nicht in realer Stückzahl vorliegen können, muss simuliert oder virtualisiert werden. Bei einer massiven Virtualisierung kann der Hostknoten schnell überlastet werden. Überlast führt bei Messungen zu falschen Resultaten und macht die Messung dadurch unbrauchbar. Die Virtualisierung muss also transparent und ohne durch die Virtualisierung erzeugte Überlast geschehen. Hierfür wurden Systeme entwickelt, bei denen das komplette Experiment in einer virtuellen Zeit läuft. Durch eine Verlangsamung der virtuellen Zeit steht dadurch "mehr" Hardware zur Verfügung, da der Hostknoten mehr Zeit zum Verarbeiten der Aufgaben der virtuellen Maschinen hat. Bei bisherigen Systemen muss der sogenannte Time Dilation Factor (TDF) manuell bestimmt werden. In dieser Diplomarbeit wird eine Methode entwickelt, wie dieser Zeitverzögerungsfaktor dynamisch an die aktuell vorherrschende Last des Hostsystems angepasst werden kann. Da dies auch bei mehreren Hostsystemen möglich sein soll, wird ein verteilter TDF-Adapter entwickelt. Dieses verteilte System besteht aus Komponenten, die von den einzelnen Hardwareknoten die Last überwachen und bei Bedarf eine Lastnachricht an einen zentralen Koordinator senden. Dort gibt es eine Komponente, die aus den empfangenen Lastnachrichten einen Global-State errechnet. Aus diesem Global-State wird in der Adapter-Komponente ein geeigneter TDF berechnet. Dieser wird an alle beteiligten Knoten verteilt und synchron eingestellt. Bei der Evaluation wird gezeigt, dass durch die entwickelten Konzepte eine Überlast verhindert wird und die Messergebnisse eines Experiments korrekt sind.



# Inhaltsverzeichnis

---

<b>1</b>	<b>Einleitung</b>	<b>9</b>
1.1	Motivation . . . . .	9
1.2	Thema dieser Diplomarbeit . . . . .	10
1.3	Outline . . . . .	11
<b>2</b>	<b>Time Virtualized Emulation Environment</b>	<b>13</b>
2.1	Knotenvirtualisierung . . . . .	13
2.2	Zeitvirtualisierung . . . . .	14
2.3	Epochen-basiertes Virtuelles Zeitkonzept . . . . .	15
2.4	Time Management auf mehreren Hardwareknoten . . . . .	15
2.5	Granularität des TDF . . . . .	16
2.6	Reaktionszeit einer TDF-Anpassung . . . . .	17
2.7	Hardware . . . . .	18
2.8	Anforderungen . . . . .	19
<b>3</b>	<b>Related Work</b>	<b>21</b>
3.1	Transmission Control Protocol - TCP . . . . .	21
3.2	Regelungstechnik . . . . .	23
3.2.1	Regelkreis . . . . .	23
3.2.2	Regler . . . . .	23
3.2.2.1	Nichtstetige Regler . . . . .	24
3.2.2.2	Stetige Regler . . . . .	25
3.2.2.3	Vergleich der Eigenschaften der Reglerarten . . . . .	27
3.2.3	Sollwertfolgeregler . . . . .	27
<b>4</b>	<b>Entwurfskriterien</b>	<b>29</b>
4.1	Ermittlung von Lastveränderungen . . . . .	29
4.2	Inhalt einer Lastnachricht . . . . .	31
4.2.1	Lastcodierung mit Lastzuständen . . . . .	31
4.2.2	Lastcodierung mit realen Werten . . . . .	31
4.2.3	Hybridcodierung . . . . .	32
4.3	Regelverhalten . . . . .	32
4.3.1	Einführung eines Optimal-Last-Wertes . . . . .	32

4.3.2	Sprunggröße . . . . .	32
4.3.3	Größe der maximalen Sprunggröße . . . . .	33
4.3.4	Extremwerte des TDF . . . . .	33
4.3.5	Häufigkeit einer Anpassung . . . . .	33
4.4	Berücksichtigung des aktuellen Zustandes . . . . .	34
4.5	Berücksichtigung der Zeit . . . . .	34
4.5.1	Berücksichtigung der Vergangenheit . . . . .	34
4.5.2	Voraussagen über die Zukunft . . . . .	34
4.6	Problematik der Ausführung eines Experiments schneller als Echtzeit . . . .	35
<b>5</b>	<b>Architektur des TDF-Adapters</b>	<b>37</b>
5.1	Komponenten des TDF-Adapters . . . . .	37
5.1.1	Relevanz-Modul . . . . .	39
5.1.2	Global-State-Modul . . . . .	39
5.1.3	Adaptions-Modul . . . . .	39
5.2	Entwurf des Relevanz-Moduls . . . . .	40
5.2.1	Architektur . . . . .	40
5.2.2	Kontinuierliche Lastsignalisierung (Spezialfall) . . . . .	41
5.2.3	Trigger . . . . .	41
5.2.3.1	Timergesteuerter Trigger . . . . .	41
5.2.3.2	Eventbasierte Trigger . . . . .	42
5.2.4	Filter . . . . .	42
5.2.4.1	Ohne Filter . . . . .	43
5.2.4.2	Differentieller Filter . . . . .	43
5.2.4.3	Schwellenwert-basierter Filter . . . . .	43
5.3	Entwurf des Global-State-Moduls . . . . .	43
5.4	Entwurf des Adaptions-Moduls . . . . .	44
5.4.1	TCP-basierter Ansatz . . . . .	44
5.4.1.1	Signalisierung der Last . . . . .	44
5.4.1.2	Regelalgorithmus . . . . .	45
5.4.1.3	Timer . . . . .	45
5.4.2	Regler-basierter Ansatz . . . . .	46
5.4.2.1	Signalisierung der Last . . . . .	46
5.4.2.2	Reglerbereich . . . . .	46
5.4.2.3	Regelalgorithmus . . . . .	47
5.4.2.4	Timer . . . . .	48
<b>6</b>	<b>Implementierung</b>	<b>49</b>
6.1	Allgemeines . . . . .	49
6.2	Implementierung des Relevanz-Moduls . . . . .	50
6.2.1	Trigger . . . . .	50
6.2.1.1	Timergesteuerter Trigger . . . . .	50
6.2.1.2	Eventgesteuerter Trigger . . . . .	51
6.2.2	Filter . . . . .	54
6.2.2.1	Differentieller Filter . . . . .	54

6.2.2.2	Schwellenwert-basierter Filter . . . . .	54
6.3	Implementierung des Global-State-Moduls . . . . .	55
6.3.1	Aktualisierung der Lastwerte . . . . .	55
6.3.2	Berechnung des Global-State . . . . .	56
6.4	Implementierung des Adaptions-Moduls . . . . .	57
6.4.1	TCP-basierter Ansatz . . . . .	57
6.4.1.1	Timer . . . . .	57
6.4.1.2	TDF Grenzwerte . . . . .	57
6.4.1.3	Anpassungsschritte . . . . .	57
6.4.1.4	Startsequenz . . . . .	58
6.4.1.5	Reaktion bei Unterlast . . . . .	58
6.4.1.6	Feintuning bei Optimallast . . . . .	59
6.4.1.7	Reaktion bei potentieller Überlast . . . . .	60
6.4.1.8	Reaktion bei Überlast . . . . .	61
6.4.1.9	Basisalgorithmus . . . . .	62
6.4.1.10	Diskussion . . . . .	62
6.4.2	Regler-basierter Ansatz . . . . .	63
6.4.2.1	Timer . . . . .	63
6.4.2.2	Optimallast . . . . .	63
6.4.2.3	TDF Grenzwerte . . . . .	63
6.4.2.4	Anpassungsschritte . . . . .	64
6.4.2.5	Startsequenz . . . . .	64
6.4.2.6	Reaktion bei Unterlast . . . . .	65
6.4.2.7	Feintuning bei Optimallast . . . . .	66
6.4.2.8	Reaktion bei Überlast . . . . .	67
6.4.2.9	Basisalgorithmus . . . . .	68
6.4.2.10	Diskussion . . . . .	69
<b>7</b>	<b>Evaluation</b>	<b>71</b>
7.1	Evaluierungsszenarien . . . . .	71
7.1.1	Optimized-Link-State-Routing . . . . .	71
7.1.2	Routerkette . . . . .	73
7.1.3	OLSR mit TCP-Verbindung . . . . .	75
7.2	Kriterien . . . . .	77
7.2.1	Korrektheit . . . . .	77
7.2.2	Overhead . . . . .	77
7.2.3	Experimentlaufzeit . . . . .	77
7.3	Ergebnisse . . . . .	78
7.3.1	Festlegung der Extremwerte des TDF . . . . .	78
7.3.2	Initiale Adaption / Reglerbereich . . . . .	79
7.3.3	Trigger . . . . .	79
7.3.3.1	Eventgesteuerter Trigger . . . . .	79
7.3.3.2	Timergesteuerter Trigger . . . . .	79
7.3.3.3	Festlegung der Timerintervalle beim timergesteuerten Trigger und Berücksichtigung des aktuellen Zustandes . . . . .	80

7.3.4	Festlegung der Intervalle des Adaption-Timers . . . . .	83
7.3.5	Festlegung des Optimalwertes beim Regler-basierten Ansatz . . . . .	84
7.3.6	Differentielle TDF-Sprünge bei Unterlast und Überlast . . . . .	85
7.3.7	Festlegung der TDF-Sprünge . . . . .	86
7.3.8	Vergleich der Filter . . . . .	86
7.3.8.1	Ohne Filter . . . . .	86
7.3.8.2	Differentieller Filter . . . . .	87
7.3.8.3	Schwellenwert-basierter Filter . . . . .	88
7.3.8.4	Fazit . . . . .	90
7.3.9	Festlegung des Überlastschwellewert beim TCP-basierten Ansatz . .	91
7.3.10	Vergleich der Adaptionkonzepte . . . . .	92
7.4	Zusammenfassung . . . . .	94
<b>8</b>	<b>Zusammenfassung und Ausblick</b>	<b>97</b>
8.1	Zusammenfassung . . . . .	97
8.2	Ausblick . . . . .	98
	<b>Abbildungsverzeichnis</b>	<b>99</b>
	<b>Tabellenverzeichnis</b>	<b>101</b>
	<b>Verzeichnis der Listings</b>	<b>103</b>
	<b>Literaturverzeichnis</b>	<b>105</b>

# Einleitung

---

## 1.1 Motivation

Eine wichtige Aufgabe bei der Entwicklung Verteilter Systeme oder Netzwerkprotokolle ist es, die entwickelten Systeme zu testen und zu evaluieren. Dabei benötigt man realistische Bedingungen. Dies bedeutet beispielsweise, dass man Szenarien erstellen muss, in denen teilweise tausende von Rechnern über das neu entwickelte Netzwerkprotokoll kommunizieren müssen [24]. Um ein solches Szenario zu realisieren ist es in der Praxis aus Kostengründen meist nicht möglich mit realer Hardware in angemessener Stückzahl zu agieren. Als Alternative gibt es hierbei die Möglichkeiten der Simulation [21] [14] oder der Emulation/Virtualisierung [23].

Der Nachteil einer Simulation liegt darin, dass man nicht mit dem realen zu evaluierenden Protokoll arbeiten kann, sondern für die Simulation entsprechende Anpassungen am Prototyp vornehmen muss. Bei der Virtualisierung hingegen kann mit dem unveränderten Prototyp gearbeitet werden.

Wenn man nun beispielsweise ein Szenario mit tausend Rechnern aufsetzen will benötigt man entsprechend viele Hardwareknoten oder man verfügt über entsprechend hochperformante Hardwareknoten, auf denen die entsprechende Anzahl an Virtuellen Maschinen läuft. Dies ist allerdings meist sehr teuer.

Wenn auf einem Hardwareknoten hundert Virtuelle Knoten laufen sollen die an der Evaluierung beteiligt sind hat man das Problem, dass die Performance des Hardwareknotens nicht ausreicht und somit der Knoten überlastet ist. Wenn der Hardwareknoten mit Gigabit an das Netzwerk angeschlossen ist und jeder dieser hundert Virtuellen Knoten mit 100 Megabit Daten versenden will ist der Hardwareknoten voll ausgelastet und kann die anfallende Datenflut nicht ohne Fehler bewältigen. Selbiges kann man mit der CPU beobachten. Das Problem bei Überlast in einer Evaluation besteht darin, dass die Messergebnisse verfälscht werden, weil es beispielsweise zu Timeouts kommen kann, die eigentlich in der Realität nicht aufgetreten wären, sondern nur durch die massive Virtualisierung entstanden sind. Die Lösung des Problems besteht darin, bessere Hardware zur Verfügung zu stellen, mehr Hardware zu besorgen oder, der Ansatz der in dem Projekt dieser Diplomarbeit (Time Virtualized Emulation Environment - TVEE [15]) angewandt

wird, nämlich die Zeit für das Experiment zu verlangsamen. Wenn auf allen beteiligten Knoten die Zeit gedehnt wird steht quasi mehr Hardware zur Verfügung. Konkret, wenn man die Zeit um den Faktor 10 verlangsamt hat man entsprechend die 10fache Kapazität, da beispielsweise alle Sendewünsche für Daten über das Netzwerk 10 mal so viel Zeit haben gesendet zu werden. Also kommt der reale Hardwareknoten mit der Last klar ohne dass es zu Verzögerungen aufgrund von Überlast kommt. Somit sind die Messergebnisse korrekt und nicht wie bei Überlast verfälscht.

Man erreicht also durch die Verlangsamung der Zeit einen Hardware-/Performancegewinn. Dies ermöglicht es, auch mit älterer Hardware Experimente durchzuführen, die auf Supercomputern durchgeführt werden müssten, um in Echtzeit ablaufen zu können. Zudem gibt es Szenarien in denen man Protokolle auf ihre Zuverlässigkeit, mit beispielsweise Netzwerkbandbreiten die aktuell noch gar nicht verfügbar sind, testen will. Auch dies ist mit einer Verlangsamung der Zeit realisierbar.

Ein anderer Vorteil der Anpassung der Zeit besteht darin, Experimente schneller als Echtzeit ablaufen zu lassen und damit Zeit zu gewinnen und evtl. auch Kosten für Hochleistungsrechner zu sparen. Wenn ein Experiment nur minimale Anforderungen an die Hardware stellt kann die Virtuelle Zeit erhöht werden. Somit erreicht man einen Zeitgewinn durch die Beschleunigung der Experimentlaufzeit.

Die Time Virtualized Emulation Environment (TVEE) realisiert genau diesen Ansatz. Auf einem Hardwareknoten (pNode) können beliebig viele Virtuelle Knoten laufen. Durch die Anpassung des Time Dilation Factors (TDF) wird die Zeit in den Virtuellen Maschinen an die Bedürfnisse angepasst, so dass es nie zu andauernder Überlast bzw. Unterlast kommen kann. Ein hoher TDF bedeutet dabei, dass die Zeit langsamer, ein niedriger bzw. negativer TDF bedeutet, dass die Zeit schneller als Echtzeit läuft. Mit der entsprechenden Anzahl an Hardwareknoten können somit fast beliebige Szenarien mit realistischen Messergebnissen ohne Anpassung des Probanden evaluiert werden.

## 1.2 Thema dieser Diplomarbeit

Problematisch ist nun die Auswahl des richtigen TDF für ein Experiment. Bisher kann man den TDF vor dem Start des Experiments einstellen. Während des Experiments bleibt diese Einstellung bestehen und der TDF kann nicht geändert werden. Wählt man den Faktor zu Beginn des Experiments zu hoch, also verlangsamt man das Experiment zu sehr, dauert es länger und man vergeudet Zeit und Ressourcen. Wählt man den TDF zu niedrig kann es sein, dass es zu einer länger andauernden Überlast kommt und die gesammelten Messergebnisse dadurch unbrauchbar werden. Das Experiment müsste mit höherem TDF, also langsamerer Virtueller Zeit, von Neuem beginnen.

Wünschenswert ist also eine dynamische Anpassung des TDF an die aktuelle Last. Genau mit diesem Thema beschäftigt sich diese Diplomarbeit. Es sollen Verfahren und Strategien entwickelt werden, um den TDF abhängig von der aktuell anliegenden Last möglichst optimal zu bestimmen. Optimal bedeutet in diesem Fall, dass sich die Last

eines Hardwareknotens nie dauerhaft in Überlast befindet, aber die Last immer relativ hoch ist, so dass die gesamte Experimentlaufzeit minimiert wird. Das Hauptaugenmerk liegt allerdings auf der Korrektheit der gesammelten Messergebnisse. Hierbei gibt es diverse Faktoren und Anforderungen die beachtet werden müssen. Diese werden in einem separaten Kapitel erläutert.

## 1.3 Outline

In den folgenden Kapiteln dieser Diplomarbeit werden zunächst die Konzepte und die Idee des dieser Diplomarbeit zugrunde liegenden Projekts „Time Virtualized Emulation Environment“ vorgestellt. Dabei werden auch die Anforderungen an den in dieser Diplomarbeit zu entwickelnden TDF-Adapter erläutert und anschließend einige Faktoren genannt, die in diesem Zusammenhang beachtet und untersucht werden müssen.

In Kapitel 3 (Related Work) wird auf bisherige Forschungsprojekte Bezug genommen, aus denen Ideen und Konzepte übernommen werden können.

Kapitel 4 (Entwurfskriterien) beschäftigt sich mit der Entwicklung geeigneter Strategien zur Anpassung des TDF, die dann in den folgenden Kapiteln umgesetzt und evaluiert werden.

Kapitel 5 (Architektur des TDF-Adapters) gibt einen Einblick in die Architektur des verteilten TDF-Adapters. Die Konzepte und Ansätze, die die Aufgabe der Adaption umsetzen, werden hier entwickelt.

In Kapitel 6 (Implementierung) werden die entwickelten Konzepte implementiert und detailliert vorgestellt.

In Kapitel 7 (Evaluation) werden letztlich die entwickelten Konzepte evaluiert und es werden sinnvolle Werte für die Parameter der Konzepte gefunden.

Im letzten Kapitel dieser Diplomarbeit wird dann eine kurze Zusammenfassung über die komplette Diplomarbeit und ein Ausblick über die zukünftige Arbeit in diesem Bereich gegeben.



# Time Virtualized Emulation Environment

---

Das dieser Diplomarbeit zu Grunde liegende Projekt, die „Time Virtualized Emulation Environment“ (TVEE) verfolgt einen Hybrid-Virtualisierungs-Ansatz, der sowohl Knotenvirtualisierung als auch Zeitvirtualisierung realisiert. Hierbei wird die Knotenvirtualisierung mit sehr wenig Overhead erreicht. Die Zeitvirtualisierung ist dabei so umgesetzt, dass sie komplett transparent für die eigentliche Testumgebung ist. Es sind also keinerlei Modifikationen an der zu evaluierenden Software vorzunehmen.

## 2.1 Knotenvirtualisierung

Die Knotenvirtualisierung in TVEE ist in zwei Ebenen aufgebaut. In der ersten Ebene befindet sich auf jedem Hardwareknoten (pNode) genau eine Virtuelle Maschine mit XEN [7]. In der zweiten Ebene laufen in dieser Virtuellen Maschine mittels Virtuellem Routing, das mit OpenVZ [1] realisiert ist, beliebig viele Virtuelle Maschinen (vNodes). Auf jedem Hardwareknoten läuft nur genau eine „echte“ Virtuelle Maschine, da „echte“ Virtuelle Maschinen jeweils ein eigenes OS und wesentlich mehr Performance benötigen als bei Virtuellem Routing. Bei einer Netzwerkkommunikation zwischen zwei „echten“ Virtuellen Maschinen sind zum Beispiel jedes Mal Kontextwechsel zwischen den Maschinen notwendig. Dies ist sehr ineffizient. Beim Virtuellen Routing hingegen gibt es nur ein Betriebssystem. Die Prozesse und Netzwerkstacks der vNodes sind aber voneinander isoliert, so dass es keine gegenseitige Beeinflussung gibt. Prinzipiell verhält sich jeder Knoten, der mit Virtual Routing erzeugt wurde (vNode), wie eine „echte“ Virtuelle Maschine, jedoch ist dieses Konzept mit Virtual Routing wesentlich effizienter. Bei einem Vergleich zwischen Virtuellen Maschinen und Virtual Routing [18] wurde gezeigt, dass auf einem Hardwareknoten sechs Virtuelle Maschinen parallel laufen können. Mit Virtual Routing konnten auf demselben Hardwareknoten hingegen über 30 Maschinen laufen. Somit hat man also bei Virtual Routing ca. die 5fache Kapazität an Maschinen als bei „echten“ Virtuellen Maschinen. Abbildung 2.1 zeigt den Aufbau dieser Zwei-Ebenen-Virtualisierung.

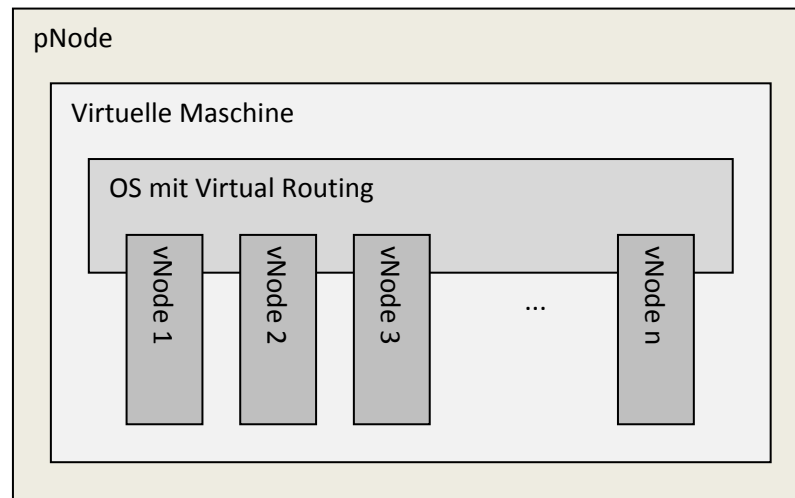


Abbildung 2.1: Architektur des TVEE - vNodes mittels Virtual Routing innerhalb einer Virtuellen Maschine

## 2.2 Zeitvirtualisierung

Prinzipiell gibt es zwei Arten mit denen man Zeitvirtualisierung realisieren kann. Die erste Möglichkeit besteht darin das komplette Betriebssystem zu modifizieren und somit die Unterstützung für die Zeitvirtualisierung in den Kernel einzubauen. Dies bedeutet aber extremen Programmieraufwand. Die andere Möglichkeit, die auch in TVEE verwendet wird, behilft sich mit einer Virtuellen Maschine. Gupta et al. [16] führte eine Zeitvirtualisierung mit einer Virtuellen Maschine ein. Hierbei sind nur sehr geringe Änderungen an der Virtuellen Maschine notwendig. Mit dem hierbei eingeführten Time Dilation Faktor (TDF) kann der Zeitfaktor bestimmt werden um den die Zeit in der Virtuellen Maschine in Bezug auf die Realzeit verändert wird. Da die komplette Virtuelle Maschine in dieser Virtuellen Zeit läuft, ist die Zeitvirtualisierung transparent für alle Inhalte der Virtuellen Maschine. Es sind also keinerlei Anpassungen der Software innerhalb der Virtuellen Maschine notwendig. Da man nur eine Virtuelle Maschine auf jedem pNode besitzt muss nur diese angepasst werden. Für alle vNodes, die mit dem Virtual Routing in dieser Virtuellen Maschine laufen, ist die Zeit ohne sonstige Aktivitäten sofort angepasst, da diese sich an der Zeit der Virtuellen Maschine orientieren.

Jetzt ist auch klar warum in TVEE das Virtual Routing nicht direkt auf dem pNode läuft, sondern der Zwischenschritt über die Virtuelle Maschine mit XEN vorgenommen wird. Indem man die Zeit in der Virtuellen Maschine anpasst hat man bereits für alle vNodes innerhalb dieser Virtuellen Maschine die Zeit entsprechend angepasst.

## 2.3 Epochen-basiertes Virtuelles Zeitkonzept

Bisherige zeitvirtualisierende Systeme basieren auf einem Virtuellen Zeitkonzept, bei dem der TDF zu Beginn des Experiments eingestellt wird und zur Laufzeit des Experiments nicht verändert wird. Um nun allerdings dynamisch auf Lastschwankungen reagieren zu können und den TDF entsprechend anpassen zu können, ist es erst notwendig sogenannte Epochen einzuführen. Epochen sind Zeiträume in denen der TDF und somit die Virtuelle Zeit nicht verändert werden. TVEE basiert auf solchen Epochen. Zu Beginn einer Epoche wird der optimale TDF berechnet und eingestellt. Besser gesagt: wenn eine dynamische Anpassung stattfindet beginnt eine neue Epoche. Während dieser Epoche wird keine Zeitverschiebung vorgenommen. Alle vNodes laufen innerhalb einer Epoche mit dem selben TDF.

## 2.4 Time Management auf mehreren Hardwareknoten

Bisher betrachteten wir den Fall, dass es nur einen Hardwareknoten gibt. Da man aber die Möglichkeit haben will, dass die vNodes für ein Experiment auf diversen pNodes laufen können, benötigt man einen Mechanismus, den TDF nicht nur für einen pNode sondern für alle am Experiment beteiligten pNodes synchron einzustellen. Diese Aufgabe verteilt sich im Gesamten auf drei Teilbereiche.

- Die erste Aufgabe besteht darin, die Last im gesamten System zu überwachen und einen kompletten Überblick über das gesamte Verteilte System zu erstellen.
- Die zweite Aufgabe besteht nun darin, einen optimalen TDF aus den gesammelten Daten zu errechnen.
- Die dritte und letzte Aufgabe besteht darin, den errechneten TDF synchron auf alle pNodes zu verteilen.

In TVEE werden diese Aufgaben von drei separaten Komponenten realisiert (Abbildung 2.2).

Auf jedem Knoten läuft ein LoadMonitor, der die lokale Last des Hardwareknotens und damit die gesamte Last aller vNodes des Knotens überwacht. Die Last kann hierbei von verschiedenen Komponenten abhängen. Betrachtet werden können beispielsweise CPU-Auslastung, Netzwerkbandbreitenkapazität, Festplattenlast, Speicherausnutzung... Der LoadMonitor errechnet mit Hilfe einer Metrik einen Lastwert, der charakteristisch für die Auslastung des Knotens ist. Diese Lastinformation wird an einen zentralen Koordinatorknoten gesendet. Auf diesem wird der in dieser Diplomarbeit zu entwickelnde TDF-Adapter laufen, der anhand der gesammelten Lastinformationen den neuen, optimalen TDF errechnet. Ebenfalls auf dem zentralen Koordinator läuft der EpochSwitcher. Dieser bekommt vom TDF-Adapter den errechneten TDF und schickt ihn mittels Multicast an alle am Experiment beteiligten pNodes, bei denen der TDF dann synchron eingestellt wird. Dieser geschlossene Regelkreis (Abbildung 2.3) wird über die gesamte Dauer des Experiments zigfach durchlaufen.

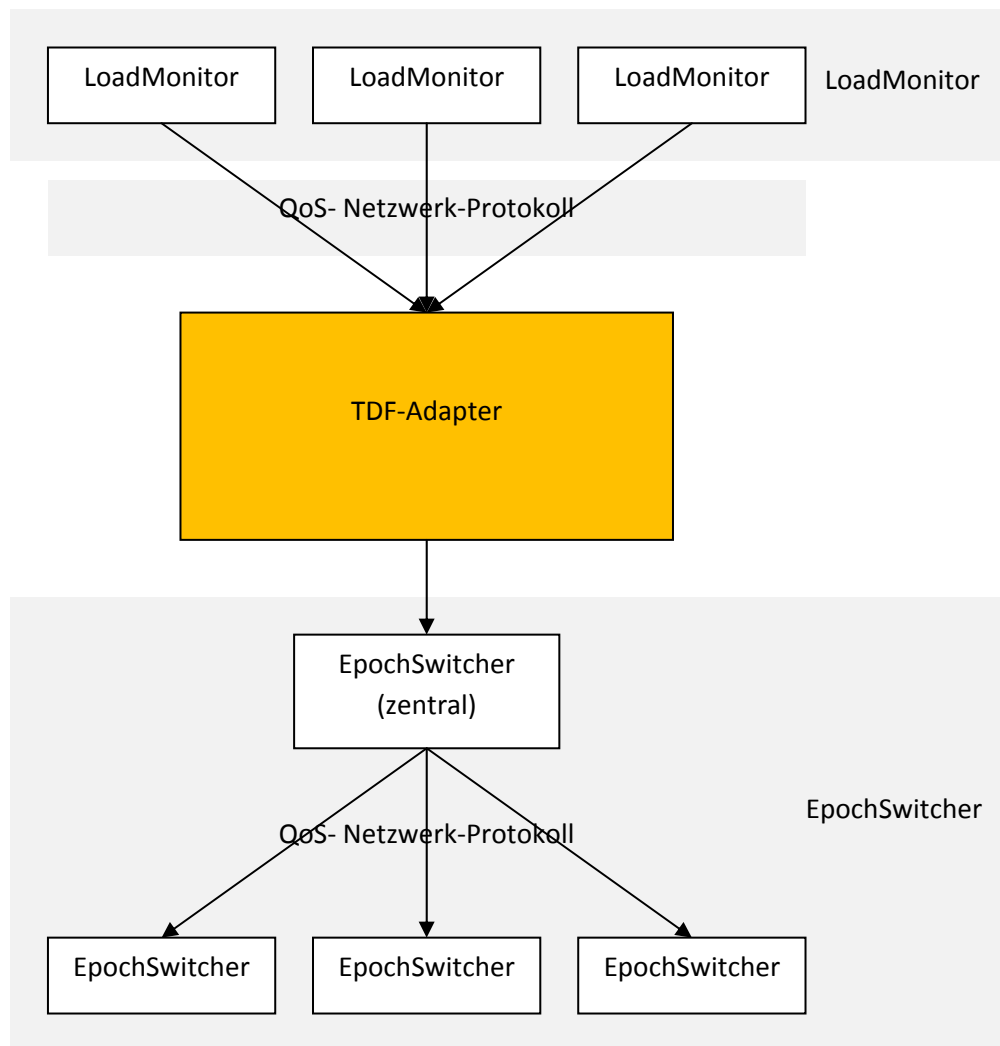


Abbildung 2.2: Zusammenspiel der Komponenten der TVEE

Sowohl der LoadMonitor als auch der EpochSwitcher sind in separaten Diplomarbeiten erarbeitet worden, bzw. werden noch erarbeitet. Deshalb wird hier nur auf die entsprechenden Diplomarbeiten verwiesen, die bei Interesse hinzugezogen werden können.

### 2.5 Granularität des TDF

Um die Virtuelle Zeit sehr feingranular bestimmen zu können wird der TDF in TVEE nicht 1:1 in Virtuelle Zeit umgerechnet sondern 100:1. Dies bedeutet, dass eine Erhöhung des TDF um 100 eine Halbierung der Virtuellen Zeit bedeutet. Entsprechend bewirkt eine

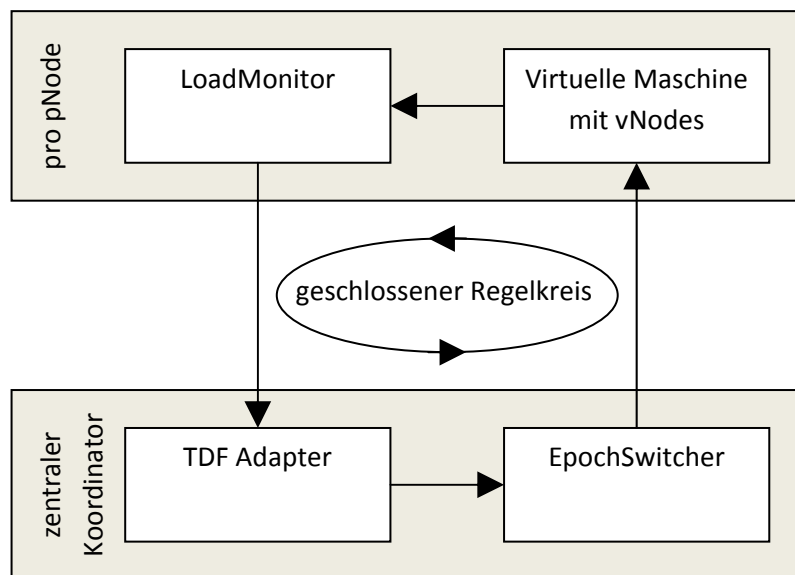


Abbildung 2.3: Regelkreis des TVEE - Regelkreis, der während der gesamten Experimentdauer immer wieder durchlaufen wird

Verringerung des TDF um 100 eine Verdopplung der Virtuellen Zeit. Im konkreten Fall lautet die Formel für die Berechnung der Virtuellen Zeit aus dem TDF:

$$T'_{virtuell} = T'_{real} \times 2^{(TDF/100)}$$

## 2.6 Reaktionszeit einer TDF-Anpassung

Die Zeit, die benötigt wird, um auf einen Lastwert zu reagieren ist nicht Null. Sie addiert sich aus diversen Operationen und Aktionen zusammen. Als erstes muss der aktuelle Lastwert eines Knotens ermittelt werden. Da dieser Vorgang automatisch jede Millisekunde durchgeführt wird, dauert dies im schlechtesten Fall 1 ms. Anschliessend wird der aktuelle Lastwert mittels des Netzwerkes zu einem Zentralen Koordinator verschickt. Diese Kommunikation dauert nicht lange, aber dennoch benötigt es Zeit. Nachdem die Information am Zentralen Koordinator angekommen ist, wird dort der Global-State gebildet und der TDF entsprechend adaptiert. Anschliessend wird der neu berechnete TDF wieder per Netzwerk an die Knoten verteilt und dort eingestellt.

Somit haben wir eine Reaktionszeit der TDF-Anpassung von ca. 2 ms. Dies bedeutet im Einzelnen, dass wir den TDF immer zeitverzögert einstellen, also beispielsweise auf eine Überlast reagieren, obwohl diese bereits nicht mehr vorhanden ist. Dies muss bei der Adaption entsprechend beachtet werden.

### 2.7 Hardware

An der Universität Stuttgart gibt es bereits einen Prototypen dieser TVEE. Die dafür im Einsatz befindliche Hardware besteht aus einem Cluster mit insgesamt 64 Knoten, die in verschiedene Experimentgruppen eingeteilt werden können. Somit ist es möglich mehrere Experimente mit verschiedener Anzahl an Knoten parallel zu betreiben. Das gesamte Cluster ist an einen extrem performanten Switch angeschlossen mit dem es möglich ist die einzelnen Experimentgruppen in verschiedenen VLANs zu kapseln. Somit behindern sich die Experimente nicht gegenseitig.

Die einzelnen Knoten sind mit einem Pentium 4 Prozessor mit 2,4 GHz ausgestattet. Jeder Knoten besitzt 512 MB Hauptspeicher und zwei Netzwerkkarten. Dabei ist ein Netzwerkanschluss mit der Geschwindigkeit 100 MBit versehen und nur für administrative Zwecke und die Realisierung der Adaption vorgesehen. Der andere Netzwerkanschluss mit der Geschwindigkeit 1 GBit ist für die eigentliche Nutzlast der Experimente vorgesehen. Somit behindern sich auf Netzwerkebene die Nutzdaten und die Kontrolldaten des Clusters nicht. Auf den Knoten läuft ein Linux-Betriebssystem [8] [12] in der Version 2.6.18 sowie XEN in der Version 3.02 [6] [9].



Abbildung 2.4: 64 Knoten Cluster der Univerität Stuttgart auf dem der Prototyp der TVEE läuft (<http://net.informatik.uni-stuttgart.de/>)

## 2.8 Anforderungen

Bei der Berechnung des TDF gelten einige Anforderungen, die nun im Folgenden erläutert werden:

- Der TDF muss so schnell und optimal berechnet und eingestellt werden, dass die Messergebnisse der Evaluation eines Probanden korrekt sind und nicht durch Virtualisierungseffekte verfälscht werden.
- Der TDF sollte immer so berechnet werden, dass die Knoten immer bei einer hohen Last betrieben werden. Ein Knoten darf sich aber nur für sehr kurze Zeit in Überlast befinden und muss dann sofort die Überlast überwunden haben.
- Die Berechnung des TDF sollte so schnell gehen, dass ca. 1.000 TDF-Anpassungen pro Sekunde möglich sein können.
- Der TDF darf niemals über einem bestimmten Wert sein, so dass die Virtuelle Zeit nicht zu langsam wird. Bei zu langsamer Virtueller Zeit ist es nicht mehr sinnvoll möglich Überwachungsaufgaben oder Modifikationen mittels einer Remoteverbindung auf einem Knoten, der mit dieser langsamen Zeit läuft, durchzuführen.
- Der TDF darf niemals unterhalb eines bestimmten Wertes liegen. Wenn die Virtuelle Zeit zu schnell wird, kann man sich beispielsweise nicht mehr per SSH auf einem Knoten einloggen, da man bereits bei der Anmeldung einen Timeout bekommt.
- Die Berechnung des TDF sollte nicht allzu oft globale Lastinformationen verlangen, da sonst zu viele Netzwerknachrichten verschickt werden müssen. Dies bedeutet unnötigen Overhead, der auf Kosten der Experimentzeit gehen kann.
- Der TDF darf keine allzugroßen Sprünge machen, da sonst die Maschinen auseinanderlaufen können. Da der TDF mittels des Netzwerkes verteilt wird kann es sein, dass die Pakete mit leichten Unterschieden bei den Zielknoten ankommen. Da zudem bei den Knoten unterschiedliche Last anliegen kann, kann es auch hierbei zu Unterschieden beim Setzen des TDF geben. Wenn man von einem sehr niedrigen TDF auf einen sehr hohen TDF schaltet, kann es dadurch vorkommen, dass die Maschinen die länger mit niedrigem TDF (also schneller Virtueller Zeit) laufen einen Timeout bei einer Netzwerkkommunikation bekommen, da der Kommunikationspartner bereits mit hohem TDF (langsame Virtuelle Zeit) läuft und nicht in der erwarteten schnellen Zeit reagiert hat.
- Der TDF muss so berechnet werden, dass kein Hardwareknoten in Überlast gerät. Es muss also eine globale Lastsicht als Grundlage der Berechnung herangezogen und nicht nur die Auslastung eines Knotens beachtet werden.
- Die Reaktion auf eine drohende Überlast muss umgehend erfolgen, so dass es niemals zu einer lang anhaltenden Überlast kommt.
- Der TDF und dessen Adaption darf kein Aufschaukeln oder Oszillieren der Last verursachen.



# Related Work

---

Zum Zeitpunkt der Erstellung dieser Diplomarbeit gibt es noch keine Arbeit darüber, wie man eine TDF Anpassung mit den zuvor beschriebenen Anforderungen und Eigenschaften für diese Aufgabe realisiert. Jedoch gibt es einige Beispiele, die ähnliche Aufgaben und Probleme zu bewerkstelligen hatten, aus denen Ideen und Vorschläge übernommen werden können. Diese werden hier teilweise vorgestellt und erörtert inwieweit Teile und Ideen für diese Diplomarbeit relevant sein könnten.

### 3.1 Transmission Control Protocol - TCP

Das Transmission Control Protokoll TCP [25] [19], das heutzutage weltweit eingesetzt wird, muss auch auf Überlast im Netzwerk reagieren [5]. Hierfür wurden einige Ideen und Mechanismen entwickelt, die heute in verschiedenen TCP Versionen umgesetzt sind. Überlast bei TCP bedeutet, dass zwischen dem Sender und dem Empfänger der Pakete ein Stau im Netzwerk vorhanden ist und somit die gesendeten Pakete nicht ankommen. Dies ist nicht sehr kritisch, da nach einem Timeout beim Sender die nicht vom Empfänger bestätigten Pakete erneut gesendet werden. Um allerdings in möglichst kurzer Zeit möglichst viele Pakete versenden zu können, ist es notwendig, eine Flusskontrolle zu haben. Da man zu Beginn einer TCP-Verbindung nicht weiß, wie die Auslastung im Netz aktuell aussieht und sich die Last auch dauernd ändern kann, kann man schwer eine konstante Flussmenge versenden.

Aus diesem Grund wird bei TCP zu Beginn einer Verbindung mit einer niedrigen Rate gesendet. Nach jedem Paket, das beim Empfänger ankommt, wird die Rate erhöht. Dies geschieht bis zu einem bestimmten Schwellenwert exponential und danach linear. Die Senderate wird so lange erhöht bis es schließlich zu einer Überlast im Netzwerk kommt. Hierbei ist für den Sender nur entscheidend, was mit seinen gesendeten Paketen geschieht. Er benötigt hierfür kein globales Wissen wie wir bei TVEE. Jeder Sender für sich versucht seine Senderate so hoch wie möglich zu treiben. TCP wird auch als *greedy* bezeichnet. Wenn nun eine Überlast vorliegt, muss entsprechend darauf reagiert werden. In verschiedenen Versionen des TCP-Protokolls wird hierauf unterschiedlich reagiert. Allen Versionen ist

gemein, dass sie natürlich die Senderate drosseln. Nachdem die Senderate beispielsweise um die Hälfte reduziert wurde, wird nun mit dem linearen Ansteigen der Senderate bis zur nächsten Überlast fortgesetzt. Auf diese Weise versucht jeder TCP-Sender für sich die optimale Senderate zu erringen.

Abbildung 3.1 zeigt ein Beispiel für eine Flusskontrolle bei TCP.

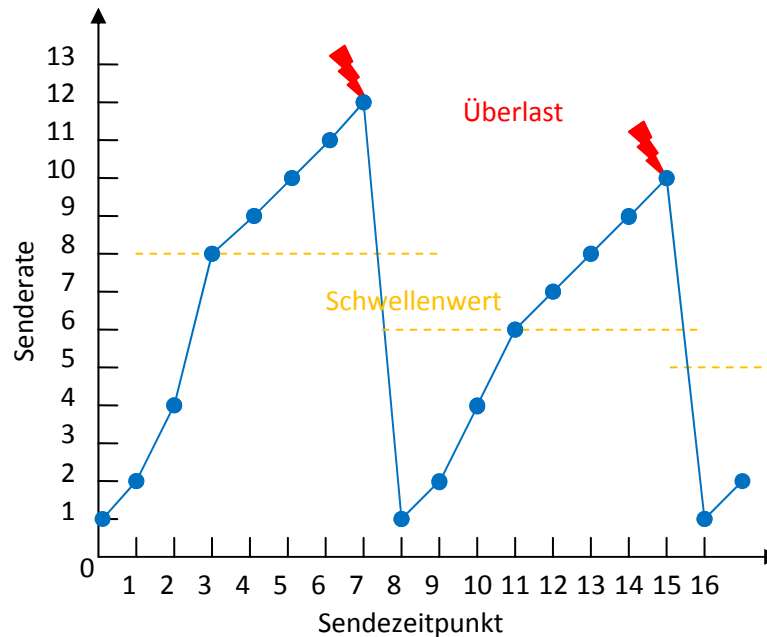


Abbildung 3.1: TCP - Flusskontrolle

Für unsere Aufgabe ist dieser Mechanismus allerdings nicht 100%ig geeignet. Bei TVEE sollte es niemals zu einer Überlast kommen. Somit muss bereits bei einer drohenden Überlast reagiert werden und der TDF erhöht werden.

Der größte Unterschied liegt wohl in der Reaktionszeit. Bei TCP sind Verzögerungen von mehreren Sekunden durchaus kein Problem. Der Timeout bei TCP kann auch bei mehreren Minuten liegen. Bei TVEE muss bereits innerhalb eines Bruchteils einer Sekunde reagiert werden. Somit kann man hier nicht mit irgendwelchen Timeouts oder anderen Mechanismen als Überlastindiz arbeiten.

Ein weiterer Grund, dass man TCP nur als Ideengeber verwenden kann liegt darin, dass bei TVEE ein globaler Überblick über alle am Experiment beteiligten Hardwareknoten vorhanden sein muss. Sobald ein Hardwareknoten eine drohende Überlast hat, muss der TDF global erhöht werden. TVEE sollte *greedy* sein, aber nicht an einem einzelnen pNode orientiert, sondern am gesamten System gesehen.

Als durchaus interessant könnte die grundlegende Idee der TCP-Überlastkontrolle sein. Man startet bei einem niedrigen Wert (Zeit) und erhöht ihn solange bis man eine potentielle Überlast bekommt. Danach drosselt man den Wert um einen Betrag  $X$  und tastet sich wieder an das Maximum heran.

## 3.2 Regelungstechnik

Da in dieser Diplomarbeit ein Regelvorgang erarbeitet werden soll, wird hier auf das Gebiet der Regelungstechnik [10] [13], und dabei speziell auf den Regelkreis und die darin enthaltenen Regler, eingegangen.

### 3.2.1 Regelkreis

Ein Regelkreis in der Regelungstechnik ist ein System, das, bestehend aus einem Regler, einer Regelkette und einer Rückführung, bestimmte Regelaufgaben umsetzt. Hierbei kann das Ziel beispielsweise eine Stabilisierung der Regelkette oder die Einhaltung einer Sollwertfolge, also eine asymptotische Annäherung des Istwertes an einen bestimmten Sollwert sein. Die Logik, die zum Regeln benötigt wird, ist im Regler verankert. Charakteristisch bei einem Regelkreis ist, dass die vorgenommenen Änderungen an der Regelstrecke und die daraus resultierende Änderung des Messwertes zurückgeführt werden. Es entsteht also ein geschlossener Regelkreis der auf Fremdeinflüsse und eigene Manipulationen der Regelkette reagiert und so die für diesen Regelkreis definierte Anwendung umsetzt. Abbildung 3.2 zeigt einen solchen Regelkreis.

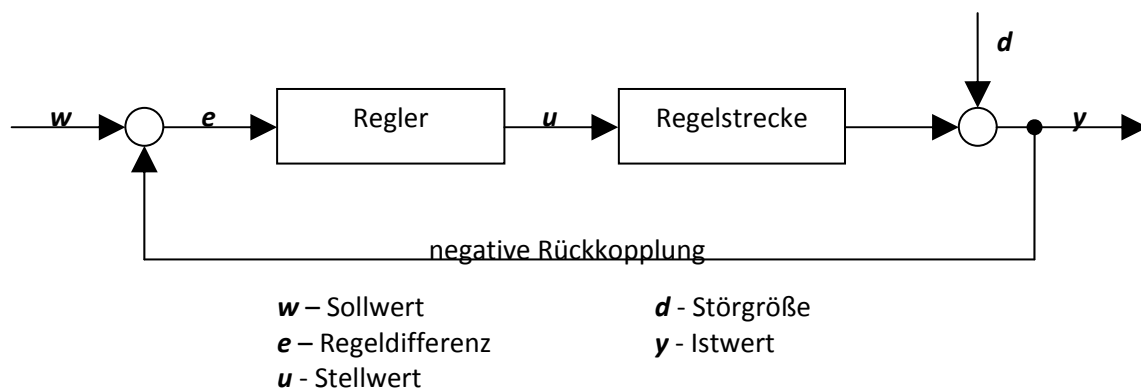


Abbildung 3.2: Blockschaltbild eines Standardregelkreises

### 3.2.2 Regler

Die Funktionsweise eines Reglers ist in dieser Diplomarbeit von besonderem Interesse. Es gibt unzählige Arten von Reglern, die aber alle in bestimmte Klassen eingeteilt werden können. Prinzipiell gibt es zwei Oberklassen von Reglern - analoge Regler und digitale Regler. Allerdings sind die zugrunde liegenden Ideen und Konzepte dieselben. Bei den analogen Reglern wird die Logik mit Hilfe von elektrischen Bauteilen realisiert, bei einem digitalen Regler kommt ein Microprozessor für diese Aufgabe zum Einsatz. Hier wird deshalb nur die Grundidee der einzelnen Reglerklassen erläutert (Abbildung 3.3).

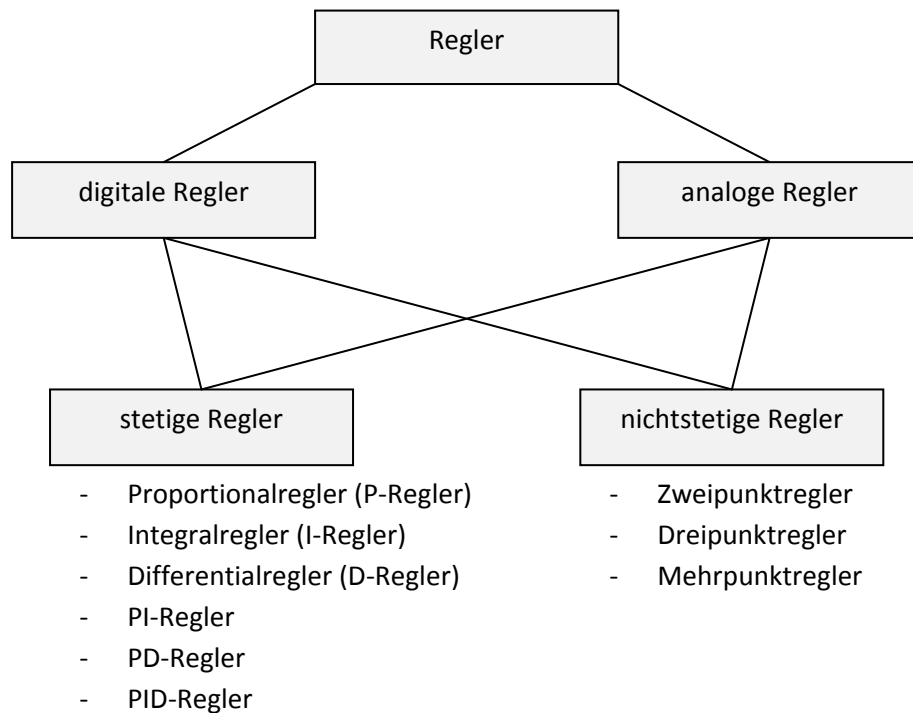


Abbildung 3.3: Klassifizierung der verschiedenen Reglerarten

### 3.2.2.1 Nichtstetige Regler

Bei nichtstetigen Reglern kann die Stellgröße nur eine geringe Anzahl fest definierter Werte annehmen. Sie sind äußerst primitiv und werden bei einfachen Anwendungen eingesetzt.

**Zweipunktregler** Ein Zweipunktregler ist ein Regler, der, wie der Name schon sagt, nur zwei Punkte/Zustände kennt. Mit ihm werden ganz einfache Regelkreise umgesetzt, bei denen es beispielsweise nur ein „AN“ und ein „AUS“ gibt. Beispielsweise wird bei einem Kühlschrank das Kühlaggregat eingeschaltet wenn man kühlen muss. Wenn die gewünschte Temperatur erreicht ist, wird das Aggregat abgeschaltet. Es gibt hier keine Zwischenzustände. Um allerdings ein ständiges Ein- und Ausschalten um den Grenzpunkt zu vermeiden gibt es eine sogenannte Hysterese. Dies bedeutet, dass der Grenzpunkt nicht exakt auch der Umschaltzeitpunkt ist. Wird beispielsweise eine Temperatur von 6°C gewünscht, so wird erst ab einer Temperatur von 7°C gekühlt und erst bei 5°C das Aggregat ausgeschaltet. Es ist also jeweils eine Tolleranz von 1°C gegeben. Abbildung 3.4 zeigt ein Beispiel eines Temperaturverlaufs eines Zweipunktregler.

**Dreipunktregler** Im Gegensatz zum Zweipunktregler besitzt der Dreipunktregler drei Zustände. Somit kann beispielsweise bei einem Elektromotor die Laufrichtung bestimmt

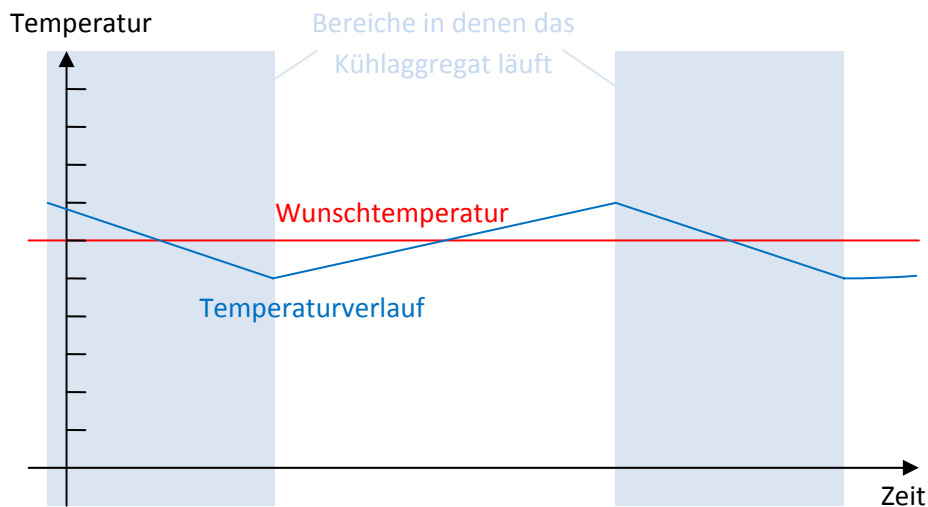


Abbildung 3.4: Beispiel eines Temperaturverlaufes eines Kühlschranks mit zweipunktgeregeltem Kühlaggregat

werden. In diesem Fall hätte der Regler die Zustände „Linkslauf“ - „Stopp“ - „Rechtslauf“. Auch hier gibt es eine Hysterese, so dass der Regler nicht permanent schalten muss.

**Mehrpunktregler** Zu guter Letzt gibt es bei den nichtstetigen Reglern auch solche, die mehr als drei Zustände besitzen. Die Funktionsweise ist in diesem Fall analog zu den beiden zuvor erläuterten Reglern.

### 3.2.2.2 Stetige Regler

Um eine kontinuierliche Regelung zu ermöglichen könnte man einen Mehrpunktregler entwickeln, der die gewünschte Anzahl an Zuständen besitzt. Für diese Zwecke gibt es aber auch die stetigen Regler, die unabhängig der definierten Zustände den kompletten Wertebereich abdecken können. Von den stetigen Reglern gibt es wiederum einige Typen, die in ihrem Regelverhalten auf unterschiedliche Dinge ausgerichtet sind.

**Proportionalregler (P-Regler)** Ein Proportionalregler regelt die Stellgröße immer proportional zur Amplitude der Regeldifferenz. Dies bedeutet, dass ein P-Regler sehr schnell auf eine Änderung reagiert. Allerdings reagiert er auch nur bei einer Änderung. Wenn sich allerdings im Laufe der Zeit kleine Fehler einschleichen, werden diese nie durch den P-Regler ausgeglichen. Mit Hilfe des Proportionalitätsfaktors des P-Reglers kann man die Intensität der Regelung steuern. Ein großer Proportionalitätsfaktor ermöglicht eine bessere Annäherung an einen Sollwert. Da die Stellgröße erhöht wird, erhöht sich allerdings auch

die Gefahr des Aufschaukelns. Ein kleiner Proportionalitätsfaktor bleibt stabiler, verhindert also das Aufschaukeln, erhöht aber die bleibende Regeldifferenz.

**Integralregler (I-Regler)** Beim Integralregler werden Regeldifferenzen zu jedem Zeitpunkt komplett ausgeglichen. Die Regelung endet erst dann, wenn die Abweichung zum Sollwert gleich Null ist, oder die maximale Stellgröße erreicht ist. Die Formel für die Berechnung der Stellgröße lautet

$$y = \frac{1}{T_N} \int e \, dt$$

wobei  $y$  = Stellgröße,  $T_N$  = Nachstellzeit und  $e$  = Regeldifferenz bedeutet. Daraus geht hervor, dass die Stellgröße proportional zum Integral des Regelfehlers ist. Die Nachstellzeit bestimmt, wie schnell die Angleichung an den Sollwert sein soll. Auch hier gilt, dass ein schnelles Angleichen, also eine kleine Nachstellzeit, ein Aufschaukeln bewirken kann. Stabiler, aber auch wesentlich träger, ist der Regelvorgang mit einer großen Nachstellzeit. Der Vorteil gegenüber dem P-Regler liegt darin, dass hier der Regelfehler ganz ausgeregelt wird.

**Differentialregler (D-Regler)** Ein Differentialregler reagiert nicht wie der P-Regler auf den Betrag eines Regelfehlers sondern auf dessen Änderungsgeschwindigkeit. Er regelt unabhängig von dessen Amplitude. So kommt es auch, dass eine kleine Regelabweichung eine große Stellgrößenänderung bewirken kann. Dies führt zu schnellem Aufschaukeln und Oszillieren. Da der D-Regler zudem auch bleibende Regelfehler bestehen lässt, wird er in der Praxis nie alleine eingesetzt, sondern nur mit anderen Reglerarten kombiniert. Diese werden nachfolgend vorgestellt.

**Proportional-Integral-Regler (PI-Regler)** Der PI-Regler ist ein kombinierter Regler mit einem proportionalen und einem integralen Regleranteil. Die beiden Regler werden parallel verschaltet. Somit kombiniert man quasi die Vorteile beider Arten und gleicht so auch die Nachteile der einzelnen Reglerarten aus. Bei diesem Regler wird das schnelle Regeln des P-Reglers mit der vollkommenen Regelung des I-Reglers kombiniert. Dies ergibt einen schnellen, stabilen und bis zur Regelabweichung Null regelnden Regler.

**Proportional-Differential-Regler (PD-Regler)** Beim PD-Regler wird die Stellgröße durch eine Parallelschaltung eines P-Reglers und eines D-Reglers verändert. Dies bewirkt im Vergleich zum P-Regler eine schnellere Annäherung an den Sollwert. Diese Regler kommen immer dort zum Einsatz, wo die P-Regler in ihrer Regeldynamik zu langsam sind.

**Proportional-Integral-Differentialregler (PID-Regler)** Der universale PID-Regler [22] vereint alle vorgestellten Reglerarten. Wie beim PD-Regler bewirkt auch hier der Differentialanteil eine Erhöhung der Regeldynamik im Vergleich zum PI-Regler. Der PID-Regler ist somit ein schneller, stabiler und bis zum Gleichgewicht ausgleichender Regler, der alle Vorteile der Reglerarten kombiniert.

### 3.2.2.3 Vergleich der Eigenschaften der Reglerarten

Reglerart	Regeldifferenz	Stellgeschwindigkeit
P-Regler	bleibend	schnell
I-Regler	keine	langsam
D-Regler	bleibend	sehr schnell
PI-Regler	keine	schnell
PD-Regler	bleibend	sehr schnell
PID-Regler	keine	sehr schnell

### 3.2.3 Sollwertfolgeregler

Der Regler, der eine Sollwertfolge realisiert, bekommt als Eingabe die Sollwertfolge und den aktuellen Istwert. Daraus berechnet der Regler die Regeldifferenz, also die Abweichung von Istwert zu Sollwert (P-Regler). Die Abweichung wird in Relation zu der zuvor berechneten Stellgröße gesetzt und eine neue Stellgröße berechnet. Die aus den Berechnungen des Reglers resultierende Stellgröße wird an die Regelkette weitergeleitet, die die entsprechende Anpassung, beispielsweise das Öffnen oder Schließen der Ventile einer Druckluftanlage, durchführt. Dabei verändert sich der Istwert. Zudem wird der Istwert von äußeren Einflüssen gesteuert. Dieser sich neu ergebende Istwert wird nun rückgeführt und dient dem folgenden Regelvorgang als Input. Dieser Regelkreis wird über die gesamte Laufzeit wiederholt.

#### Beispiel:

Der Sollwert beträgt 80. Der Regler hat als Stellgröße 1 eingestellt. Der Istwert beträgt 90. Somit ergibt sich eine Abweichung von Sollwert zu Istwert von 12,5% nach oben. Also muss die Stellgröße, um auf den Sollwert zu kommen, um 12,5% zum vorherigen Wert in die entsprechende Richtung abgeändert werden. In unserem Beispiel würde die Stellgröße um 12,5% nach unten korrigiert. Daraus ergibt sich eine neue Stellgröße von 0,875. Durch den Regelvorgang wird nun der Istwert an den Sollwert angenähert. Jedoch wird durch die äußeren Einflüsse der Istwert nach unten abgefälscht und beträgt nun bei der erneuten Eingabe in den Regler nur noch 60. Die aktuelle Abweichung zum Sollwert ist nun 25% nach unten. Somit ergibt sich aus der Berechnung des Reglers ein neuer Stellwert von 1,09. Die Stellgröße wurde um 25% erhöht. Mit dieser neuen Stellgröße sollte nun der Istwert wieder an den Sollwert angenähert sein.

### 3 Related Work

---

Dieses Verfahren ist sicherlich ganz interessant in Bezug auf die Adaption des TDF. Jedoch muss evaluiert werden, ob dieses Verfahren bei einer so schnellen Anpassung wie in unserem Fall überhaupt funktioniert. Da es auch praktisch nur äußere Einflüsse gibt, muss untersucht werden, ob die Regelung überhaupt auf eine solche Art berechnet werden kann. Ebenfalls ist zu untersuchen, ob bei einer so schnellen Adaption ein Aufschaukeln und Oszillieren entstehen kann. Problematisch kann auch hier sein, dass der Istwert von einem Moment zum nächsten von 0 auf 100 ansteigen kann. Dieser Regelmechanismus wird häufig in mechanischen Systemen verwendet, bei denen eine so schlagartige Änderung des Istwertes nicht auftreten kann. Jedoch ist die Grundidee sicherlich von großem Interesse bei dieser Aufgabe.

# Entwurfskriterien

---

Wie man im Kapitel „Related Work“ gesehen hat, gib es bereits einige Ansätze zum Thema Regelungstechnik und auch aus den anderen Gebieten lassen sich einige Ansätze wiederverwenden. In diesem Kapitel soll nun ein Einblick in die generell möglichen Optionen zur Anpassung des TDF gegeben werden.

## 4.1 Ermittlung von Lastveränderungen

Die Ermittlung einer Lastveränderung auf einem Knoten ist keine einfache Angelegenheit. Hierbei müssen die Eigenschaften der betrachteten Komponenten zur Lastermittlung beachtet werden. Als Beispiel wird hier die CPU-Last als ausschlaggebende Komponente für die Last des Knotens herangezogen.

Bei einem Prozessor besteht entweder Last oder keine Last. Entweder läuft gerade ein Prozess oder der Prozessor ist idle. Das bedeutet konkret, dass die Auslastung eines Prozessors digital ist. Je feiner man die Abtastrate der Last macht umso genauer kann man dies erkennen. Im Allgemeinen redet man bei einer Last des Prozessors immer vom Mittelwert über die Verwendung innerhalb einer gewissen Zeitspanne. Wenn der Prozessor in einer Sekunde 0,5 s verwendet wurde und 0,5 s idle war, hat er in dieser Sekunde 50% Auslastung gehabt. Je kleiner man die Intervalle wählt, umso unterschiedlicher sind die gemittelten Lastwerte in den betrachteten Intervallen. Je größer man die Intervalle macht, umso ähnlicher wird auch die durchschnittliche Last.

Abbildung 4.1 zeigt die Verteilung der Last über einige Takte des Schedulers hinweg. In den Takten in denen ein Prozess aktiv war besteht 100% Last. In den Takten ohne Last lief kein Prozess. Die Linie stellt dabei die durchschnittliche Last der CPU über alle gezeigten Takte dar.

Abbildung 4.2 zeigt die Verteilung der Last über einige Millisekunden hinweg. Die Säulen zeigen die durchschnittliche Last innerhalb dieser Millisekunden. Die Linien stellen dabei die durchschnittliche Last innerhalb der Intervalle (Länge der Linien) dar. Hier kann man erkennen, dass die Durchschnittswerte über längere Intervalle deutlich enger beieinander liegen, als die Durchschnittswerte über kleine Intervalle.

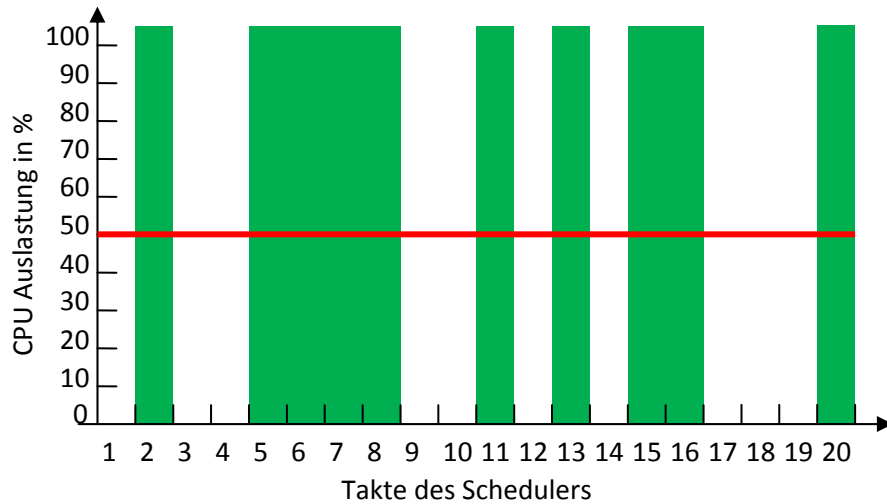


Abbildung 4.1: Übersicht über die Lastverteilung eines Prozessors während einiger Takte des Schedulers.

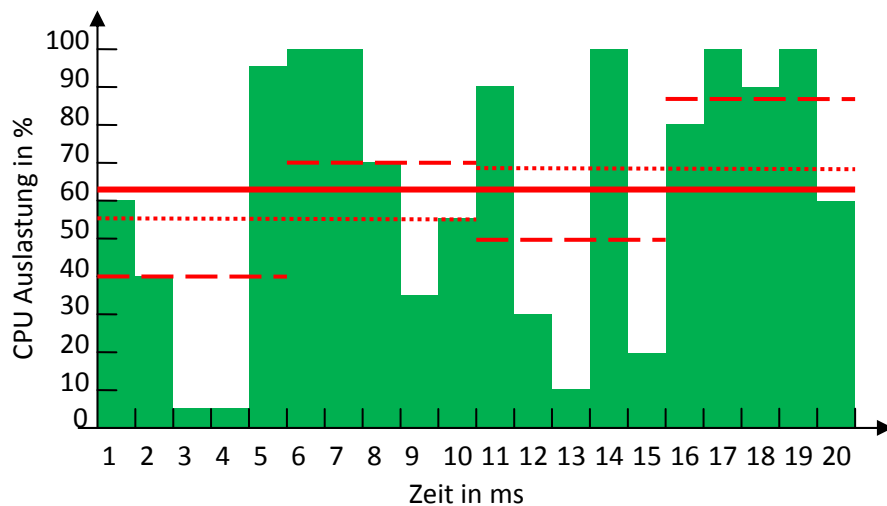


Abbildung 4.2: Übersicht über die Lastverteilung eines Prozessors während einiger Millisekunden

Für uns bedeutet dies, dass eine Abtastrate der Last von 1 ms sehr unterschiedliche und sich rasch ändernde Werte zum Vorschein bringt. Da eine Veränderung der Last unmittelbar auf die Adaption des TDF Auswirkungen haben kann, würde der TDF sehr häufig angepasst, obwohl es evtl. nicht unbedingt nötig und sinnvoll ist, da die Adaption, wie bereits beschrieben, auch eine Reaktionszeit hat (siehe Kapitel 2.6). Wenn man eine Abtastrate von 1.000 ms veranschlagt gleichen sich die Lastspitzen und Lasttäler weitgehend aus. Somit

erreicht man eine Adaption des TDF auf den Trend der Last hin und nicht auf einzelne Spitzenwerte. Dabei kann natürlich Überlast entstehen die das Experiment verfälscht. Man muss also einen Mittelweg zwischen der Genauigkeit der Adaption und dem Overhead für die Adaption finden.

### 4.2 Inhalt einer Lastnachricht

Eine Lastnachricht enthält verschiedene Informationen. Neben der Knoten ID des sendenden Knotens muss selbstverständlich auch die Last des Knotens im Paket enthalten sein. Diese Lastinformation kann auf verschiedene Weise übermittelt werden.

#### 4.2.1 Lastcodierung mit Lastzuständen

Die einfachste Form der Codierung einer Last besteht darin, einen definierten Lastzustand zu übermitteln. Dies bedeutet im konkreten Fall, dass man definierte Zustände hat, beispielsweise „Überlast“, „Unterlast“ und „Optimallast“, die im Lastnachrichtenpaket übermittelt werden.

Vorteilhaft bei dieser Lastcodierung ist, dass man in allen Komponenten nur die einzelnen Zustände behandeln und nicht den kompletten Wertebereich der Last abdecken muss. Wenn es vergleichsweise wenig Zustände gibt, ist dieser Weg relativ einfach. Allerdings ist diese Art der Lastcodierung relativ ungenau, da man die tatsächliche Last nicht genau kennt. Man weiß nur, dass sich die Last in einem gewissen Bereich befindet, allerdings nicht, ob am unteren oder am oberen Rand. Diese Information wäre für manche Operationen hilfreich.

#### 4.2.2 Lastcodierung mit realen Werten

Der ausführlichste Weg eine Lastinformation zu codieren besteht darin, den tatsächlichen Lastwert in der Lastnachricht zu übermitteln. Dazu wird einfach der genaue Lastwert in % in die Lastnachricht geschrieben und übermittelt.

Vorteilhaft bei dieser Lastcodierung ist, dass in jeder Nachricht der tatsächliche Lastwert enthalten ist. Der Empfänger bekommt so den genauen Lastwert mitgeteilt und kann diesen verarbeiten. Allerdings ist bei dieser Lastcodierung die Verarbeitung der empfangenen Lastinformation gegenüber der Codierung mit Zuständen aufwändiger. Man kann sich bei der Verarbeitung nicht auf die definierten Zustände beziehen, sondern muss den kompletten Wertebereich der Last bei der Verarbeitung abdecken.

### 4.2.3 Hybridcodierung

Der allgemeinste Fall der Lastcodierung in einer Lastnachricht besteht darin beide Formen der Codierung zu wählen. Wenn es definierte Bereiche gibt wird der entsprechende Zustand im Lastpaket mitgesendet. Zudem wird der tatsächliche Lastwert in der Lastnachricht übermittelt.

Vorteilhaft bei dieser ausführlichen Lastcodierung ist, dass man bei der Verarbeitung beide Möglichkeiten hat. Bei grobgranularen Entscheidungsprozessen kann der Zustand als Entscheidungskriterium herangezogen werden und wenn man den genauen Lastwert benötigt, kann dieser ebenfalls aus der Lastnachricht gezogen werden. Der Nachteil bei dieser Codierung ist allerdings, dass die Pakete größer werden und auch das Zusammenstellen und Trennen der Informationen erschwert wird. Zudem müssen bei jedem Weiterleitungsschritt beide Werte übertragen werden.

Prinzipiell wäre es möglich, nur den Lastwert zu verschicken und die Einteilung in die Lastbereiche auf dem Koordinator durchzuführen. Dies würde die Nachteile des höheren Sendeaufwandes neutralisieren. Allerdings muss dann der Koordinator für jedes Lastpaket diese Kategorisierung durchführen wodurch auf diesem Knoten Last erzeugt wird. Dies wird für große Systeme nicht skalieren, da der Koordinator selbst natürlich nicht mittels Zeitvirtualisierung gesteuert wird.

## 4.3 Regelverhalten

Die Reaktion auf eine Lastveränderung kann auf viele Arten geschehen. Hier werden einige wichtige Faktoren erläutert, die bei der Adaption beachtet werden müssen.

### 4.3.1 Einführung eines Optimal-Last-Wertes

Die Einführung eines Optimal-Last-Wertes ermöglicht es eine Abweichung zwischen dem aktuellen Istwert der Last und diesem Sollwert zu bestimmen. Anhand dieser Abweichung kann das Regelverhalten beeinflusst werden, so dass der aus der Adaption resultierende TDF auch die gewünschte Veränderung der Last auf den Optimalwert hin bewirkt. Dieses Verfahren wird beispielsweise in der Regelungstechnik häufig angewandt.

### 4.3.2 Sprunggröße

Abhängig vom signalisierten Lastwert können Werte definiert werden, um die der TDF in die entsprechende Richtung verändert wird. Je detaillierter die Lastinformationen sind, umso differenzierter kann auf eine Lastveränderung reagiert werden. Hier kann auch eine Differenzierung zwischen einem Sprung nach oben und einem Sprung nach unten eingeführt werden. Hierbei sollte berücksichtigt werden, dass die Behandlung einer Überlast

schneller und konsequenter durchgeführt werden muss, als eine Reaktion auf Unterlast. Der Grund für diese Unterscheidung liegt darin, dass der Adaptionsschritt bei einer Überlast so ausfallen sollte, dass der nächste gemessene Lastwert auf keinen Fall mehr Überlast aufweisen sollte. Bei der Reaktion auf eine Unterlast muss hingegen so adaptiert werden, dass auf keinen Fall zu viel geregelt wird, so dass das System nicht in Überlast gerät. Dies kann durch diese differenzierte Regelung erfolgen.

### 4.3.3 Größe der maximalen Sprunggröße

Da nicht alle Knoten ihre Virtuelle Zeit exakt im selben Moment an den neuen TDF umstellen können (Netzwerkübertragungszeit, Pufferung der Pakete,...) laufen die Zeiten der Maschinen bei jeder Anpassung minimal auseinander. Dies gleicht sich im Verlauf eines Experiments jedoch weitgehend aus, so dass es nur einen minimalen Einfluss auf die Gültigkeit der gesammelten Messdaten hat. Je größer aber der Betrag der Sprunggröße zwischen zwei Epochen ist umso mehr können die Maschinen auseinanderlaufen. Deshalb kann hier ein Maximalbetrag für eine Änderung des TDF eingeführt werden, um die Knoten auch bei Änderungen des TDF relativ synchron zu halten. Nachteilig ist bei diesem Verfahren, dass eine extreme Anpassung dadurch nicht mehr möglich wird und somit zwei Anpassungsschritte nötig werden können, wodurch die Knoten länger in Überlast betrieben werden und auch damit die Experimentergebnisse verfälscht werden können.

### 4.3.4 Extremwerte des TDF

Da der TDF unmittelbar die Virtuelle Zeit eines Knotens beeinflusst ist es wichtig, eine maximal sinnvolle und minimal sinnvolle Virtuelle Zeit und somit einen minimalen und einen maximalen TDF anzugeben. Sollte die Zeit in der Virtuellen Maschine zu schnell laufen, kann es passieren, dass eine Remoteverbindung per TCP von Extern nicht mehr möglich ist, da es zu einem Timeout kommt. Eventuell können hierfür eigene Protokolle über UDP entwickelt werden.

Bei zu langsamer Virtueller Zeit kann es passieren, dass es nicht mehr möglich ist, mit dem System interaktiv zu arbeiten. Zudem wird die Laufzeit eines Experiments durch einen sehr hohen TDF sehr verlängert, so dass es sehr lange dauern kann, bis Ergebnisse vorliegen. Sollte bei der langsamsten Zeit immer noch Überlast auftreten muss man andere Wege zur Korrektur (beispielsweise mehr Hardware einsetzen) suchen. Der maximale TDF bestimmt hierbei den Faktor, um den die Zeit maximal gedehnt, also verlangsamt wird. Der minimale TDF bestimmt hierbei den Faktor, um den die Virtuelle Zeit beschleunigt wird.

### 4.3.5 Häufigkeit einer Anpassung

Je häufiger man den TDF an die aktuelle Last der Knoten anpasst umso perfekter kommt man im Gesamten gesehen an die Optimallast und somit an den besten Experimentverlauf. Allerdings benötigt jede Anpassung auch Zeit und verbraucht Ressourcen. Es müssen

die entsprechenden Lastinformationen gesammelt und verarbeitet werden. Der optimale TDF muss berechnet und danach an alle Verbraucher gesendet werden. Diese Aufgaben verbrauchen zwar nicht viel Netzwerkbandbreite und Computingpower jedoch macht es in der Summe doch schon einen merklichen Anteil am Gesamtverbrauch der Ressourcen aus. Somit muss herausgefunden werden, wie häufig und vor allem wann eine Anpassung sinnvoll ist, um nicht durch eine zu häufige Anpassung zu viel Ressourcen und Zeit zu verbrauchen, was sich dann auf die Experimentlaufzeit niederschlagen kann.

### 4.4 Berücksichtigung des aktuellen Zustandes

Das Sammeln von Lastinformationen und die Adaption auf eine gegebene Last können unter Umständen anhand des aktuellen Zustandes der im System herrscht variiert werden. Beispielsweise muss eine Überlast in Echtzeit (TDF = Null) deutlich schneller erkannt und darauf entsprechend reagiert werden, als wenn die Virtuelle Zeit wesentlich langsamer läuft. Eine Überlast, die beispielsweise 2 ms bei Virtueller Zeit = Realzeit anliegt, bedeutet, dass 2 ms eine Überlast anliegt. Eine 2 ms andauernde Überlast, die bei Virtueller Zeit = Realzeit / 100 anliegt, bedeutet ja für das laufende Experiment nur eine Überlast von 0,02 ms - ist also wesentlich unkritischer als bei Betrieb in Echtzeit. Somit können einige Parameter, abhängig von der aktuellen Virtuellen Zeit, also dem aktuellen TDF, entsprechend verstärkt oder verringert werden, so dass eine optimale Regelung durchgeführt werden kann. Zudem wird der Overhead der TDF-Adaption durch die Anpassung an den aktuellen Zustand verringert, da beispielsweise weniger Lastnachrichten gesendet werden können.

### 4.5 Berücksichtigung der Zeit

#### 4.5.1 Berücksichtigung der Vergangenheit

Hierbei soll erörtert werden, ob es möglich und sinnvoll ist, bei der TDF Berechnung Informationen aus der Vergangenheit mit zu berücksichtigen. Somit kann zum Beispiel bei schnell oszillierenden Lastverläufen durch die Auswahl eines geeigneten TDF verhindert werden, dass ständig eine Anpassung des TDF nötig ist. Dadurch spart man den Aufwand, der zum Wechseln der Epoche nötig ist, riskiert aber, dass die Experimentlaufzeit durch die Wahl eines höheren TDF als nötig, unnötig in die Länge gezogen wird. Zudem ist es schwer herauszufinden wann ein Lastverlauf so geartet ist, dass man Rückschlüsse aus der Vergangenheit machen kann. Zudem kann bei den Lastspitzen eine Überlast entstehen, die die Messergebnisse verfälscht.

#### 4.5.2 Voraussagen über die Zukunft

Dieser Punkt ist sehr eng mit dem vorherigen Punkt „Berücksichtigung der Vergangenheit“ gekoppelt. Um Voraussagen über die Zukunft machen zu können benötigt man Wissen

aus der Vergangenheit. Generell stellt sich die Frage ob es überhaupt möglich ist, etwas über die Zukunft voraussagen zu können. Man kann ja nie wissen, wie sich die Last in Zukunft ändern wird. Wenn überhaupt, kann die Voraussage nur eine sehr kleine Hilfe sein, um einen optimalen TDF zu bestimmen. Die Grundannahme auf der die Adaption generell basiert ist die, dass der nächste gemessene Lastwert ähnlich dem zuvor gemessenen Lastwert ist. Die Adaption muss so regeln, als wäre die durch das Experiment erzeugte Last im nächsten Schritt dieselbe. Nur durch diese simple Grundannahme kann eine Adaption überhaupt stattfinden. Eine weiterreichende Voraussage über die Zukunft kann nur sehr schwer getroffen werden.

#### **4.6 Problematik der Ausführung eines Experiments schneller als Echtzeit**

Durch die Adaption ist es möglich, einen negativen TDF einzustellen und dadurch die Virtuelle Zeit schneller als Echtzeit laufen zu lassen und somit die Ausführung eines Experiments extrem zu beschleunigen. Dies birgt aber einige Gefahren. Lastschwankungen sollten schnell erkannt und durch die Adaption ausgeglichen werden. Da eine Überlast immer auf Kosten der Korrektheit der gemessenen Ergebnisse eines Experiments geht und das Experiment als Zeit nur die Virtuelle Zeit kennt, muss also auch die Lastschwankung in Virtueller Zeit berücksichtigt werden. Dies bedeutet im konkreten Fall, dass eine Überlast schnell in Virtueller Zeit erkannt werden muss. Da wir aber nur in Echtzeit überwachen und adaptieren können ist es sehr gefährlich das Experiment schneller als Echtzeit ablaufen zu lassen. Dadurch beschleunigt sich das Auftreten einer zu falschen Ergebnissen führenden Überlast, die evtl. gar nicht erkannt werden kann, da sie zu schnell für die Überwachung in Echtzeit ist und bereits vorbei sein kann, bevor sie erkannt wird. Deshalb sollte ein Experiment nur so schnell ablaufen, wie es überwacht werden kann. Eine schnellere Ausführung ist möglich, sollte aber nur zu Testzwecken ausgeführt werden. Die in dieser Diplomarbeit entwickelten Konzepte versuchen eine optimale Adaption bei positivem und negativem TDF zu realisieren. Allerdings sei bei der Ausführung eines Experiments immer auf diese Gefahren hingewiesen.



# Architektur des TDF-Adapters

---

Die Architektur von TVEE sieht die Komponenten LoadMonitor, TDF-Adapter und den EpochSwitcher vor. Diese Komponenten laufen auf verschiedenen Knoten. Somit ist TVEE ein Verteiltes System.

In diesem Kapitel werden nun die einzelnen Komponenten des TDF-Adapters und ihre Funktionsweise und Aufgabe erklärt. Die einzelnen Komponenten werden ebenfalls auf verschiedenen Knoten laufen, somit beschreibt auch die Architektur des TDF-Adapters ein Verteiltes System.

## 5.1 Komponenten des TDF-Adapters

Der TDF-Adapter muss insgesamt drei Aufgaben erfüllen.

- Die erste Aufgabe besteht darin, aus einer Lastinformation, die lokal auf einem Hardwareknoten vom LoadMonitor mitgeteilt wird, die Relevanz für die Adaption zu bestimmen. Diese Aufgabe ist auf dem Hardwareknoten direkt durchzuführen, bevor ein Lastinformationspaket über das Netzwerk verschickt wird. Dazu gibt es das **Relevanz-Modul**, das direkt auf dem Hardwareknoten läuft. Sollte die aktuelle Lastinformation relevant für die TDF-Adaption sein, so wird diese über das Netzwerk zum Zentralen Koordinator geschickt.
- Auf dem Zentralen Koordinator läuft das **Global-State-Modul**. Dieses sammelt von allen am Experiment beteiligten Knoten die gesendeten Lastinformationen und ermittelt aus den gesammelten Daten einen Global-State.
- Dieser Global-State wird nun zum eigentlichen **Adaptions-Modul** geleitet. Dieses berechnet aus der neuen Information mit Hilfe eines Regelalgorithmus den neuen TDF. Dieser wird dann an den EpochSwitcher weitergeleitet, der die Einstellung des TDF im gesamten System übernimmt.

## 5 Architektur des TDF-Adapters

Für den Transport der Nachrichten über das Netzwerk wird ein speziell hierfür entwickeltes schnelles Protokoll verwendet, das bestimmte Quality of Services für diese Art von Paketen bereit stellt. Dieses Protokoll wurde im Zusammenhang mit der Diplomarbeit [11] für den EpochSwitcher entwickelt. Abbildung 5.1 zeigt die Architektur des TDF-Adapters.

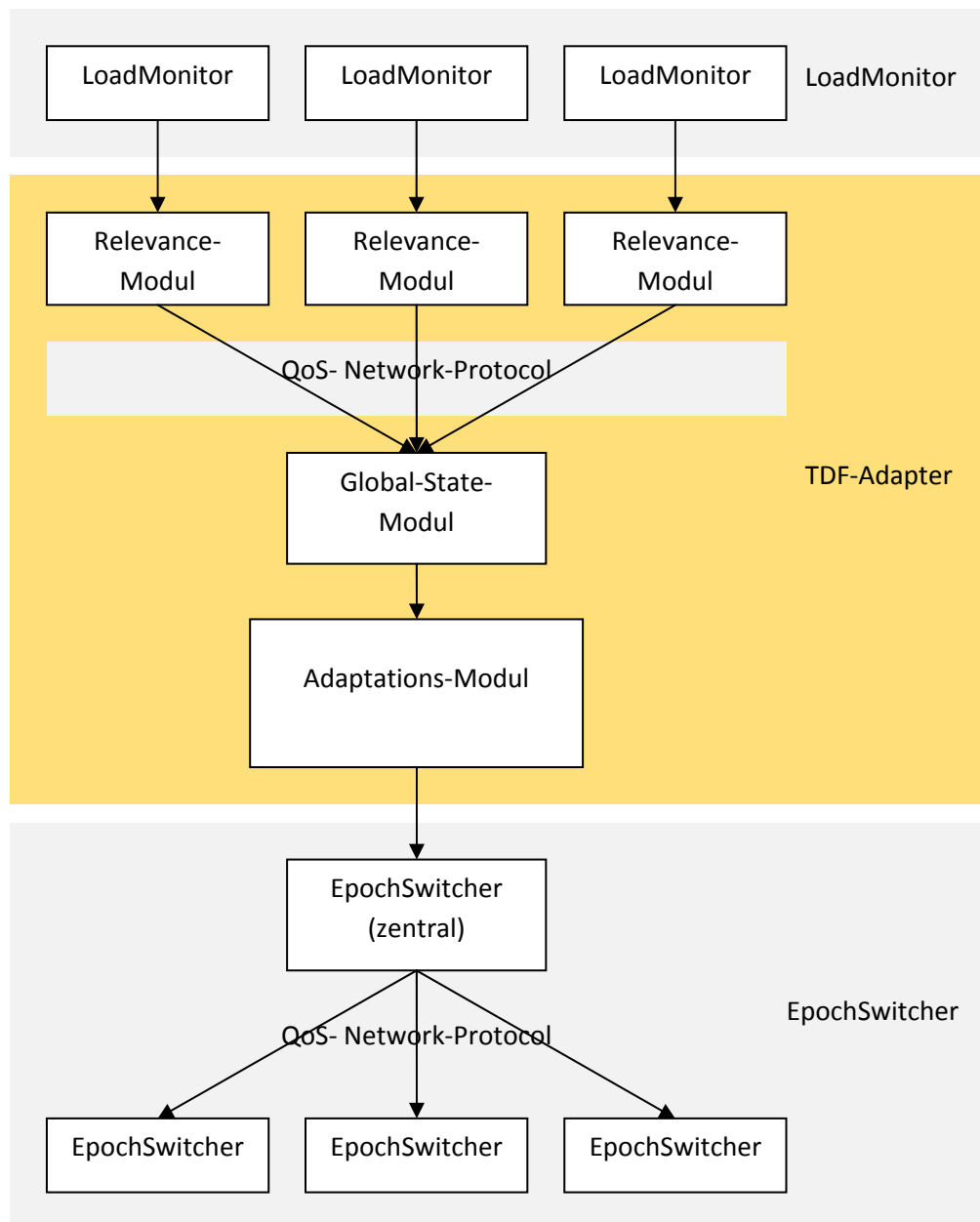


Abbildung 5.1: Architektur des TDF-Adapters

Aus dieser Architektur kristallisieren sich zwei wichtige Themen, die in dieser Diplomarbeit bearbeitet werden, heraus:

- **Relevanz einer Lastinformation**  
Bei dieser Aufgabe geht es darum, eine geeignete Logik zu entwickeln, die die ermittelten Lastinformationen nach ihrer Relevanz für die TDF-Adaption bewertet und weiterleitet. Es kann nicht jede Lastinformation bei der TDF-Adaption beachtet werden, da dieser Weg für große Systeme nicht skaliert. Es würden viel zu viele Lastinformationen der einzelnen Knoten gesendet werden. Somit muss eine Logik entwickelt werden die, passend zur Adaptionlogik, nur die wichtigsten und relevanten Lastinformationen kommuniziert.
- **geeignete Adaption des TDF als Reaktion auf den Global-State**  
Hierbei geht es darum, den TDF als Reaktion auf den ermittelten Global-State so anzupassen, dass die globale Last optimal wird. Optimal bedeutet hierbei, dass die Last in einem hohen Bereich verläuft, allerdings keine Überlast auftritt. Die Anforderungen an diese Adaption sind bereits in der Einführung zu TVEE (Kapitel 2.8) erläutert worden. Hier soll nur darauf verwiesen werden.

### 5.1.1 Relevanz-Modul

Das Relevanz-Modul bekommt periodisch den aktuellen Lastwert des Knotens vom Load-Monitor mitgeteilt. Abhängig von diesem Wert und der hinterlegten Logik entscheidet dieses Modul nun, ob die aktuelle Lastinformation relevant für die TDF-Adaption ist. Je nach dem welche Informationen in einer Lastnachricht enthalten sind und welche Adaptionlogik verwendet wird, kann es unterschiedliche Relevanzkriterien geben. Diese werden im Kapitel 5.2 „Entwurf des Relevanz-Moduls“ vorgestellt.

Ist die Relevanz des neuen Lastwertes festgestellt worden, so wird die neue Lastinformation über das schnelle Netzwerkprotokoll an den Zentralen Koordinator gesendet. Dort empfängt und bearbeitet das Global-State-Modul die Pakete aller Knoten des Experiments.

### 5.1.2 Global-State-Modul

Das Global-State-Modul bekommt von allen am Experiment beteiligten Systemen die für die Adaption relevanten Lastinformationspakete. Aus diesen vielen Informationen wird der Global-State berechnet. Sobald sich dieser durch eine neue Nachricht ändert, wird der neue Global-State an das Adaptions-Modul weitergereicht.

### 5.1.3 Adaptions-Modul

Das Adaptions-Modul ist die eigentliche Kernkomponente des TDF-Adapters. Dieses Modul beinhaltet die Logik zum Adaptieren des TDF als Reaktion auf den gegebenen Global-State. Das Adaptions-Modul bekommt vom Global-State-Modul bei jeder Änderung des globalen

Lastzustandes selbigen und adaptiert darauf den TDF, so dass die Last idealerweise wieder auf den Optimalwert verändert wird. Hierbei gibt es mehrere Möglichkeiten wie eine solche Adaption realisiert werden kann. Diese werden im Kapitel 5.4 „Entwurf des Adaption-Moduls“ vorgestellt.

## 5.2 Entwurf des Relevanz-Moduls

### 5.2.1 Architektur

In diesem Abschnitt wird die Logik im Relevanz-Modul entwickelt. Abbildung 5.2 zeigt die Architektur des Relevanz-Moduls. Die Pfeile symbolisieren den Datenfluss der Lastinformation. Die Breite der Pfeile soll die Anzahl der Nachrichten verdeutlichen.

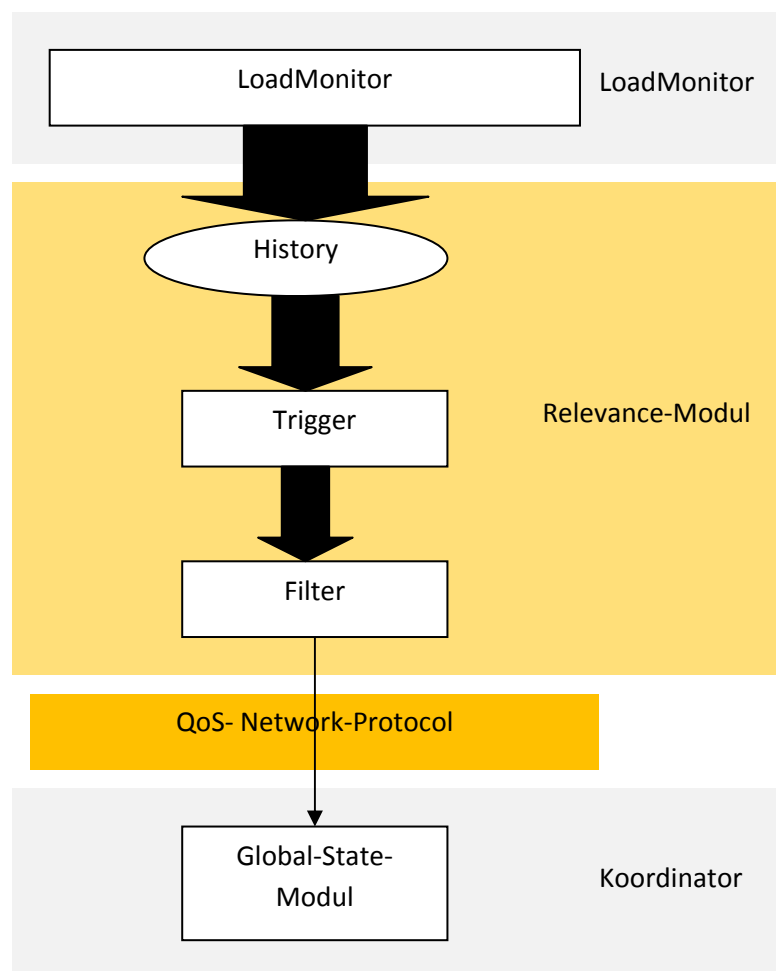


Abbildung 5.2: Architektur des Relevanz-Moduls

Das Relevanz-Modul bekommt periodisch vom LoadMonitor die aktuelle Last des Knotens mitgeteilt. Diese Informationen werden in einer History gespeichert, so dass bei Bedarf auf einen Teil der Vergangenheit (beispielsweise eine Sekunde) zurückgegriffen werden kann. Der sogenannte Trigger bestimmt, wann eine Lastinformation berücksichtigt wird. Der ermittelte Lastwert wird anschließend an den sogenannten Filter weitergereicht, der die Relevanz für die Adaption ermittelt und dann die Lastnachricht an den Zentralen Koordinator zum Relevanz-Modul sendet.

### 5.2.2 Kontinuierliche Lastsignalisierung (Spezialfall)

Um zu jeder Zeit die exakte Last im System zu kennen kann man eine kontinuierliche Lastsignalisierung verwenden. Hierbei wird jede Lastveränderung dem TDF-Adapter mitgeteilt. Somit kann dieser immer auf die aktuellen Werte reagieren und den TDF zu jedem Zeitpunkt optimal bestimmen. Problematisch ist hier der Overhead, der durch die vielen Lastnachrichten entsteht. Wenn viele Knoten am Experiment beteiligt sind, werden permanent von jedem Knoten Lastinformationspakete zum TDF-Adapter gesendet. Dadurch wird der Koordinator überlastet und kann die anfallende Datenflut nicht bewältigen. Somit kann er nicht mehr zuverlässig reagieren. Diese Signalisierung skaliert nicht für große Systeme. Neben dem Overhead spielt auch die in Kapitel 4.1 erläuterte Eigenschaft der Lastveränderung eine wichtige Rolle, die dieses Konzept zum Scheitern bringt. Wenn tatsächlich binäre Lastinformationen als Grundlage zur Adaption dienen sollen, wird das System keine sinnvolle Adaption erwirken. Bei unserem System findet diese Variante daher keine Verwendung.

### 5.2.3 Trigger

Trigger bestimmen, wann die Last eines Knotens ermittelt wird. Dies kann timergesteuert geschehen oder eventgesteuert. Der ermittelte Lastwert wird hierauf an einen Filter weitergereicht.

#### 5.2.3.1 Timergesteuerter Trigger

Beim timergesteuerten Trigger hat man die Möglichkeit einen Timer anzugeben, nach dessen Ablauf, die in dieser vergangenen Zeit anliegende mittlere Last ermittelt wird. Die Last wird also in periodischen Zeitabständen ermittelt. Je enger man die Intervalle wählt, um so öfter werden Lastinformationen ermittelt und zum Filter weitergegeben. Das Timerintervall bestimmt hier die maximale Anzahl an Nachrichten, die von einem Knoten zum Koordinator gesendet werden kann.

Hier kann zum ersten Mal die Berücksichtigung des aktuellen Zustandes eine Rolle spielen. Im Prinzip wird hier definiert, wie schnell eine Lastschwankung erkannt wird. Wenn das Timerintervall unveränderlich ist, wird die Last alle X Millisekunden der Echtzeit ermittelt. Interessanter ist allerdings, wie lange es braucht, eine Lastschwankung in Virtueller Zeit zu

erkennen. Wenn der TDF bereits bei 500 steht, läuft die Virtuelle Zeit 32 Mal langsamer als Echtzeit. Somit kann das Intervall prinzipiell um das 32-fache verlängert werden. Damit wäre eine Lastschwankung nach X Millisekunden in Virtueller Zeit erkannt. Da eine proportionale Steigung der Virtuellen Zeit und des Timerintervalls bedeuten würde, dass man das Timerintervall auch extrem dehnt, kann bei hohem TDF keine sinnvolle Erkennung von Lastschwankungen mehr erfolgen (Beispiel: Timerintervall bei TDF 0 = 4 ms -> Timerintervall bei TDF 1000 = < 4 s). Deshalb muss eine Obergrenze für das Timerintervall definiert werden. Die Formel zur Berechnung des aktuellen Timers lautet:

$$\text{aktuellerTimer} = \text{MIN}(2^{\text{aktuellerTDF}/100} * \text{MinTimer}, \text{MaxTimer})$$

Dies bedeutet, dass wir das Timerintervall mit jeder Verdopplung der virtuellen Zeit ebenfalls verdoppeln. Allerdings nur bis zum maximalen Timerwert. Zudem sollte die Adaption an den aktuellen Zustand nur bei positivem TDF erfolgen. Hierbei sei noch einmal auf die Problematik mit negativem TDF hingewiesen (siehe Kapitel 4.6).

### 5.2.3.2 Eventbasierte Trigger

Der timergesteuerte Trigger hat den Nachteil, dass er starr und nicht flexibel ist. Deshalb kann es beim timergesteuerten Trigger bei zu langem Timerintervall vorkommen, dass eine spontan auftretende Überlast nicht oder erst sehr spät erkannt wird, da sie kurz nachdem eine timergesteuerte Last getriggert wurde auftrat. Um dieses Problem zu lösen kann man das Auftreten bestimmter Events als Trigger verwenden. Ein Event ist hierbei beispielsweise das Auftreten einer Überlast oder Unterlast. Die einzelnen Bereiche werden mittels Schwellenwerten voneinander getrennt. Hierbei kann definiert werden, wie lange das entsprechende Event ohne Unterbrechnung auftreten muss, damit das Event kommuniziert wird. Ist das Event entsprechend lange aufgetreten, wird der entsprechende Lastwert ermittelt und an den Filter weitergegeben. Hier bestimmt die Länge der Events die maximale Anzahl an Lastnachrichten, die zum Koordinator gesendet werden können. Die Dauer des Auftretens eines Events kann anhand des aktuellen Zustands variiert werden. Es gilt, dass ein Event an der Dauer des Auftretens in Virtueller Zeit bewertet werden muss und nicht nach der Dauer des Auftretens in Echtzeit. Deshalb wird mit der Formel

$$\text{aktuelleEventDauer} = \text{MIN}(2^{\text{aktuellerTDF}/100} * \text{MinEventDauer}, \text{MaxEventDauer})$$

die aktuelle Dauer eines Events berechnet.

### 5.2.4 Filter

Filter bekommen Lastwerte vom Trigger und werten diese nach der Relevanz für die Adaption aus. Hierbei gib es mehrere Varianten:

### 5.2.4.1 Ohne Filter

Wenn kein Filter verwendet wird, wird umgehend jeder getriggerte Lastwert zum zentralen Koordinator gesendet. Der Vorteil hierbei ist, dass wirklich jeder getriggerte Lastwert beim Koordinator ankommt und man somit die höchste Genauigkeit beim Bilden des Global-State hat. Der Nachteil besteht darin, dass eventuell sehr viele unnötige Lastnachrichten gesendet werden.

### 5.2.4.2 Differentieller Filter

Beim Differentiellen Filter werden die Lastinformationen nur dann gesendet, wenn sie sich um einen gewissen Betrag zur vorherigen Lastsignalisierung geändert haben. Dies kann beispielsweise jedes Mal dann der Fall sein, wenn sich die Last eines Knotens um 5% zum vorher gesendeten Lastwert ändert. Bei dieser Variante bekommt man, im Vergleich zur filterlosen Variante, einen ungenaueren Überblick über die aktuelle Systemlast. Allerdings hat man auch nur eine geringe Menge an Lastsignalisierungen, wodurch wenig Aufwand zur Kommunikation betrieben wird. Je enger man diese Lastdifferenzintervalle macht, umso genauer kann der Global-State bestimmt werden, es sind aber auch wesentlich mehr Lastnachrichten zu verarbeiten.

### 5.2.4.3 Schwellenwert-basierter Filter

Ein spezieller Fall des Differentiellen Filters ist das Filtern mit Schwellenwerten. Hierbei wird der komplette Lastbereich eines Knotens in einzelne Bereiche eingeteilt. Diese Bereiche werden durch Schwellenwerte voneinander getrennt. Jedes Mal wenn die getriggerte Last einen Schwellenwert durchläuft, wird eine Lastinformation weitergeleitet. Man hat also nur Lastnachrichten, wenn sich die Last in einen anderen Lastteilbereich bewegt. Auch hier gilt, je mehr Bereiche man wählt umso detailliertere Informationen über die Last sind verfügbar. Jedoch nimmt auch die Anzahl der Lastsignalisierungen damit zu.

## 5.3 Entwurf des Global-State-Moduls

Das Global-State-Modul bekommt von den einzelnen Knoten alle als relevant festgestellten Lastinformationen zugeschickt. Diese werden in einer Liste der einzelnen Knoten gespeichert und bei jeder Änderung entsprechend aktualisiert. In diesem Modul ist es nur notwendig für jeden Knoten den aktuellen Lastwert vorzuhalten. Eine Historie vorzuhalten ist hier nicht sinnvoll, da nur der globale Systemzustand, unabhängig des Lastverlaufes eines einzelnen Knotens, ermittelt wird. Bei jedem Eintreffen eines neuen Lastpakets wird der Eintrag des entsprechenden Knotens aktualisiert. In jeder Millisekunde wird nun der Global-State berechnet. Sollte sich dieser ändern, wird der neue Global-State an das Adaption-Modul weitergeleitet.

## 5.4 Entwurf des Adaption-Moduls

Im Adaption-Modul befindet sich der Regelalgorithmus der den optimalen TDF für den gegebenen Global-State berechnet. Hierfür gibt es mehrere Ansätze die hier im Einzelnen vorgestellt werden.

### 5.4.1 TCP-basierter Ansatz

Beim TCP-basierten Ansatz wird als Grundkonzept das Konzept von TCP herangezogen und verfeinert, bzw. an die Bedürfnisse der TDF-Adaption angepasst.

#### 5.4.1.1 Signalisierung der Last

Dieses Konzept reagiert auf Signalisierung der Last mittels Lastzuständen. Abhängig vom Global-State, der vom Global-State-Modul ermittelt wird, wird der TDF entsprechend adaptiert. Hierbei hängt die Logik des Relevanz-Moduls eng mit der Logik des Adaption-Moduls zusammen. Alle Stati, die das Relevanz-Modul (Schwellenwert-basierter Filter) definiert hat, müssen vom Adaption-Modul behandelt werden können. Somit besteht hier eine enge Kopplung. Die Bereiche/Stati, die von beiden Logiken unterstützt werden, sind im Einzelnen:

- Überlastbereich (*OVERLOADED*) - befindet sich das System in diesem durch Schwellenwerte abgetrennten Bereich, befindet es sich im Status in dem das System überlastet ist. Es muss sehr zügig und massiv auf die Überlast reagiert werden.
- Bereich potentieller Überlast (*POSSIBLE\_OVERLOAD*) - dies ist der Bereich, in dem eine potentielle Überlast existiert. Es muss bereits hier der drohenden Überlast entgegengesteuert werden.
- Optimalbereich (*OPTIMAL*) - wenn sich die Last in diesem Bereich befindet ist alles in Ordnung und eine Anpassung des TDF ist nicht notwendig. Allerdings kann hier durch eine geeignete Adaption die Last an die Obergrenze dieses Bereiches angenähert werden, so dass das System eine höhere Last hat, allerdings sich immer noch im Status *OPTIMAL* befindet.
- Unterlastbereich (*UNDERLOAD*) - wenn sich die Last in diesem Bereich befindet, bedeutet es, dass das System nicht optimal ausgelastet ist und eine höhere Virtuelle Zeit also ein niedrigerer TDF gefahren werden kann.

Abbildung 5.3 zeigt die Lastbereiche beim TCP-basierten Ansatz.

Als Trigger können beide vorgestellten Konzepte (5.2.3) verwendet werden, da sie keinerlei Aussage und Einschränkung bzgl. der Genauigkeit einer Lastnachricht treffen. Als Filter kann hier nur der Schwellenwert-basierte Filter eingesetzt werden, da nur er die nötigen Bereiche definiert. Die Variante ohne Filter macht keinen Sinn, da Aufgrund der Eigenschaft dieses Konzeptes auf jeden Fall eine Einteilung in die Bereiche stattfinden muss. Dadurch

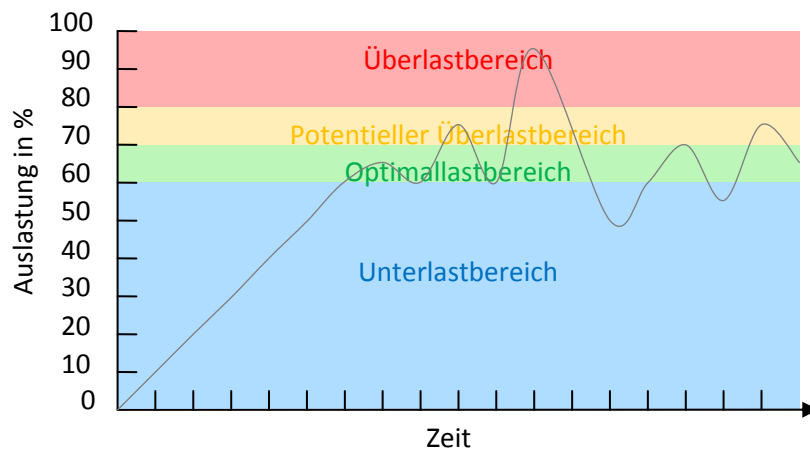


Abbildung 5.3: Übersicht über den eingeteilten Lastbereich beim TCP-basierten Ansatz

kann es sein, dass gesendete Lastwerte nicht beachtet werden. Somit kann man sich diesen Aufwand sparen und gleich den Schwellenwert-basierten Filter verwenden.

#### 5.4.1.2 Regelalgorithmus

Der Regelalgorithmus des TCP-basierten Ansatzes muss auf alle definierten Stati reagieren können. Für jeden definierten Status gibt es eine Behandlung, die den TDF so adaptieren soll, dass die Last wieder im Optimalbereich zu liegen kommt. Zu Beginn der Adaption muss eine Initiale Adaption stattfinden, die, ähnlich wie bei TCP, mit einem niedrigen Wert anfängt und sich langsam an den optimalen Wert herantastet. Dies bedeutet, dass der TDF zu Beginn auf den maximalen Wert gesetzt und langsam verringert wird. Des weiteren muss es eine Behandlung gegen Unterlast, potentielle Überlast und Überlast geben. Die Reaktion auf Unterlast wird, wie bereits beschrieben, weniger stark ausfallen, als die Reaktion auf Überlast. Zudem gibt es für den Optimallastbereich eine Feintuning. Dieses sorgt dafür, dass die Last im Optimallastbereich am oberen Rand dieses Bereiches verläuft, so dass die Ausführungszeit eines Experiments dadurch etwas beschleunigt wird. Die Algorithmen und die konkrete Umsetzung des TCP-basierten Ansatzes werden im Kapitel 6 „Implementierung“ vorgestellt.

#### 5.4.1.3 Timer

Die Anpassung des TDF wird grundsätzlich durch eine Änderung des Systemstatus angestoßen, die vom Global-State-Modul signalisiert wird. Sollte allerdings ein Adaptionsschritt nicht genügen um das gewünschte Ziel (Statusänderung) zu erreichen, so müssen ein oder mehrere Adaptionsschritte folgen. Da speziell zu Beginn (bei der Initialen Adaption) keine Statusänderungen zu erwarten sind, ist es unabdingbar einen Timer zu definieren, der eine

periodische Anpassung, auch ohne Interrupt vom Global-State-Modul, anstößt. Allerdings muss hierbei die Reaktionszeit einer TDF Anpassung berücksichtigt werden. Deshalb sollte dieser Timer nicht zu niedrig sein, da sonst evtl. eine erneute Adaption angestoßen wird, obwohl das Feedback der zuvor angestoßenen Adaption noch gar nicht da sein kann. Da dieser Timer vom Triggertimer abhängt (Feedback der vorgehenden Adaption) muss dieser Timer ebenfalls an den aktuellsten Zustand angepasst werden. Dies kann durch die folgende Formel geschehen. Für den Timer muss ebenfalls eine obere Grenze definiert werden.

$$\text{aktuellerTimer} = \text{MIN}(2^{\text{aktuellerTDF}/100} * \text{MinTimer}, \text{MaxTimer})$$

### 5.4.2 Regler-basierter Ansatz

Beim Regler-basierten Ansatz werden Konzepte aus der Regelungstechnik herangezogen und bekannte Regler aus der Regelungstechnik verwendet. In diesem Ansatz wird ein PI-Regler verwendet.

#### 5.4.2.1 Signalisierung der Last

Im Gegensatz zum TCP-basierten Ansatz wird hier nicht auf Lastzustände, sondern auf den genauen Lastwert reagiert. Dies bedeutet, dass wir immer exakt so weit korrigieren können, so dass der Lastwert nach der TDF Adaption theoretisch wieder im Optimallastbereich zu liegen kommt. Abhängig davon, ob sich das System in Unter- oder Überlast befindet wird die entsprechende Behandlung aktiviert. Dieses Konzept kennt nur drei Zustände (*UNDERLOADED*, *OVERLOADED* und *OPTIMAL*). Prinzipiell kann der Sollwert als exakte Grenze zwischen Unter- und Überlastbereich fungieren, allerdings bedeutet dies, dass der TDF extrem häufig angepasst werden würde. Deshalb wird ein Optimallastbereich um den Sollwert definiert.

Dieser Optimallastbereich ist abhängig vom verwendeten Filter. Bei der Variante ohne Filter, ist dieser Bereich nur der Optimallastwert selbst. Beim Differentiellen Filter orientiert er sich am eingestellten Differenzwert des Filters. Beim Schwellenwertbasierten Filter stellen die dort eingestellten Schwellenwerte auch beim Regler die Schwellenwerte für den Optimallastbereich dar.

Als Trigger können beide vorgestellten Konzepte (5.2.3) dienen, da sie keine Unterscheidung zwischen Lastsignalisierung und Statussignalisierung machen.

#### 5.4.2.2 Reglerbereich

Das Problem bei der Initialisierungsphase eines (mechanischen) Regelkreises besteht darin, dass zu Beginn noch kein realer Istwert anliegt. Der Istwert und der Sollwert liegen sehr weit auseinander. Somit berechnet der Regler eine extreme Regeldifferenz und steuert somit den Regelkreis stark in Richtung des Optimalwertes. Wenn sich nun der Istwert

anwendungsbedingt an den Sollwert annähert und der Regler zudem eine starke Regelung in diese Richtung anstößt, kann das System schnell überlastet werden, bzw. der Regler muss schnell wieder in die entgegengesetzte Richtung regeln. Dies bewirkt ein unnötiges Schwingen zu Beginn der Inbetriebnahme des Regelkreises.

Aus diesem Grund definieren wir einen Bereich, indem der Regler überhaupt nur tätig wird. Befindet sich in unserem Fall die Last zu weit oberhalb oder zu weit unterhalb der Optimallast werden andere Mechanismen zur Regelung herangezogen. So gesehen ist dieser Ansatz im Kern nur eine Feinregelung. Daraus resultierend muss ein Verfahren definiert werden, das ausserhalb dieses Reglerbereiches greift. In unserem Fall wird dieser Bereich nur bei zu starker Unterlast liegen, so dass gerade bei der Initialisierungsphase das initiale Verfahren greift.

Abbildung 5.4 zeigt die Bereiche beim Regler-basierten Ansatz.

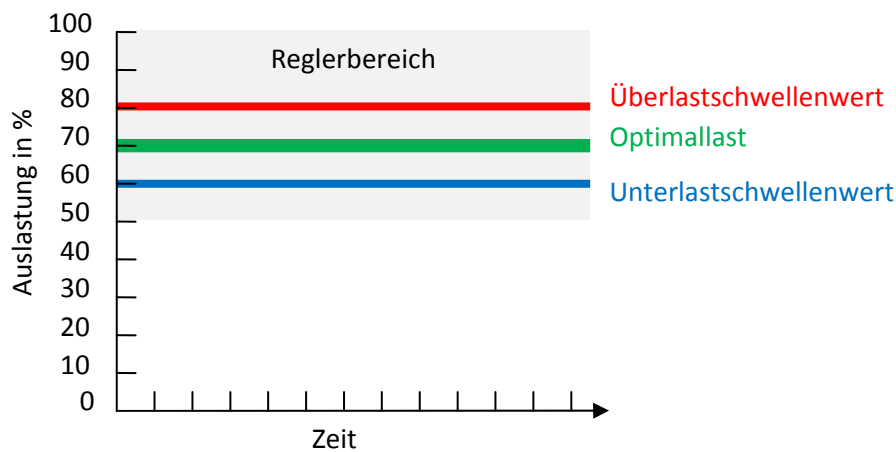


Abbildung 5.4: Übersicht über den Lastbereich beim Regler-basierten Ansatz

### 5.4.2.3 Regelalgorithmus

Der Regelalgorithmus des Regler-basierten Ansatzes regelt im Vergleich zum TCP-basierten Ansatz nicht mit konstanten Sprüngen, sondern berechnet den optimalen Sprung mit Hilfe einer Formel. Er regelt zudem immer auf einen Optimalwert hin. Die Sprünge, die verwendet werden, werden immer so berechnet, dass die Last nach dem Adaptionsschritt theoretisch exakt auf dem Optimallastwert zu liegen kommt. Deshalb ist hier auch kein zusätzlicher Status wie beim TCP-basierten Ansatz nötig, da sich der Sprung entsprechend an den aktuellen Lastzustand anpasst.

Zu Beginn der Regelung wird auch hier mit einer Initialen Adaption begonnen, die so lange greift, bis die Last innerhalb des Reglerbereiches zu liegen kommt. Ab dann wird mit dem eigentlichen Regelvorgang begonnen. Hierbei gibt es eine Unterlastbehandlung, die den TDF entsprechend verringert und eine Überlastbehandlung die den TDF entsprechend erhöht. Sollte man einen Filter gewählt haben, der einen breiten Optimallastbereich definiert,

so kommt zudem, wie auch beim TCP-basierten Ansatz, ein Feintuning im Optimallastbereich zum Einsatz. Dieses Feintuning erwirkt eine Ausführung des Experiments bei möglichst optimaler Last und beschleunigt somit die Ausführungszeit. Es werden zudem die Sprünge bei Unterlast verringert, so dass eine schwächere Behandlung der Unterlast erfolgt. Die Algorithmen und die konkrete Umsetzung des Regler-basierten Ansatzes kann im Kapitel 6 „Implementierung“ nachvollzogen werden.

### 5.4.2.4 Timer

Die Anpassung wird grundsetzlich durch eine Signalisierung des Global-State-Moduls angestoßen. Sollte allerdings ein Adaptionsschritt nicht genügen um das gewünschte Ziel zu erreichen, so müssen ein oder mehrere Adaptionsschritte folgen. Da speziell zu Beginn in der Startsequenz keine großen Laständerungen zu erwarten sind, ist es unabdingbar einen Timer zu definieren, der eine periodische Anpassung, auch ohne Interrupt vom Global-State-Modul, anstößt.

Wie auch beim TCP-basierten Ansatz hängt dieser Timer stark vom Triggertimer ab (Feedback der vorgehenden Adaption). Somit muss dieser Timer ebenfalls an den aktuellsten Zustand angepasst werden. Dies kann durch die folgende Formel geschehen. Es muss ebenfalls eine obere Grenze für den Timer definiert werden.

$$\text{aktuellerTimer} = \text{MIN}(2^{\text{aktuellerTDF}/100} * \text{MinTimer}, \text{MaxTimer})$$

# Implementierung

---

In diesem Kapitel werden nun die vorgestellten Konzepte implementiert und in das bestehende System (TVEE) integriert.

## 6.1 Allgemeines

Der hier entwickelte TDF-Adapter soll in den bestehenden Prototypen der TVEE an der Universität Stuttgart integriert werden. Dieser Prototyp läuft auf einem Linux-Cluster. Somit wird der TDF-Adapter auch für Linux implementiert. Die einzelnen Komponenten werden jeweils als Linux-Kernel-Modul realisiert. Dies bedeutet, dass als Programmiersprache C zum Einsatz kommt. Dies birgt zugleich einige Komplikationen, da für die Adaption auch kompliziertere Rechenoperationen, wie zum Beispiel der Logarithmus, benötigt werden. Diese müssen in den Linux-Kernel-Modulen durch komplizierte Hilfsoperationen umgesetzt werden und können nicht, wie bei Hochsprachen wie Java oder C#, direkt aus einer Bibliothek angesprochen werden. Weiterhin muss bei einfachen Rechenoperation, wie der Division, darauf geachtet werden, dass eine sinnvolle Reihenfolge eingehalten wird. Prinzipiell gilt, dass man so spät wie möglich dividieren soll, da es in den Modulen auch keine Gleitkommazahlen gibt, sondern nur ganzzahlige Integerwerte. Somit ginge durch frühes dividieren die Genauigkeit der Ergebnisse verloren. Die hier vorgestellten Algorithmen sind allerdings in vereinfachter Darstellung abgebildet. Der vorgestellte Code beinhaltet auch Rechenoperationen, die in der realen Implementierung deutlich komplizierter umgesetzt werden müssen.

Die Implementierung ist zudem gründlich zu testen, da die Module im Kernspace laufen und somit bei einem Fehler im Code den kompletten Rechner zum Absturz bringen können. Des weiteren laufen diese Module in der domo von XEN, so dass keinerlei Änderungen am eigentlichen Linux-Betriebssystem nötig sind. Da es sich um Kernel-Module handelt können sie auch zur Laufzeit des Rechners geladen, entladen und ausgetauscht werden. Bei der Implementierung wird zudem darauf geachtet, dass alle verwendeten Parameter über Kommandozeilenbefehle konfigurierbar sind. Sie können dann über das Proc-Dateisystem des Rechners ausgelesen oder verändert werden.

### 6.2 Implementierung des Relevanz-Moduls

Die Implementierung des Relevanzmoduls beinhaltet die Umsetzung der Triggerlogik und der Filterlogik.

#### 6.2.1 Trigger

##### 6.2.1.1 Timergesteuerter Trigger

Der timergesteuerte Trigger ermittelt nach Ablauf eines Trigger-Timers die im vergangenen Intervall anliegende durchschnittliche Last des aktuellen Knotens. Hierbei werden die Lastwerte aus der History verwendet. Dieser Durchschnittswert wird nun an den Filter weitergegeben, der die Relevanz entscheidet. Je kleiner das Timerintervall des Triggers gewählt wird, um so exakter kann der Global-State ermittelt werden und um so schneller werden Lastschwankungen erkannt, aber auch um so mehr Pakete können zum zentralen Koordinator gesendet werden. Dieser Trigger skaliert bei angepassten Intervallgrößen für nahezu jede Systemgröße. Zudem bestimmt die Intervallgröße die maximale Anzahl an Lastnachrichten die zum Koordinator gesendet werden können. Allerdings geht eine Erhöhung der Intervallgröße auf Kosten der Genauigkeit. Hierbei kann es durch fehlende Lastsignalisierungen zu einer Überlast auf dem lokalen Knoten kommen, die nicht erkannt und somit nicht umgehend behandelt werden kann. Der Wert  $X$  bestimmt hier die Intervallgröße in Millisekunden.

Listing 6.1 zeigt den Algorithmus des timergesteuerten Triggers.

---

**Listing 6.1** Algorithmus für den timergesteuerten Trigger

---

```
1 void Time_Based_Trigger()
2 {
3     while (true) {
4         CurrentLoad = History.Get_Average_Load(X);
5
6         SendLoadToFilter(CurrentLoad);
7
8         Sleep(X);
9     }
10 }
```

---

Durch die Berücksichtigung des aktuellen Zustandes kann der Triggertimer variiert werden. Im Kapitel 5.2 „Entwurf des Relevanz-Moduls“ wurde die Formel für diese Aufgabe bereits vorgestellt.

Der abgeänderte Algorithmus für den timergesteuerten Trigger wird in Listing 6.2 dargestellt.

---

**Listing 6.2** Algorithmus für den timergesteuerten Trigger mit Berücksichtigung des aktuellen Zustandes

---

```
1 void Time_Based_Trigger()
2 {
3     while (true) {
4         //Calc Timerintervall
5         if (CurrentTDF > 100) {
6             CurrentTimer = 2**((CurrentTDF / 100) * MinTimer);
7
8             if (CurrentTimer > MaxTimer) {
9                 CurrentTimer = MaxTimer);
10            }
11        }
12
13        //Check Load
14        CurrentLoad = History.Get_Average_Load(CurrentTimer);
15
16        SendLoadToFilter(CurrentLoad);
17
18        Sleep(CurrentTimer);
19    }
20 }
```

---

### 6.2.1.2 Eventgesteuerter Trigger

Der eventgesteuerte Trigger bestimmt anhand des Auftretens von Events, zu welchem Zeitpunkt ein Lastwert zum Filter gesendet wird. Für jedes zu übermittelnde Event gib es einen Counter. Sobald ein Event eine gewisse Zeit lang aufgetreten ist, wird ein charakteristischer Lastwert ermittelt und an den Filter weitergereicht. Die kleinste Zeiteinheit beträgt eine Millisekunde. Jede Millisekunde wird die aktuelle Last auf die zugehörigen Events geprüft. Sobald ein Event auftritt wird der entsprechende Counter erhöht. Sollte das Event in einer Millisekunde nicht aufgetreten sein, so wird der Counter wieder auf Null gesetzt. Wenn ein Event eine definierte Dauer lang (*MaxXCounter*) hintereinander aufgetreten ist, wird die in diesem Intervall durchschnittlich angelegene Last zur Bewertung an den Filter weitergereicht.

Listing 6.3 zeigt beispielhaft den Algorithmus des eventgesteuerten Triggers für die Events OVERLOADED, POSSIBLE\_OVERLOAD und UNDERLOADED.

---

### Listing 6.3 Algorithmus für den eventgesteuerten Trigger

---

```
1 void Event_Based_Trigger()
2 {
3     OverloadCounter = 0;
4     PossibleOverloadCounter = 0;
5     UnderloadCounter = 0;
6
7     while (true) {
8         //Catch Overload
9         if (History.GetCurrent_load() >= OVERLOAD_WARNING_THRESHOLD) {
10            OverloadCounter += 1;
11        } else {
12            OverloadCounter = 0;
13        }
14
15        //Catch Possible_Overload
16        if (History.GetCurrent_load() >= POSSIBLE_OVERLOAD_WARNING_THRESHOLD) {
17            PossibleOverloadCounter += 1;
18        } else {
19            PossibleOverloadCounter = 0;
20        }
21
22        //Catch Underload
23        if (History.GetCurrent_load() <= UNDERLOAD_THRESHOLD) {
24            UnderloadCounter += 1;
25        } else {
26            UnderloadCounter = 0;
27        }
28
29        //Send Load
30        if (OverloadCounter >= MaxOverloadCounter) {
31            OverloadCounter = 0;
32            PossibleOverloadCounter = 0;
33            UnderloadCounter = 0;
34
35            SendLoadToFilter(History.Get_Average_Load(MaxOverloadCounter));
36        }
37        if (PossibleOverloadCounter >= MaxPossibleOverloadCounter) {
38            OverloadCounter = 0;
39            PossibleOverloadCounter = 0;
40            UnderloadCounter = 0;
41
42            SendLoadToFilter(History.Get_Average_Load(MaxPossibleOverloadCounter));
43        }
44        if (UnderloadCounter >= MaxUnderloadCounter) {
45            OverloadCounter = 0;
46            PossibleOverloadCounter = 0;
47            UnderloadCounter = 0;
48
49            SendLoadToFilter(History.Get_Average_Load(MaxUnderloadCounter));
50        }
51
52        Sleep(1);
53    }
54 }
```

---

Die Berücksichtigung des aktuellen Zustandes spielt auch hier eine wichtige Rolle. Die Formel für die Anpassung der Länge der Events wurde bereits im Kapitel 5.2 „Entwurf des Relevanz-Moduls“ vorgestellt. Dazu werden jeweils *MinXCounter* und *MaxXCounter* eingeführt.

Den entsprechend abgewandelten Algorithmus zeigt Listing 6.4 (nur für den Status Überlast).

---

**Listing 6.4** Algorithmus für den eventgesteuerten Trigger mit Berücksichtigung des aktuellen Zustandes

---

```
1 void Event_Based_Trigger()
2 {
3     OverloadCounter = 0;
4
5     while (true) {
6         //Calc Counter
7         if (CurrentTDF > 100) {
8             CurrentMaxOverloadCounter = 2**((CurrentTDF / 100) *
9                 MinOverloadCounter;
10
11             if (CurrentMaxOverloadCounter > MaxOverloadCounter) {
12                 CurrentMaxOverloadCounter = MaxOverloadCounter;
13             }
14
15
16             //Catch Overload
17             if (History.Get_Current_load() >= OVERLOAD_WARNING_THRESHOLD) {
18                 OverloadCounter += 1;
19             } else {
20                 OverloadCounter = 0;
21             }
22
23             //Send Load
24             if (OverloadCounter >= CurrentMaxOverloadCounter) {
25                 OverloadCounter = 0;
26
27                 SendLoadToFilter(History.Get_Average_Load(CurrentMaxOverloadCounter));
28             }
29
30             Sleep(1);
31         }
32 }
```

---

### 6.2.2 Filter

Prinzipiell besteht die Möglichkeit den Filter komplett wegzulassen und jeden getriggerten Lastwert direkt zum Koordinator zu senden. Allerdings werden dabei einige unnötige Lastwerte zum Koordinator gesendet, die völlig irrelevant für die Adaption des TDF sind. Deshalb ist es sinnvoll einen der hier vorgestellten Filter zu verwenden.

#### 6.2.2.1 Differentieller Filter

Beim Differentiellen Filter wird die Last des Knotens nur dann an den Zentralen Koordinator weitergeleitet, wenn sie sich im Vergleich zum zuletzt gesendeten Wert um einen gewissen Betrag verändert hat. Hierbei kann zwischen einer Veränderung der Last nach oben und unten unterschieden werden, so dass eine Erhöhung der Last eher kommuniziert wird als eine Verringerung. *DOWNWARD\_VARIANCE* und *UPWARD\_VARIANCE* stellen hierbei den Betrag der Abweichung der Last nach unten oder oben bei der die Lastveränderung gesendet wird dar. Je größer man die Beträge der Abweichungen wählt um so weniger Lastnachrichten werden kommuniziert. Jedoch verliert man damit auch die Möglichkeit schnell auf eine Lastveränderung reagieren zu können.

Dieses Konzept skaliert wunderbar für alle Systemgrößen. Mit dem bereits vorgestellten timergesteuerten Trigger kann die maximale Anzahl der Lastpakete, die in einer Sekunde gesendet werden, bestimmt und durch die Beträge der Abweichungen kann der Filter für die Relevanz bestimmt werden. Somit hat man extrem vielseitige Möglichkeiten die Genauigkeit und den Aufwand der Signalisierung zu bestimmen. Selbiges kann beim eventgesteuerten Trigger mit der maximalen Länge des Auftretens eines Events realisiert werden.

Listing 6.5 zeigt den Algorithmus für den Differentiellen Filter.

---

#### Listing 6.5 Algorithmus für den Differentiellen Filter

---

```
1 int LastSentLoad = 0;
2
3 void Differential_Filter(int TriggeredLoad)
4 {
5     if (TriggeredLoad < (LastSentLoad - DOWNWARD_VARIANCE)) {
6         SendLoadToCoordinator(TriggeredLoad);
7         LastSentLoad = TriggeredLoad;
8     } else if (TriggeredLoad > (LastSentLoad + UPWARD_VARIANCE)) {
9         SendLoadToCoordinator(TriggeredLoad);
10        LastSentLoad = TriggeredLoad;
11    }
12 }
```

---

#### 6.2.2.2 Schwellenwert-basierter Filter

Bei dieser Art von Filter werden Lastbereiche definiert, die durch Schwellenwerte voneinander getrennt werden. Sobald die Last in einen anderen Bereich wechselt, wird der neue

Bereich/Zustand und die neue Last signalisiert. Dies ist relativ ähnlich dem des Differentiellen Filters. Jedoch sind, im Vergleich zum vorher erwähnten Fall, die Signalisierungsbereiche fest definiert und ändern sich nicht wie beim Differentiellen Filter mit jeder Signalisierung. Hier wird auch das erste Mal der Lastzustand ermittelt und kommuniziert. Prinzipiell können beliebig viele Lastbereiche definiert werden. In diesem Beispiel werden vier Bereiche definiert (UNDERLOADED, OPTIMAL, POSSIBLE\_OVERLOAD, OVERLOADED). Diese Bereiche werden von drei Schwellenwerten getrennt (UNDERLOAD\_THRESHOLD, POSSIBLE\_OVERLOAD\_WARNING\_THRESHOLD, OVERLOAD\_WARNING\_THRESHOLD). Da diese Relevanzbestimmung ebenfalls gute Filtereigenschaften besitzt skaliert dieses System, wie das vorige, sehr gut für alle Systemgrößen.

Der Algorithmus für diesen Filter ist in Listing 6.6 dargestellt.

---

**Listing 6.6** Algorithmus für den Schwellenwert-basierten Filter

---

```
1 LastState = UNDEFINED;
2
3 void Threshold_Based_Filter(int TriggeredLoad)
4 {
5     if (TriggeredLoad < UNDERLOAD_THRESHOLD) {
6         CurrentState = UNDERLOADED;
7     } else if (TriggeredLoad < POSSIBLE_OVERLOAD_WARNING_THRESHOLD) {
8         CurrentState = OPTIMAL;
9     } else if (TriggeredLoad < OVERLOAD_WARNING_THRESHOLD) {
10        CurrentState = POSSIBLE_OVERLOAD;
11    } else if (TriggeredLoad >= OVERLOAD_WARNING_THRESHOLD) {
12        CurrentState = OVERLOADED;
13    }
14
15    if (CurrentState != LastState) {
16        LastState = CurrentState;
17        SendLoadToCoordinator(TriggeredLoad, CurrentState);
18    }
19 }
```

---

## 6.3 Implementierung des Global-State-Moduls

Die Implementierung des Global-State-Moduls realisiert die Bildung des Global-State. Hierfür gibt es eine Methode, die die empfangene Last der Knoten aktualisiert und eine Methode, die periodisch den Global-State bildet.

### 6.3.1 Aktualisierung der Lastwerte

Diese Methode empfängt von jedem Knoten den aktuellen Lastwert und speichert ihn in einer Liste am entsprechenden Platz des Knotens. Sie sorgt dafür, dass die Einträge der Knoten beim Eintreffen einer neuen Nachricht aktuell gehalten sind.

Listing 6.7 zeigt den Algorithmus dieser Methode.

---

### Listing 6.7 Algorithmus für die Aktualisierung der Knotenlast im Global-State-Modul

---

```
1 void Set_Load(int NodeID, int NodeLoad, int NodeState)
2 {
3     GlobalLoadList[NodeID] = NodeLoad;
4     GlobalStateList[NodeID] = NodeState;
5 }
```

---

### 6.3.2 Berechnung des Global-State

Der Global-State wird periodisch aus allen in der Liste verfügbaren Lastinformationen der Knoten gebildet. Dabei wird einfach der höchste Lastwert gesucht und bei einer Änderung an das Adaptionmodul weitergereicht.

Listing 6.8 zeigt den Code für diese Methode.

---

### Listing 6.8 Algorithmus für die Bildung des Global-State im Global-State-Modul

---

```
1 LastMaxLoad = 0;
2
3 void Get_Global_State()
4 {
5     while(true) {
6         MaxLoad = 0;
7         CurrentState = UNDEF;
8
9         for (i = 0; i < GlobalLoadList.Count; i++) {
10            if (GlobalLoadList[i] > MaxLoad) {
11                MaxLoad = GlobalLoadList[i];
12                CurrentState = GlobalStateList[i];
13            }
14        }
15
16        if (MaxLoad != LastMaxLoad) {
17            Adapt_TDF(MaxLoad, CurrentState);
18            LastMaxLoad = MaxLoad;
19        }
20
21        Sleep(1);
22    }
23 }
```

---

## 6.4 Implementierung des Adaption-Moduls

In diesem Abschnitt wird die Implementierung der Adaption-Konzepte vorgestellt.

### 6.4.1 TCP-basierter Ansatz

Wie bereits beim Entwurf des Adaption-Moduls erläutert wurde, reagiert dieser Ansatz auf eine Lastsignalisierung mittels Lastzuständen. Daher muss für jeden Status eine entsprechende Methode zur Behandlung definiert werden. Zudem müssen einige weitere Parameter für eine erfolgreiche Adaption definiert werden.

#### 6.4.1.1 Timer

Da es möglich sein muss eine Behandlung auch ohne Statusänderung fortzuführen, muss der Adaption-Timer definiert werden. Hierzu wird der Parameter *Timer* definiert. Da der Adaptionstimer durch die bereits vorgestellte Formel an den aktuellen Zustand angepasst werden kann, muss auch ein entsprechender oberer Grenzwert für den Adaption-Timer angegeben werden. Dies ist der Parameter *MaxTimer*.

#### 6.4.1.2 TDF Grenzwerte

**Oberer Grenzwert** Um nicht in die Gefahr einer zu langsamen Ausführung des Experiments zu gelangen und damit die Steuerbarkeit des Experiments zu verlieren, muss der TDF nach oben begrenzt werden. Dieser obere Grenzwert wird zudem in der Initialen Adaption als Einstiegswert verwendet. Der entsprechende Parameter heißt *MaxTDF*.

**Unterer Grenzwert** Da das Experiment auch nicht zu schnell ablaufen darf, da sonst die Steuerbarkeit verloren geht, muss der minimale TDF bestimmt werden. Der Parameter hierfür ist der *MinTDF*.

#### 6.4.1.3 Anpassungsschritte

**TDF-Sprung bei Unterlast** Um eine Adaption des TDF bei Unterlast vornehmen zu können muss der Anpassungsschritt nach unten (*UnderloadTDFStep*) definiert werden. Dieser Schritt sollte nicht zu groß sein, da durch einen großen Wert die Adaptiongranularität vergrößert wird und zudem die Anpassung eventuell zu hart ist, so dass das System den Status *OPTIMAL* überspringt. Allerdings darf er auch nicht zu niedrig gewählt werden, da sonst bei einer großen Änderung zu viele Adaptionsschritte nötig sind.

**TDF-Sprung bei potentieller Überlast** Sollte das System in den Bereich der potentiellen Überlast geraten muss der TDF so korrigiert werden, dass die Last nach Möglichkeit nach einem Anpassungsschritt wieder im Bereich des Optimums liegt. Abhängig von der Größe des Bereiches und der Granularität der TDF-Schritte muss dieser Schritt (*PossibleOverloadTDFStep*) entsprechend dimensioniert werden.

**Maximaler TDF-Sprung (bei Überlast)** Um der Gefahr des Auseinanderlaufens der einzelnen Maschinen durch zu hohe Sprünge entgegenzuwirken, muss man den maximalen TDF Sprung (*MaxTDFStep*) definieren. Er sollte aber so dimensioniert sein, dass auf eine Überlast noch stark genug reagiert werden kann.

### 6.4.1.4 Startsequenz

Da zu Beginn eines Experiments nicht klar ist, welcher TDF gewählt werden sollte um im Status *OPTIMAL* zu sein, wird mit dem höchsten TDF (*MaxTDF*) begonnen. Typischerweise befindet sich die Last nun im Unterlastbereich. Man kann hier auch keinen, wie bei TCP üblichen Schwellenwert angeben bis zu dem exponentiell angestiegen wird, da dieser Wert nicht im vorhinein bestimmt werden kann und damit evtl. durch einen zu schnellen Anstieg der Last das System in Überlast geraten würde. Deshalb wird hier der TDF im Timerrhythmus um den Betrag (*UnderloadTDFStep*) verringert, also die Last theoretisch linear erhöht. Dies wird so lange fortgeführt bis die Last den Optimalbereich oder der TDF den Wert *MinTDF* erreicht. Befindet sich nun die Last im Optimalbereich oder wird der Grenzwert des TDF erreicht endet diese Adaption.

Das Listing 6.9 zeigt den Algorithmus für diese Initialisierungsphase.

---

#### Listing 6.9 Initialisierungsalgorithmus beim TCP-basierten Ansatz

---

```
1 void Init()
2 {
3     SetTDF(MaxTDF);
4     Systemstate = UNDERLOAD;
5
6     StartUnderloadHandling();
7 }
```

---

### 6.4.1.5 Reaktion bei Unterlast

Befindet sich das System im Zustand *UNDERLOADED*, so wird wie bei der Startsequenz verfahren. Zum Erhöhen der Last wird wiederum periodisch der TDF um den *UnderloadTDFStep* verringert bis sich das System wieder im Optimallastbereich befindet oder der Wert *MinTDF* erreicht wird. Sobald dieser Optimallastbereich oder der Grenzwert erreicht wird, wird diese Adaption beendet. Der Timer hierfür kann durch die Berücksichtigung des aktuellen Zustands durch die oben angegebene Formel variiert werden. Zudem wird

der Timer verdoppelt, um die Unterlastbehandlung noch langsamer voranzutreiben. Listing 6.10 zeigt den Algorithmus für die Unterlastbehandlung.

---

**Listing 6.10** Algorithmus zur Unterlastbehandlung beim TCP-basierten Ansatz

---

```
1 void StartUnderloadHandling()
2 {
3     while (Systemstate == UNDERLOADED) {
4         newTDF = TDF - UnderloadTDFStep;
5
6         if (newTDF > MinTDF) {
7             SetTDF(newTDF);
8         }
9         else {
10            SetTDF(MinTDF);
11            break;
12        }
13
14        Sleep(X * 2);
15    }
16 }
```

---

Anmerkung: Der Systemstatus kann jederzeit durch einen Interrupt des Global-State-Moduls geändert werden. Damit kann ein anderer Algorithmus angestoßen werden und der aktuelle Algorithmus terminiert nach Durchlaufen der Schleife.

#### 6.4.1.6 Feintuning bei Optimallast

Da die Optimallast ein Bereich und kein diskreter Wert ist kann es sein, dass die Last durch die Über- oder Unterlastbehandlung an die untere Grenze des Optimallastbereiches gebracht wird. Sollte nun die Last konstant bleiben und keine weitere Adaption angestoßen werden, verschenkt man damit Zeit, da die Last durchaus im oberen Bereich des Optimallastbereiches sein könnte. Somit benötigt man jetzt einen Algorithmus der den TDF in gewissen Abständen (beispielsweise alle 500 ms) um einen minimalen Betrag erhöht, so dass gewährleistet ist, dass die Last möglichst im oberen Bereich des Optimallastbereiches verläuft.

Listing 6.11 zeigt dieses Feintuning.

---

**Listing 6.11** Algorithmus zum Feintuning im Optimallastbereiches beim TCP-basierten Ansatz

---

```
1 void StartFinetuning()
2 {
3     while (Systemstate == OPTIMAL) {
4         newTDF = TDF - 1;
5
6         if (newTDF > MinTDF) {
7             SetTDF(newTDF);
8         } else {
9             SetTDF(MinTDF);
10            break;
11        }
12
13        Sleep(500);
14    }
15 }
```

---

Anmerkung: Der Systemstatus kann jederzeit durch einen Interrupt des Global-State-Moduls geändert werden. Damit kann ein anderer Algorithmus angestoßen werden und der aktuelle Algorithmus terminiert nach Durchlaufen der Schleife.

### 6.4.1.7 Reaktion bei potentieller Überlast

Gerät das System in den Bereich der potentiellen Überlast so wird der TDF um den *PossibleOverloadTDFStep* erhöht der eine Änderung der Last um die gewünschte Prozentzahl erbringt, so dass die Last wieder in den Bereich des Optimums zurückkehrt. Sollte die Adaption nicht erfolgreich verlaufen, also die Last immer noch zu hoch sein, so wird nach Ablauf des Timers ein erneuter Adaptionsschritt durchgeführt. Der Timer kann hierbei durch die Berücksichtigung des aktuellen Zustands verändert werden. Dies wird so lange durchgeführt bis sich das System wieder mindestens im Bereich der Optimallast befindet oder der Wert *MaxTDF* erreicht wird. Gerät dabei das System in den Unterlastbereich so wird entsprechend der Unterlastbehandlung reagiert. Pendelt sich das System wieder im Optimalbereich ein, so ist diese Adaption damit beendet.

Der Algorithmus zur potentiellen Überlastbehandlung ist in Listing 6.12 dargestellt.

**Listing 6.12** Algorithmus zur potentiellen Überlastbehandlung beim TCP-basierten Ansatz

---

```

1 void StartPossibleOverloadHandling()
2 {
3     while (Systemstate == POSSIBLE_OVERLOAD) {
4         newTDF = TDF + PossibleOverloadTDFStep;
5
6         if (newTDF < MaxTDF) {
7             SetTDF(newTDF);
8         } else {
9             SetTDF(MaxTDF);
10            break;
11        }
12
13        Sleep(X);
14    }
15 }

```

---

Anmerkung: Der Systemstatus kann jederzeit durch einen Interrupt des Global-State-Moduls geändert werden. Damit kann ein anderer Algorithmus angestoßen werden und der aktuelle Algorithmus terminiert nach Durchlaufen der Schleife.

#### 6.4.1.8 Reaktion bei Überlast

Wenn die Last des Systems in den Überlastbereich gerät wird der TDF umgehend um den Betrag des *MaxTDFStep* erhöht. Genügt dies nicht, um die Last wieder in den Optimalbereich zu befördern, wird auch dieser Schritt wiederholt. Auch hier kann der Timer durch die Berücksichtigung des aktuellen Zustandes variiert werden. Dies wird fortgeführt, bis das System wieder im Optimalbereich oder im Bereich potentieller Überlast ankommt oder der Wert *MaxTDF* erreicht wird. Sollte dabei eine Unterlast entstehen so wird diese entsprechend behandelt.

Listing 6.13 zeigt diesen Algorithmus.

**Listing 6.13** Algorithmus zur Überlastbehandlung beim TCP-basierten Ansatz

---

```

1 void StartOverloadHandling()
2 {
3     while (Systemstate == OVERLOADED) {
4         newTDF = TDF + MaxTDFStep;
5
6         if (newTDF < MaxTDF) {
7             SetTDF(newTDF);
8         } else {
9             SetTDF(MaxTDF);
10        }
11
12        Sleep(X);
13    }
14 }

```

---

Anmerkung: Der Systemstatus kann jederzeit durch einen Interrupt des Global-State-Moduls geändert werden. Damit kann ein anderer Algorithmus angestoßen werden und der aktuelle Algorithmus terminiert nach Durchlaufen der Schleife.

### 6.4.1.9 Basisalgorithmus

Listing 6.14 zeigt den Basisalgorithmus des TCP-basierten Ansatzes.

---

#### Listing 6.14 Basisalgorithmus des TCP-basierten Ansatzes

---

```
1 void AdaptTDF(newSystemstate)
2 {
3     Systemstate = newSystemstate;
4
5     if (Systemstate == UNDERLOADED) {
6         StartUnderloadHandling();
7     } else if (Systemstate == POSSIBLE_OVERLOAD) {
8         StartPossibleOverloadHandling();
9     } else if (Systemstate == OVERLOADED) {
10        StartOverloadHandling();
11    } else if (Systemstate == OPTIMAL)    {
12        StartFinetuning();
13    }
14 }
```

---

Jedes Mal wenn das Global-State-Modul einen neuen Statusbericht an das Adaption-Modul sendet wird die Methode *AdaptTDF()* aufgerufen und ihr der neue Systemstatus übergeben. Dort wird die globale Variable entsprechend neu belegt und der entsprechende Algorithmus zur Reaktion auf den neuen Status gestartet. Dieser läuft so lange bis entweder ein Grenzwert für den TDF erreicht wird oder das Ziel der Behandlung erreicht wurde und sich der Systemstatus dadurch geändert hat. Dies wird durch eine neue Nachricht des Global-State-Moduls mitgeteilt. Die Schleife der einzelnen Algorithmen wird daraufhin verlassen und die Methode beendet.

### 6.4.1.10 Diskussion

Mit diesem TCP-ähnlichen Ansatz haben wir ein System geschaffen, das sich zu Beginn eines Experiments mittels eines timergesteuerten Zyklus langsam an die Optimallast herantastet. Ist diese Optimallast erreicht beginnt die Korrekturphase. Dabei wird abhängig vom aktuellen Zustand reagiert. Bei drohender Überlast wird so stark reagiert, dass im Mittel die Last wieder in den Optimalbereich abfällt. Bei Überlast wird der TDF um den maximal möglichen Schritt erhöht. Somit wird der TDF bei zu starker Last relativ hart nach oben korrigiert. Da Unterlast nicht kritisch sondern nur zeitverzögernd auf das Experiment einwirkt, muss nicht so drastisch adaptiert werden. In diesem Fall wird der TDF analog zu TCP linear um kleine Schritte verringert. Auch der Anforderung an eine Ausführung des Experiments bei möglichst hoher Last wird dadurch Rechnung getragen, dass selbst

im Optimallastbereich ein Feintuning betrieben wird, welches die Last möglichst in den oberen Bereich des Optimallastbereiches befördert.

Durch die differenzierte Adaption nach oben und unten wird auch das Problem des Aufschaukelns und Oszillierens weitgehend gebremst. Da die Adaption nach oben sehr zügig geht, die Adaption nach unten aber sehr gemächlich, kann sich das System nur minimal selbst zum Schwingen bringen.

### 6.4.2 Regler-basierter Ansatz

Im Gegensatz zum TCP-basierten Adaption-Ansatz wird hier anhand der genauen Lastwerte adaptiert. Deshalb müssen erst einige Parameter für eine funktionstüchtige Adaption definiert werden.

#### 6.4.2.1 Timer

Da es möglich sein muss, eine Behandlung auch ohne erneute Signalisierung des Global-State-Moduls fortzuführen muss der Adaption-Timer definiert werden. Hierzu wird der Parameter *Timer* definiert. Da der Adaptionstimer durch die bereits vorgestellte Formel an den aktuellen Zustand angepasst werden kann, muss auch ein entsprechender oberer Grenzwert für den Adaption-Timer angegeben werden. Dies ist der Parameter *MaxTimer*.

#### 6.4.2.2 Optimallast

Da der Regler immer auf eine Abweichung des Istzustandes zu einem Sollzustand reagiert muss bei diesem Ansatz ein Sollwert/Optimalwert (*OptimalLoad*) der Last definiert werden.

#### 6.4.2.3 TDF Grenzwerte

**Oberer Grenzwert** Um nicht in die Gefahr einer zu langsamen Ausführung des Experiments zu gelangen und damit die Steuerbarkeit des Experiments zu verlieren, muss der TDF nach oben begrenzt werden. Dieser obere Grenzwert wird zudem in der Initialen Adaption als Einstiegswert verwendet. Der entsprechende Parameter heißt *MaxTDF*.

**Unterer Grenzwert** Da das Experiment auch nicht zu schnell ablaufen darf, da sonst ebenfalls die Steuerbarkeit verloren geht, muss auch der minimale TDF bestimmt werden. Der Parameter hierfür ist der *MinTDF*.

### 6.4.2.4 Anpassungsschritte

**Maximaler TDF-Sprung** Um der Gefahr des Auseinanderlaufens der einzelnen Maschinen durch zu hohe Sprünge entgegenzuwirken muss man den maximalen TDF Sprung (*MaxTDFStep*) definieren.

**Initialer TDF-Sprung** Ebenso muss ein Parameter für den Sprung bei der Initialen Adaption definiert werden, der für die Regelung ausserhalb des Reglerbereiches herangezogen wird. Der Parameter hierfür heißt *InitialTDFStep*.

### 6.4.2.5 Startsequenz

Zu Beginn eines Experiments ist nicht klar, welcher TDF gewählt werden sollte um die Last in den Optimallastbereich zu befördern. Deshalb wird mit dem höchsten TDF (*MaxTDF*) begonnen. Typischerweise befindet sich das System nun in Unterlast und außerhalb des Reglerbereiches. Nun wird analog des TCP-basierten Ansatzes verfahren. Wie dort beschrieben, wird der TDF zyklisch um den Betrag *InitialTDFStep* verringert, also die Last theoretisch linear erhöht. Dies wird so lange fortgeführt bis die Last den Reglerbereich oder der TDF den Wert **MinTDF** erreicht. Befindet sich nun die Last im Reglerbereich beginnt die Regelung mit Hilfe des dort definierten Regelalgorithmus und diese Initiale Adaption endet. Wird der Reglerbereich nicht erreicht da der TDF bereits den Wert **MinTDF** erreicht hat endet diese Adaption ebenfalls.

Listing 6.15 zeigt diesen initialen Algorithmus.

---

#### Listing 6.15 Initialisierungsalgorithmus beim Regler-basierten Ansatz

---

```
1 void Init()
2 {
3     SetTDF(MaxTDF);
4     Systemstate = INITIAL;
5
6     while (Systemstate == INITIAL) {
7         newTDF = TDF - TDFStep;
8
9         if (newTDF > MinTDF) {
10            SetTDF(newTDF);
11        } else {
12            SetTDF(MinTDF);
13            break;
14        }
15
16        Sleep(X);
17    }
18 }
```

---

Anmerkung: Der Systemstatus kann jederzeit durch einen Interrupt des Global-State-Moduls geändert werden. Damit kann ein anderer Algorithmus angestoßen werden und der aktuelle Algorithmus terminiert.

#### 6.4.2.6 Reaktion bei Unterlast

Sobald das Global-State-Modul Unterlast signalisiert beginnt der Regelvorgang durch den Regelalgorithmus. Hierbei wird der Quotient aus dem optimalen Lastwert und dem aktuellen Lastwert ermittelt. Dieser Quotient bedeutet, dass die aktuelle Virtuelle Zeit mit diesem Quotienten multipliziert werden müsste um die Last auf den Optimalwert zu bringen. Aus der anfangs vorgestellten Formel zur Berechnung der Virtuellen Zeit anhand des TDFs:

$$T'_{virtuell} = T'_{real} \times 2^{(TDF/100)}$$

lässt sich für diesen Fall die Formel

$$TDFStep = \ln\left(\frac{\text{optimalerLastwert}}{\text{aktuellerLastwert}}\right) * 100$$

herleiten. Das Ergebnis ist der TDF-Schritt, um den der aktuelle TDF verringert werden muss, damit die Last auf den Optimalwert befördert wird. Da die Reaktion auf Unterlast allerdings langsamer als die Reaktion auf Überlast erfolgen soll wird der TDF-Schritt noch halbiert. Ist dieser Schritt nun kleiner oder gleich dem Wert des **MaxTDFStep / 2**, wird der neue TDF eingestellt. Ist der Sprung zu groß, wird der alte TDF nur um den Betrag des **MaxTDFStep / 2** verringert. Dieses Vorgehen wird so lange wiederholt bis die Last wieder dem Optimalwert angenähert ist. Sollte zwischenzeitlich der Wert des TDF auf den **MinTDF** abgefallen sein wird diese Adaption hierdurch beendet. Der Timer X kann durch die Berücksichtigung des aktuellen Zustands variiert werden. Zudem wird der Timer verdoppelt, so dass die Reaktion auf Unterlast noch langsamer erfolgt.

Der komplette Algorithmus zur Unterlastbehandlung beim Regler-basierten Ansatz kann in Listing 6.16 nachvollzogen werden.

---

**Listing 6.16** Algorithmus zur Unterlastbehandlung beim Regler-basierten Ansatz

---

```
1 void StartUnderloadHandling()
2 {
3     while (Systemstate == UNDERLOADED) {
4         LocalTDFStep = (ln(OptimalLoad / CurrentLoad) * 100) / 2;
5
6         if (LocalTDFStep > MaxTDFStep) {
7             LocalTDFStep = MaxTDFStep;
8         }
9
10        newTDF = TDF - LocalTDFStep;
11
12        if (newTDF > MinTDF) {
13            SetTDF(newTDF);
14        } else {
15            SetTDF(MinTDF);
16            break;
17        }
18
19        Sleep(X * 2);
20    }
21 }
```

---

Anmerkung: Der Systemstatus kann jederzeit durch einen Interrupt des Global-State-Moduls geändert werden. Damit kann ein anderer Algorithmus angestoßen werden und der aktuelle Algorithmus terminiert.

### 6.4.2.7 Feintuning bei Optimallast

Da die Optimallast ein Bereich und kein diskreter Wert sein kann, kann es sein, dass die Last durch die Über- oder Unterlastbehandlung an die untere Grenze dieses Bereiches gebracht wird. Sollte nun die Last konstant bleiben und keine weitere Adaption angestoßen werden verschenkt man damit Zeit da die Last durchaus im oberen Bereich des Optimallastbereiches sein könnte. Somit benötigt man einen Algorithmus der den TDF in gewissen Abständen (beispielsweise alle 500 ms) um einen minimalen Betrag erhöht, so dass gewährleistet ist, dass die Last möglichst im oberen Bereich des Optimallastbereiches verläuft.

Listing 6.17 zeigt dieses Feintuning.

**Listing 6.17** Algorithmus zum Feintuning im Optimallastbereiches beim Regler-basierten Ansatz

```

1 void StartFinetuning()
2 {
3     while (Systemstate == OPTIMAL) {
4         newTDF = TDF - 1;
5
6         if (newTDF > MinTDF) {
7             SetTDF(newTDF);
8         } else {
9             SetTDF(MinTDF);
10            break;
11        }
12
13        Sleep(500);
14    }
15 }

```

Anmerkung: Der Systemstatus kann jederzeit durch einen Interrupt des Global-State-Moduls geändert werden. Damit kann ein anderer Algorithmus angestoßen werden und der aktuelle Algorithmus terminiert nach Durchlaufen der Schleife.

**6.4.2.8 Reaktion bei Überlast**

Sobald das Global-State-Modul Überlast signalisiert beginnt der Regelvorgang durch den Regelalgorithmus. Hierbei wird, ähnlich der Unterlastbehandlung, der Quotient aus dem aktuellen Lastwert und dem optimalen Lastwert ermittelt. Allerdings kommt hierbei ein Wert heraus durch den die aktuelle Virtuelle Zeit geteilt werden muss um die Last auf den Optimalwert zu bringen. Aus der anfangs vorgestellten Formel für die Berechnung der Virtuellen Zeit aus dem TDF:

$$T'_{virtuell} = T'_{real} \times 2^{(TDF/100)}$$

lässt sich für diesen Fall die Formel

$$TDFStep = \ln\left(\frac{\text{aktuellerLastwert}}{\text{optimalerLastwert}}\right) * 100$$

herleiten. Das Ergebnis ist der TDF-Schritt, um den der aktuelle TDF erhöht werden muss, damit die Last auf den Optimalwert befördert wird. Ist dieser Schritt nun kleiner oder gleich dem Wert des **MaxTDFStep**, wird der neue TDF eingestellt. Ist der Sprung zu groß, wird der alte TDF nur um den Betrag des **MaxTDFStep** erhöht. Dieses Vorgehen wird so lange wiederholt bis die Last wieder dem Optimalwert angenähert ist. Sollte zwischenzeitlich der Wert des TDF auf den **MaxTDF** angestiegen sein wird diese Adaption hierdurch beendet. Auch hier kann der Timer X durch die oben angegebene Formel für die Berücksichtigung

des aktuellen Zustands variiert werden.

Listing 6.18 zeigt den Algorithmus zur Überlastbehandlung.

---

**Listing 6.18** Algorithmus zur Überlastbehandlung beim Regler-basierten Ansatz

---

```
1 void StartOverloadHandling()
2 {
3     while (Systemstate == OVERLOADED) {
4         LocalTDFStep = ln(CurrentLoad / OptimalLoad) * 100;
5
6         if (LocalTDFStep > MaxTDFStep) {
7             LocalTDFStep = MaxTDFStep;
8         }
9
10        newTDF = TDF + LocalTDFStep;
11
12        if (newTDF < MaxTDF) {
13            SetTDF(newTDF);
14        } else {
15            SetTDF(MaxTDF);
16            break;
17        }
18
19        Sleep(X);
20    }
21 }
```

---

Anmerkung: Der Systemstatus kann jederzeit durch einen Interrupt des Global-State-Moduls geändert werden. Damit kann ein anderer Algorithmus angestoßen werden und der aktuelle Algorithmus terminiert nach Durchlaufen der Schleife.

### 6.4.2.9 Basisalgorithmus

Listing 6.19 zeigt den Basisalgorithmus des Regler-basierten Ansatzes.

---

**Listing 6.19** Basisalgorithmus des Regler-basierten Ansatzes

---

```
1 void AdaptTDF(newSystemstate, newSystemload)
2 {
3     Systemstate = newSystemstate;
4     CurrentLoad = new Systemload;
5
6     if (Systemstate == UNDERLOADED) {
7         StartUnderloadHandling();
8     } else if (Systemstate == OVERLOADED) {
9         StartOverloadHandling();
10    } else if (Systemstate == OPTIMAL) {
11        StartFinetuning();
12    }
13 }
```

---

Jedes Mal wenn das Global-State-Modul einen Statusbericht sendet wird die Methode *AdaptTDF()* aufgerufen und ihr der neue Systemstatus und die neue Last übergeben. Dann werden die globalen Variablen entsprechend neu belegt und der entsprechende Algorithmus zur Reaktion auf den neuen Status gestartet. Dieser läuft so lange bis entweder ein Grenzwert für den TDF erreicht wird oder das Ziel der Behandlung erreicht wurde und sich der Systemstatus dadurch geändert hat. Diese Änderung wird durch das Global-State-Modul mitgeteilt. Die Schleife der einzelnen Algorithmen wird daraufhin verlassen und die Methode beendet.

### 6.4.2.10 Diskussion

Mit diesem Regler-basierten Ansatz haben wir ein System geschaffen welches sich zu Beginn eines Experiments mittels eines timergesteuerten Zyklus langsam an die Optimallast herantastet. Ist diese Optimallast erreicht beginnt die Korrekturphase. Dabei wird abhängig vom aktuellen Lastwert reagiert. Im Gegensatz zum TCP-basierten Ansatz wird beim Regeln nur zwischen drei Zuständen unterschieden: *UNDERLOADED*, *OPTIMAL* und *OVERLOADED*. Hier wird zudem ein optimaler Regelwert errechnet und nicht, wie beim TCP-basierten Ansatz, um einen konstanten Faktor adaptiert. Da in beide Richtungen unterschiedlich stark geregelt wird, wird auch das Aufschaukeln minimiert. Durch das Feintuning wird eine Ausführung des Experiments bei möglichst hoher Last realisiert.



# Evaluation

---

Bei der Evaluation werden die einzelnen Konzepte mit verschiedenen Einstellungen und Kombinationen an diversen Szenarien getestet und verbessert. Die dabei auftretenden Probleme werden untersucht und geeignete Lösungen entwickelt. Zunächst werden die Szenarien vorgestellt anhand derer die Evaluation und der Test stattfindet.

## 7.1 Evaluierungsszenarien

Da ein möglichst allgemeines Konzept zur Adaption des TDF entwickelt werden soll, ist es notwendig, die entwickelten Konzepte und Algorithmen an verschiedenen Szenarien zu testen. Hierbei sollten im Kern unterschiedlich geartete Szenarien zum Einsatz kommen. Prinzipiell lassen sich alle Problembereiche in zwei Kategorien einteilen.

Zum einen gibt es Anwendungen in denen die Last sehr schwankend ist. Das kommt meist davon, dass viele Prozesse gleichzeitig auf einer Maschine laufen. Diese Prozesse werden in unterschiedlichen Abständen aktiv und erzeugen so eine Last.

Zum anderen gibt es Szenarien mit eher gleichbleibender Last. Dies kommt daher, dass eher weniger Prozesse auf einer Maschine laufen, die dann länger laufen und so eine konstante Last erzeugen.

Wir haben uns deshalb für insgesamt drei Szenarien entschieden. Eines, bei dem der Lastverlauf relativ wechselhaft ist, eines, bei dem eher ein konstanter Lastverlauf auftritt und eines, bei dem die beiden Problembereiche überlagert werden, also eine Anwendung mit wechselhaftem Lastverlauf und eine mit konstantem Lastverlauf parallel auf einem Knoten laufend.

### 7.1.1 Optimized-Link-State-Routing

Das Optimized-Link-State-Routing ist ein Szenario mit eher wechselhaftem Lastverlauf. Im gesamten System wird ein Netz aus mobilen Knoten aufgebaut (beispielsweise 10 x 10 Knoten) die untereinander ein OLSR-Protokoll [17] [4] fahren. Eingesetzt wird ein Open-Source OLSR Daemon [3]. Somit haben wir in Summe 100 Knoten die sich in unserem Fall auf 2

Computenodes verteilen. Das ergibt auf jedem Computenode 50 vNodes. Die Knoten sind so angeordnet, dass sie nur in einer Gitternetzform miteinander kommunizieren können. Abbildung 7.1 zeigt die Verteilung und Anordnung der Knoten auf den beiden Computenodes.

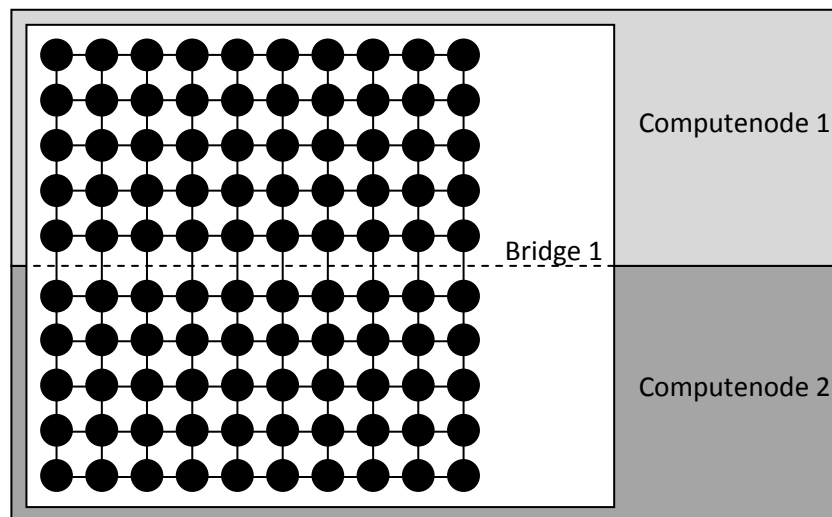


Abbildung 7.1: Anordnung der Knoten beim OLSR-Szenario

Da das Protokoll ständig Link-State-Nachrichten zur Aufrechterhaltung und Aktualisierung sendet, gibt es einen regen Kommunikationsbedarf. Da jeder Knoten für sich seine Pakete scheduled schwankt die Last in gewisser Weise andauernd.

Abbildung 7.2 zeigt einen Lastverlauf, wie er in diesem Szenario vorkommt.

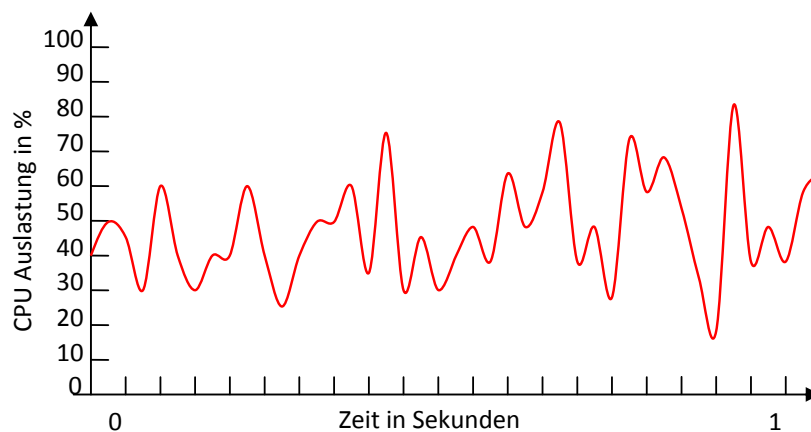


Abbildung 7.2: Beispiel für einen eher wechselhaften Lastverlauf wie er beim OLSR-Szenario entsteht

Mit diesem Szenario lassen sich die Adaptionseigenschaften in einem wechselhaften Szenario gut untersuchen. Allerdings ist dieses Szenario mit Vorsicht zu genießen, da die Last der Knoten so gering ist, dass der TDF negativ ist. Hier sei noch einmal auf die Problematik der Ausführung eines Experiments schneller als Echtzeit (Kapitel 4.6) hingewiesen. Dennoch wird ein solches Szenario evaluiert, da prinzipiell die Adaption auch für solche Szenarien funktionieren sollte.

Für die Evaluierung wird das aufgesetzte Szenario für 120 Sekunden gemessen. Diese Messung wird  $3 \times 4$  Mal vollzogen. Heißt konkret, dass drei Mal jeweils vier Messungen durchgeführt werden. Dies hat den Grund, dass aus einem laufenden System Daten ausgelesen werden. Dadurch wird Last erzeugt die nichts mit der eigentlichen Experimentlast zu tun hat. Dadurch werden die Messergebnisse verfälscht. Deshalb wird ein 4er-Block so gemessen, dass erst eine Messfehlerbeseitigung stattfindet (Messung des Overhead durch das Auslesen) und danach vier Messdurchläufe gemacht werden. Die Messfehler werden danach von jedem Messdurchlauf abgezogen. Ein Teilergebn ergibt sich aus dem Mittelwert der Ergebnisse der vier Messdurchläufe von denen die Messfehler abgezogen sind. Da auch diese Messfehlerbeseitigung Schwankungen unterliegt wird dieser Messdurchgang insgesamt drei Mal wiederholt. Das Endergebnis ist der Mittelwert der jeweiligen drei Teilergebnisse.

Über die Korrektheit kann nur eine sehr vage Aussage getroffen werden. Als Metrik kann die Anzahl der gesendeten OLSR-Nachrichten genommen werden. Allerdings kann nicht ermittelt werden ob die Nachrichten „pünktlich“ angekommen, oder einige durch eine Überlast verspätet beim Empfänger eingetroffen sind. Dieses Szenario ist relativ unempfindlich gegenüber kurz auftretender Überlast. Messungen haben ergeben, dass die Anzahl der Nachrichten auch bei oft auftretender Überlast korrekt scheint. Wir wollen aber ein System entwickeln, das selbst die kleinste Überlast erkennt und behandelt. Daher können wir dieses Szenario für unsere empfindliche Anwendung so nicht verwenden. Mit dem noch vorzustellenden Szenario mit TCP-Verbindungen kann man die absolute Korrektheit überprüfen, da eine TCP-Verbindung sehr empfindlich auf selbst kleine Überlast reagiert. Verzögerte Pakete machen sich dabei direkt beim Durchsatz bemerkbar.

### 7.1.2 Routerkette

In diesem Szenario soll die Adaption mit einer Routerkette beliebiger Länge evaluiert werden. Auch hier kommen zwei Computenodes zum Einsatz. Auf einem Knoten befindet sich ein Sender und ein Empfänger der TCP-Pakete. Auf dem anderen Knoten wird eine beliebig lange (maximal 253 Router) Routerkette aufgesetzt. Nun wird eine TCP-Verbindung beliebiger Länge aufgebaut, die zwischen dem Sender und Empfänger durch die Routerkette führt und mit voller Leistung sendet. Zum Senden der Pakete wird netperf [2] verwendet.

Abbildung 7.3 zeigt die Anordnung der Routerkette und der Knoten auf den beiden Computenodes.

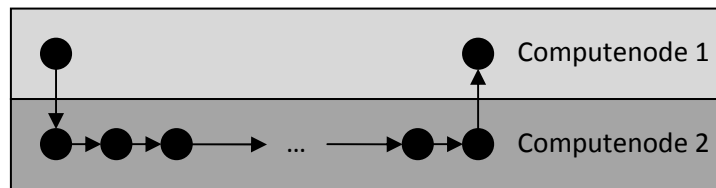


Abbildung 7.3: Anordnung der Routerkette und der Knoten beim Routerketten-Szenario

Dieses Szenario produziert einen eher glatten Lastverlauf, der, basierend auf der Anzahl der Verbindungen und der Anzahl der Router, mehr Last oder eben weniger Last erzeugt. In Abbildung 7.4 sieht man ein Beispiel eines solchen Lastverlaufes.

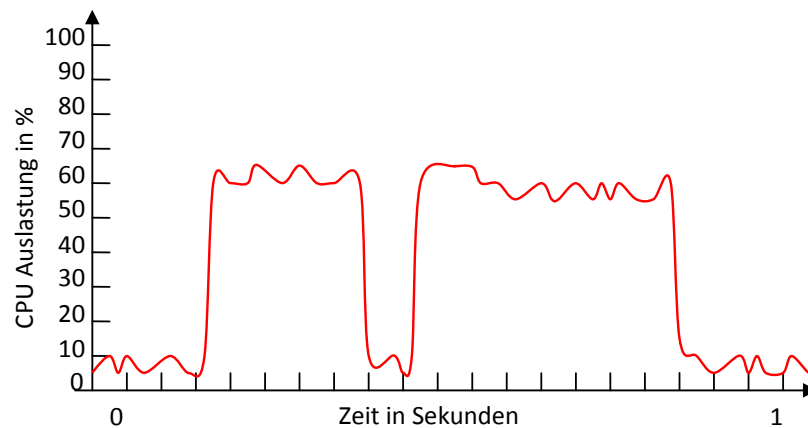


Abbildung 7.4: Beispiel für einen eher glatten Lastverlauf wie er bei der Routerkette entsteht

Bei der Routerkette lassen sich große Lastsprünge und die Adaption darauf untersuchen. Da im normalen Zustand kaum Last anliegt, kann die Virtuelle Zeit relativ schnell laufen. Sobald nun eine TCP-Verbindung gestartet wird tritt schlagartig Last auf. Abhängig davon, wie lange die Routerkette ist, fällt dieser rasante Lastanstieg entsprechend stark aus. Die Adaption muss also umgehend den extremen Lastanstieg bemerken und sofort die Virtuelle Zeit verlangsamen. Nachdem die TCP-Verbindung steht und die Übertragung im Gange ist ändert sich die Last kaum. Allerdings kann es bei einer langen Routerkette zu kleinen Schwankungen kommen, die dann durch die Adaption ausgeglichen werden sollten. Diese rühren daher, dass hier viele Router parallel auf einem System zum Einsatz kommen. Nachdem die Übertragung abgeschlossen wurde sollte die Virtuelle Zeit wieder entsprechend beschleunigt werden.

Für die Evaluierung verwenden wir eine TCP-Verbindung die zwei Sekunden andauert. Davor und danach werden jeweils 10 Sekunden ohne Verbindung gemessen. Die Zeiten sind deshalb so gewählt, da die 10 Sekunden beim niedrigsten TDF ablaufen und somit sehr schnell vorbei sind (bei TDF -400 ca. 0,625 s), die TCP-Verbindung hingegen bei relativ

hohem TDF, so dass hier die Zeit zum Messen extrem lang wird (bei TDF 600 ca. 128 s). Das gesamte Szenario dauert also 22 Sekunden Virtueller Zeit und abhängig davon wie lange die Routerkette ist bis zu zwei Minuten in Echtzeit. Hierbei befindet sich die Routerkette auf dem Knoten 2 und der Sender und Empfänger auf dem Knoten 1. Die Links zwischen den einzelnen Routern sind jeweils Gigabit-Links. Der Sender und der Empfänger sind mit 100 Megabit an die Routerkette angeschlossen. Die Messergebnisse der Evaluation bilden sich ebenfalls aus dem Mittelwert der  $3 \times 4$  Messungen. Da auch hier die Daten aus einem laufenden System ermittelt werden entsteht ein Overhead. Dieser wird wie oben beschrieben mittels eines Messfehlerlaufs ermittelt und von den Ergebnissen abgezogen. Deshalb erfolgen hier auch  $3 \times 4$  Messungen. Der maximal erreichbare TCP Durchsatz in diesem Szenario, bei unterschiedlich langen Routerketten, ist in Abbildung 7.5 dargestellt. Der TDF wurde jeweils so gewählt, dass keine Überlast entstehen kann, da die Last nie über 40% (Messintervall = 100 ms) steigt.

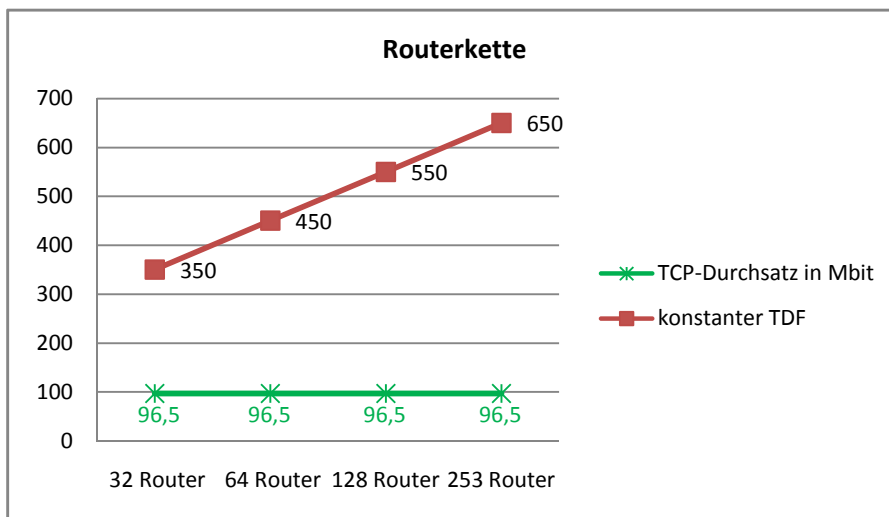


Abbildung 7.5: maximal erreichbarer TCP-Durchsatz des Routerketten-Szenarios bei unterschiedlich langen Routerketten.

### 7.1.3 OLSR mit TCP-Verbindung

Das dritte Szenario besteht aus einer Kombination aus den beiden vorher beschriebenen Szenarien. Hier soll als Basislast der Knoten das OLSR-Szenario dienen. Zudem befinden sich in einem abgetrennten Netzwerk auf einem Computeknoten zwei weitere Knoten, zwischen denen in bestimmten Abständen TCP-Verbindungen aufgebaut und die Übertragung mit Gigabit getestet wird. Auch hier kommen der OLSR-Deamon [3] und netperf [2] zum Einsatz.

Abbildung 7.6 zeigt die Anordnung der Knoten in diesem Szenario.

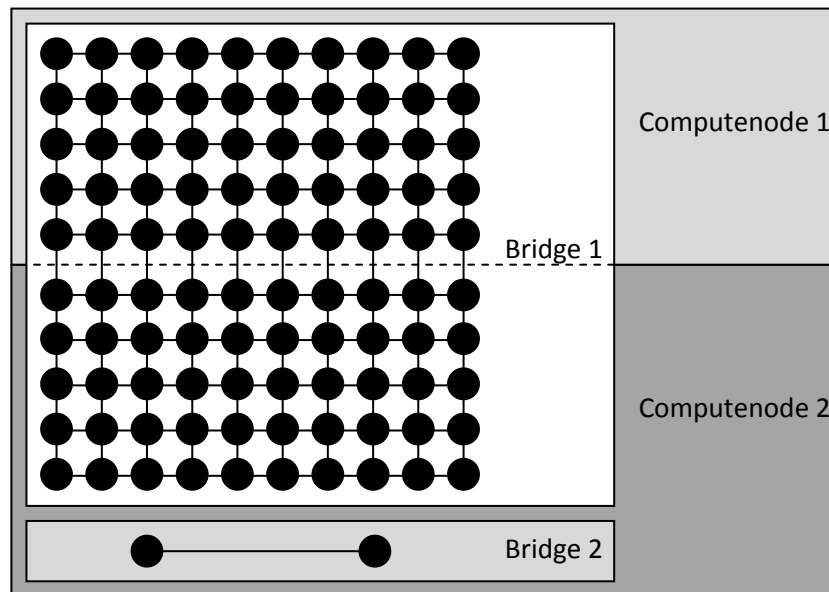


Abbildung 7.6: Anordnung der Knoten beim OLSR-Szenario in Kombination mit den parallel verlaufenden TCP-Verbindungen

Man kann jetzt also die Adaption mit einer wechselhaften Grundlast, kombiniert mit einem rasanten Anstieg der Last und dann zusätzlich zur wechselhaften Basislast die höhere durch die TCP-Verbindung erzeugte konstante Last, evaluieren. Der TDF ist in den Phasen, in denen keine TCP-Verbindung besteht, negativ, so dass das Experiment schneller als Echtzeit abläuft. Während der Verbindungen steigt der TDF im Schnitt auf ca. 200 an. Durch die Einführung der TCP-Verbindungen kann nun eine Korrektheit ermittelt werden. Somit hat man das Problem der Korrektheitermittlung des reinen OLSR-Szenarios behoben.

Für die Evaluation werden insgesamt drei TCP-Verbindungen unterschiedlicher Länge aufgebaut und gemessen. Die TCP-Verbindungen laufen auf dem Knoten 2. Der Knoten 1 hat nur das OLSR-Szenario. Vor der ersten, nach der letzten und zwischen jeder Verbindung wird das OLSR-Szenario 10 Sekunden gemessen. Die erste TCP-Verbindung hat eine Länge von einer Sekunde. Die zweite Verbindung dauert 9 s und die dritte 5 s lang. Somit ergibt sich eine Gesamtlauzeit einer Messung von 55 Sekunden Virtueller Zeit. Diese Längen wurden deshalb so gewählt, da eine kurze, eine lange und eine mittlere TCP-Verbindung berücksichtigt werden soll. Diese Messung wird wegen den durch das Auslesen der Messergebnisse entstehenden Messfehlern 3 x 4 Mal wiederholt und die Ergebnisse durch Mittelwertbildung berechnet. Der Sender und der Empfänger der TCP-Verbindung sind mit einem Gigabit-Link miteinander verbunden. Der maximal erreichbare TCP Durchsatz in diesem Szenario liegt demnach bei ca. 965,2 MBit/s. Dieser Wert wurde mit einem konstanten TDF von 250 ermittelt. Auch hier entstand keine Überlast.

## 7.2 Kriterien

Bei jeder Evaluation gibt es Kriterien die erfüllt werden müssen, bzw. nach denen eine Bewertung stattfindet. Diese werden hier kurz erläutert.

### 7.2.1 Korrektheit

Da wir ein System erschaffen wollen, welches es ermöglicht nahezu alle Anwendungen, ohne durch die Virtualisierung erzeugte Messfehler, korrekt ablaufen zu lassen, steht das oberste Augenmerk auf der Transparenz der Adaption, also auf der Korrektheit der gemessenen Ergebnisse. Bei der Routerkette und den TCP-Verbindungen kann man relativ genau sagen, ob die gemessenen Ergebnisse korrekt oder durch die Virtualisierung verfälscht sind, da man den optimalen Durchsatz für eine Verbindung anhand der Netzwerkeigenschaften ermitteln kann bzw. bereits kennt. Für die Routerkette beträgt der maximale Durchsatz ca. 96,5 MBit/s. Bei den TCP-Verbindungen beim OLSR-Szenario kommt man auf 965,2 MBit/s. Wenn man nun eine Korrektheit von 99,5% anstrebt ergibt sich ein minimaler korrekter Durchsatz bei den Routerketten von 96,0 MBit/s. Bei den Gigabit Verbindungen beim OLSR-Szenario ergibt dies einen minimalen korrekten Durchsatz von ca. 960,4 MBit/s. Im Verlauf der Evaluation werden viele Messungen durchgeführt. Wenn von korrekten Ergebnissen gesprochen wird heißt dies immer, dass der TCP-Durchsatz über diesen beiden Werten liegt.

### 7.2.2 Overhead

Der Overhead, der durch die Adaption entsteht, sollte möglichst gering sein. Es sollten beispielsweise nur sehr wenige Lastinformationsnachrichten der einzelnen Computenodes zum Koordinator gesendet werden. Somit skaliert die Adaption auch für große Systeme. Des Weiteren sollten natürlich diese wenigen Lastnachrichten genügen um den TDF so anzupassen, dass die oben erwähnte Korrektheit erreicht wird. Zudem sollte die Anzahl der TDF-Anpassungen, wie die Anzahl der Lastnachrichten, minimal sein, so dass auch der umgekehrte Netzwerkanal vom Koordinator zu den Computenodes nicht unnötig überfrachtet wird. In der Evaluation wird der Gesamtaufwand oft berechnet. Dieser setzt sich aus allen gesendeten Lastnachrichten und den TDF-Anpassungen zusammen.

### 7.2.3 Experimentlaufzeit

Als weiteres Kriterium sollte selbstverständlich die zeitliche Komponente eine Rolle spielen. Die Ausführung eines Experiments sollte korrekt sein, wenig Overhead durch die Adaption verbrauchen und natürlich relativ optimal im Zeitverbrauch sein. Somit muss auch darauf Wert gelegt werden, die Ausführungszeit eines Experiments so schnell wie möglich zu machen.

## 7.3 Ergebnisse

Im Folgenden werden die einzelnen Konzepte mit verschiedenen Werten getestet. Hierbei wird eine optimale Konfiguration der einzelnen Parameter herausgefunden. Zudem werden hier die Konzepte auf ihre Brauchbarkeit und Funktionsweise überprüft. Dabei können sich einzelne Ideen als wertlos herausstellen, andere hingegen als unerlässlich. Der Aufbau der Evaluation beschreibt den Weg zur optimalen Konfiguration der Adaption.

### 7.3.1 Festlegung der Extremwerte des TDF

Bei der Festlegung der Extremwerte des TDF gibt es prinzipiell keine Grenzen. Man kann allerdings beobachten, dass ein kleinerer TDF als -400 für den minimalen TDF keinen Sinn macht, da die Idle-Last eines Computenodes dadurch so hoch ist, dass automatisch adaptiert wird (Abb. 7.7 links). Diese „Idle-Adaption“ geschieht ab einem TDF von ca -420. Somit hat man selbst dann wenn kein Experiment läuft ständig Adaptionen. Dies bewirkt überflüssigen Overhead. Zudem kann man sehen, dass durch diese Adaption der TDF bis -500 gehen kann, so dass es bei einem realen Szenario durch die schnelle Virtuelle Zeit zu Überlast kommen kann. Hier sei nochmals auf die Problematik der Ausführung eines Experiments schneller als Echtzeit hingewiesen (Kapitel 4.6). Bei einem minimalen TDF von -400 kommt eine „Idle-Adaption“ noch nicht vor (Abb. 7.7 rechts).

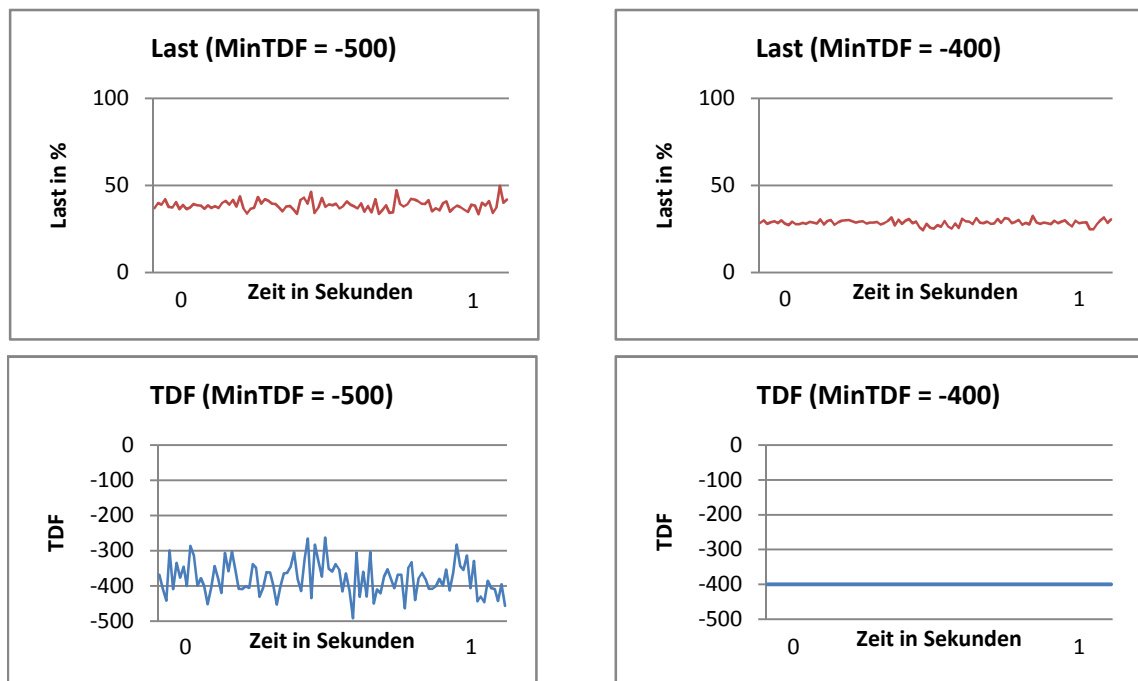


Abbildung 7.7: Minimaler TDF im Vergleich

Bei der Festlegung des maximalen TDF sind im Prinzip keine Grenzen gesetzt. Aus technischen Gründen haben wir hier den maximalen TDF auf 1.000 gesetzt. Dies ermöglicht allerdings immer noch eine Ausführung eines Experiments in 1.000fach verlangsamer Geschwindigkeit. In der Praxis kam dieser Fall bei der vorliegenden Evaluation nie vor.

### **7.3.2 Initiale Adaption / Reglerbereich**

Bei den Adaptionskonzepten wurde ein Konzept zur initialen Adaption vorgestellt. Dabei wurde der TDF auf den maximalen Wert gesetzt und dann periodisch um einen minimalen Faktor reduziert. Bei der Evaluation sieht man allerdings, dass diese langsame Reduzierung des TDF keinen Vorteil bringt, sondern nur auf Kosten der Experimentlaufzeit geht. Wenn man sofort mit der regulären Regelung beginnt kommen dieselben korrekten Messergebnisse heraus und es geht deutlich schneller. Somit gibt es beim Regler-basierten Ansatz auch keinen Nicht-Reglerbereich mehr, es wird sofort mit dem normalen Regelvorgang begonnen. Dennoch wird der TDF zu Beginn der Adaption auf den maximalen TDF gesetzt.

### **7.3.3 Trigger**

#### **7.3.3.1 Eventgesteuerter Trigger**

Der eventgesteuerte Trigger meldet die vorherrschende Last nur bei Auftreten eines Events. Das Event muss eine gewisse Dauer ohne Unterbrechnung aufgetreten sein. Hierbei kann es passieren, dass die Events nie lange genug ohne Unterbrechnung auftreten und somit nie eine Last getriggert wird. Dadurch wird nie adaptiert. Ein weiteres Problem besteht darin, dass ein Überlastevent schneller erkannt werden muss als ein Unterlastevent. Somit ist die Dauer der Events unterschiedlich lang. Wenn nun ein Überlastevent auftritt und die Adaption darauf beginnt, wird der TDF bis zum nächsten Interrupt durch ein anderes Event periodisch erhöht. Da aufgrund der Berücksichtigung des aktuellen Zustandes die Dauer, die ein Event auftreten muss, ständig wächst und somit die benötigte Dauer für ein Unterlastevent wächst, wird der TDF durch das vorausgehende Überlastevent bis zum maximalen Wert erhöht. Aufgrund des fehlenden Unterlastevents kann das System in ein Deadlock geraten. Dies kann in Abbildung 7.8 beobachtet werden. Deshalb ist der eventgesteuerte Trigger für unser System keine Option.

#### **7.3.3.2 Timergesteuerter Trigger**

Der timergesteuerte Trigger ist ein starrer Trigger, der immer nach Ablauf einer gewissen Zeit die in dieser Zeit vorherrschende Last dem Filter meldet. Es wird demnach ein kontinuierlicher Lastverlauf getriggert, der eine gute Grundlage zur Adaption darstellt. Durch die event-unabhängige Signalisierung kann dieser Trigger nicht in ein Deadlock geraten. Mit Hilfe der Berücksichtigung des aktuellen Zustandes kann die Empfindlichkeit des Triggers eingestellt werden.

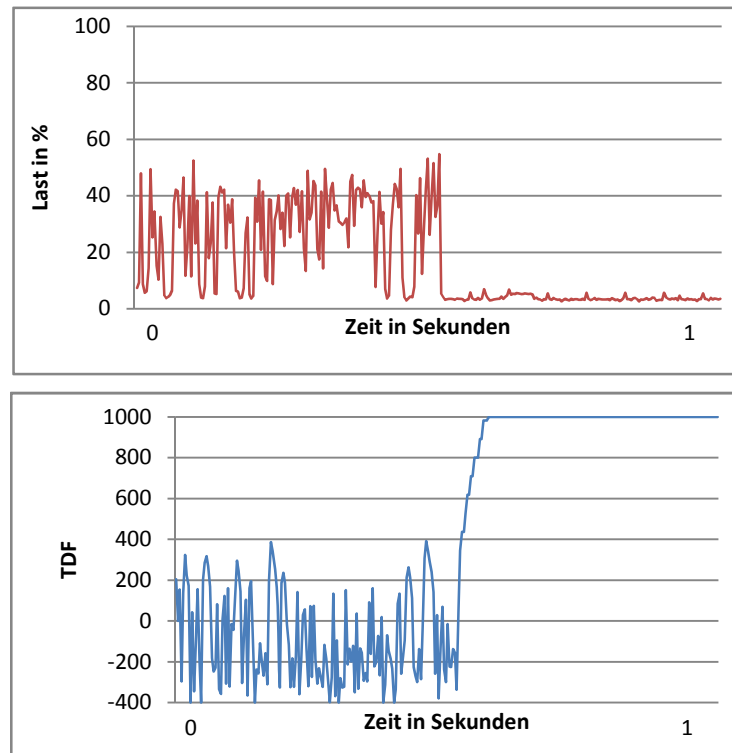


Abbildung 7.8: Deadlock des eventgesteuerten Triggers

### 7.3.3.3 Festlegung der Timerintervalle beim timergesteuerten Trigger und Berücksichtigung des aktuellen Zustandes

Bei der Festlegung der Timerintervalle des timergesteuerten Triggers kann beobachtet werden, dass als minimales Timerintervall 2 ms optimal sind. Dies ist die minimalste Zeit, die genommen werden kann. Wie bereits in Kapitel 2.6 beschrieben hat ein Adaptionsschritt eine Reaktionszeit. Diese kann im Worst-Case mit ca. 2 ms angenommen werden. Somit ist die minimalste Zeit festgelegt. Allerdings sollte dieser Timer auch nicht größer sein, da sonst große Lastschwankungen zu spät erkannt werden.

Abbildung 7.9 zeigt den Vergleich einzelner Timerintervalle am Beispiel der Routerkette mit 32 Routern. Es wird kein Filter verwendet.

Man kann deutlich erkennen, dass der TCP-Durchsatz bei einem Timerintervall über 2 ms unter den gewünschten 96,0% liegt. Deshalb kommt für dieses Timerintervall nur der Wert 2 ms in Frage.

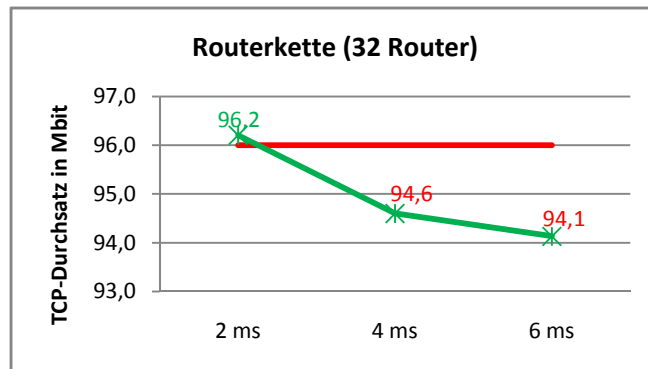


Abbildung 7.9: Vergleich der Korrektheit der ermittelten Messergebnisse bei verschiedenen Timer-Intervallen des Triggers.

Die Berücksichtigung des aktuellen Zustandes bewirkt, dass sich das Timerintervall, wie auch die Virtuelle Zeit, bei einem Anstieg des TDFs um 100 verdoppelt oder wieder halbiert. So kann erreicht werden, dass die maximale Anzahl an Nachrichten durch die Virtuelle Zeit bestimmt wird und nicht durch die Echtzeit. Dadurch sinkt die maximale Anzahl der Nachrichten pro Echtzeitdauer. Der Wert des maximalen Timerintervalls bestimmt wie lange der aktuelle Zustand berücksichtigt werden soll. Prinzipiell ist dieser Wert beliebig. Allerdings bedeutet dies, dass er bei TDF = 1.000 bis auf 2 s ansteigen kann. Dies ermöglicht sicherlich keine sinnvolle Adaption mehr, da hier Lastschwankungen nicht mehr erkannt werden können. Um den aktuellen Zustand bis zu einem TDF von 800 mitzuberechnen genügt hierfür ein maximales Timerintervall von 512 ms.

Abbildung 7.10 zeigt die Timerintervalle bei der Berücksichtigung des aktuellen Zustandes.

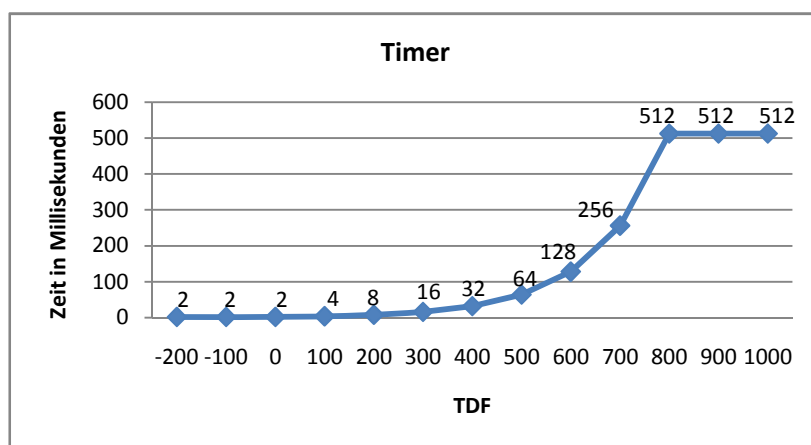


Abbildung 7.10: Verlauf des Timerintervalls bei der Berücksichtigung des aktuellen Zustandes.

Abbildung 7.11 zeigt die Ausführung einiger Experimente einmal mit Berücksichtigung des aktuellen Zustandes und einmal ohne Berücksichtigung des aktuellen Zustandes im Vergleich.

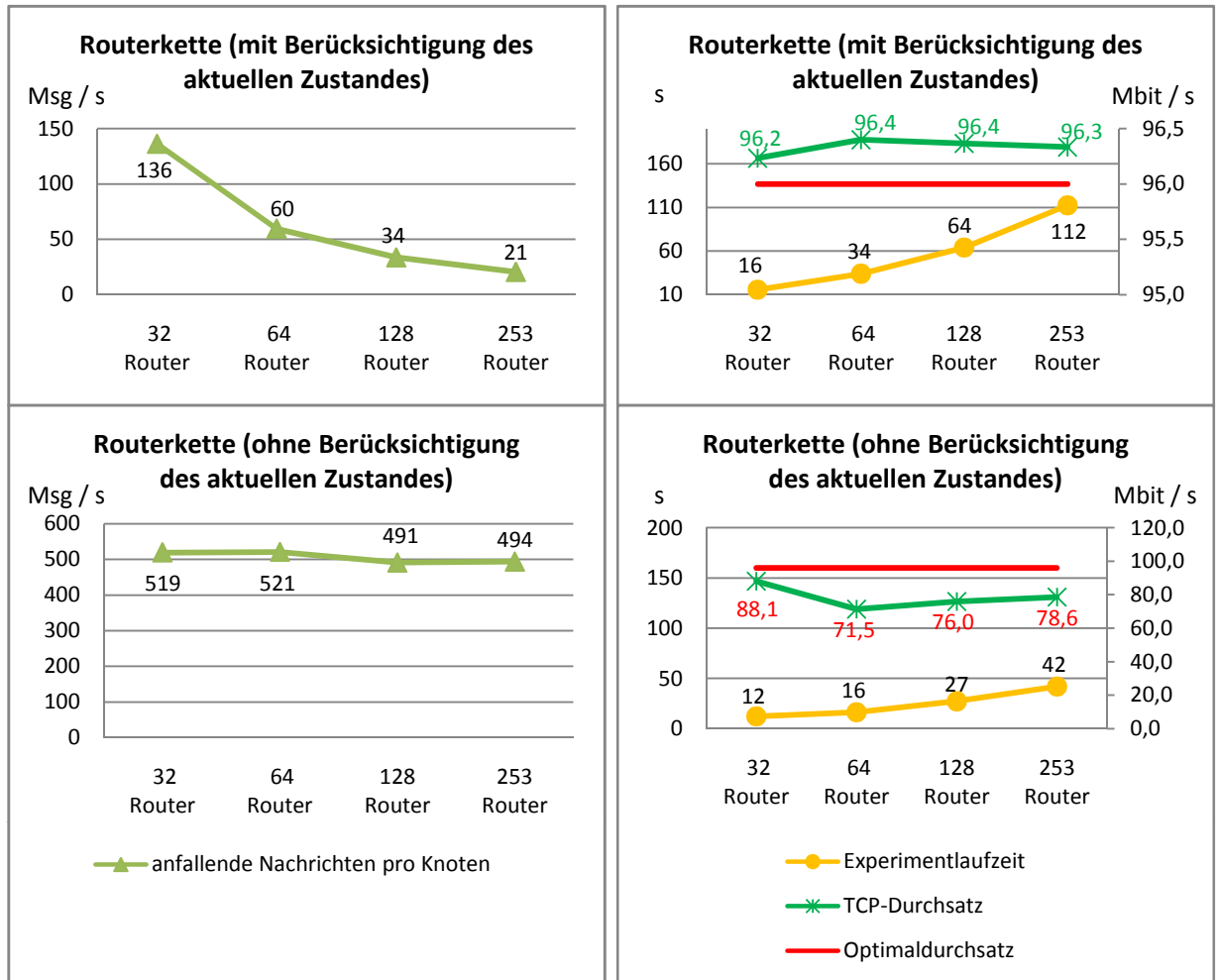


Abbildung 7.11: Vergleich der Ausführung von Routerketten verschiedener Länge mit Berücksichtigung des aktuellen Zustandes und ohne Berücksichtigung des aktuellen Zustandes.

Man kann bei dem Fall mit „Berücksichtigung des aktuellen Zustandes“ deutlich erkennen, dass, obwohl die Experimentlaufzeit mit der Länge der Routerkette ansteigt, die Anzahl der anfallenden Lastnachrichten pro Sekunde deutlich sinkt. Zudem sind alle Ergebnisse korrekt. Ohne Berücksichtigung des aktuellen Zustandes bleibt die Anzahl der gesendeten Nachrichten pro Sekunde konstant. Zudem sind hier die Ergebnisse nicht mehr korrekt.

Ein weiterer Effekt bei der „Berücksichtigung des aktuellen Zustandes“ ist, dass die Adaption bei hohem TDF ruhiger und dadurch erst richtig möglich wird. Die beiden Lastverläufe (Abbildung 7.12) zeigen die Adaption einer Routerkette mit 253 Routern einmal mit Berücksichtigung des aktuellen Zustandes und einmal ohne.

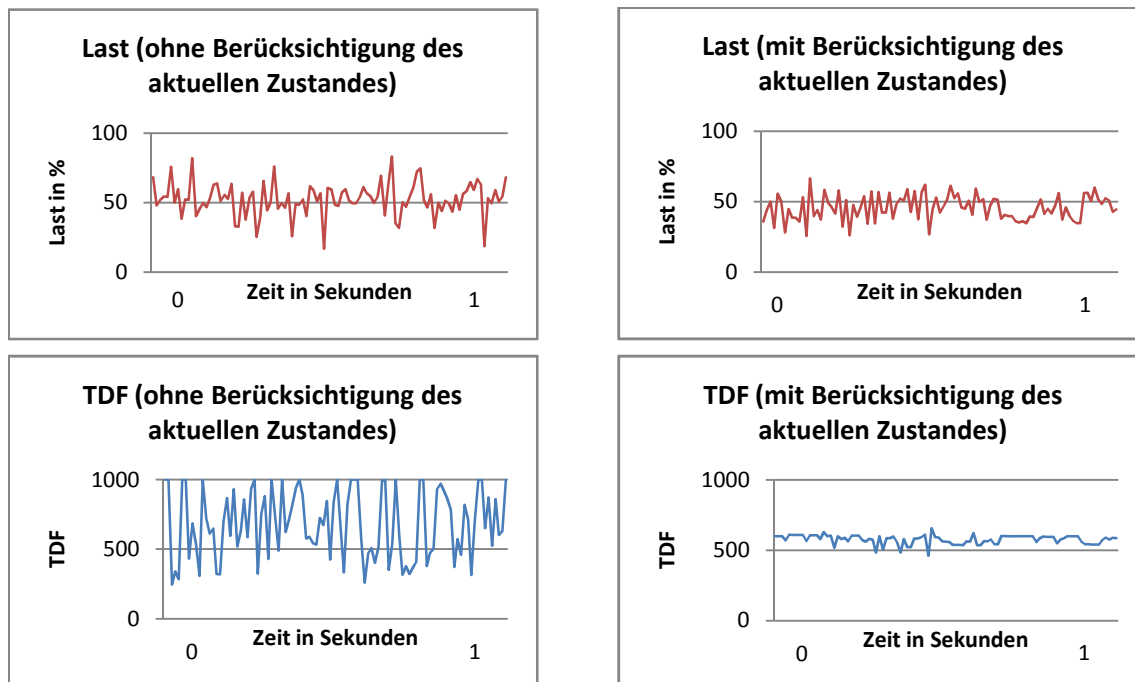


Abbildung 7.12: Vergleich der Adaption mit Berücksichtigung des aktuellen Zustandes und ohne Berücksichtigung des aktuellen Zustandes.

Man kann deutlich erkennen, dass im Fall ohne „Berücksichtigung des aktuellen Zustandes“ eine sehr unruhige Adaption stattfindet, die keine korrekten Ergebnisse produziert (Abbildung 7.11).

### 7.3.4 Festlegung der Intervalle des Adaptions-Timers

Der Adaptionstimer ist eng mit dem Trigger-Timer verbunden. Eine durch den Adaptionstimer angestoßene Adaption sollte nicht vor einer Adaption, die durch einen neuen globalen Zustand angestoßen wurde stattfinden, so dass die vorhergehende Adaption auch ein Feedback liefern kann. Da der neue globale Zustand anhand neuer Nachrichten der Knoten aufgebaut wird und die wiederum abhängig vom Timer des Triggers sind, besteht hier eine enge Kopplung. Der Adaptionstimer sollte also etwas länger als der Trigger-Timer sein. Jedoch sollte es kein vielfaches des Trigger-Timers sein da hierbei Anomalien auftreten können. Zu lang darf der Timer aber auch nicht sein, da sonst zu lange auf eine erneute timergesteuerte Adaption gewartet werden muss und dies zu störender Überlast

führen kann.

Da der Trigger-Timer als minimalen Wert 2 ms eingestellt hat gilt für den Adaptionstimer, dass der Wert 3 ms der Beste ist. Als maximalen Wert wollen wir hier, wie auch beim Trigger-Timer, mit der Berücksichtigung des aktuellen Zustandes den TDF = 800 unterstützen. Damit ergibt sich ein maximaler Adaptionstimer von 768 ms.

### 7.3.5 Festlegung des Optimalwertes beim Regler-basierten Ansatz

Der Regler-basierte Ansatz regelt den TDF immer so, dass die Last auf einem Optimalwert zu liegen kommt. Dieser Wert muss definiert werden. Der Wert sollte relativ hoch sein, um die Ausführungszeit zu beschleunigen. Allerdings kann ein zu hoher Wert die Knoten zu häufig in Überlast bringen, so dass die Messergebnisse nicht mehr korrekt sind. Abbildung 7.13 zeigt die Korrektheit der Messergebnisse und die Experimentlaufzeit bei verschiedenen Optimallastwerten. Hierbei wird die Variante „ohne Filter“ verwendet, um die Korrektheit nicht durch einen falsch konfigurierten Filter zu beeinflussen. Als Trigger kommt der timergesteuerte Trigger zum Einsatz.

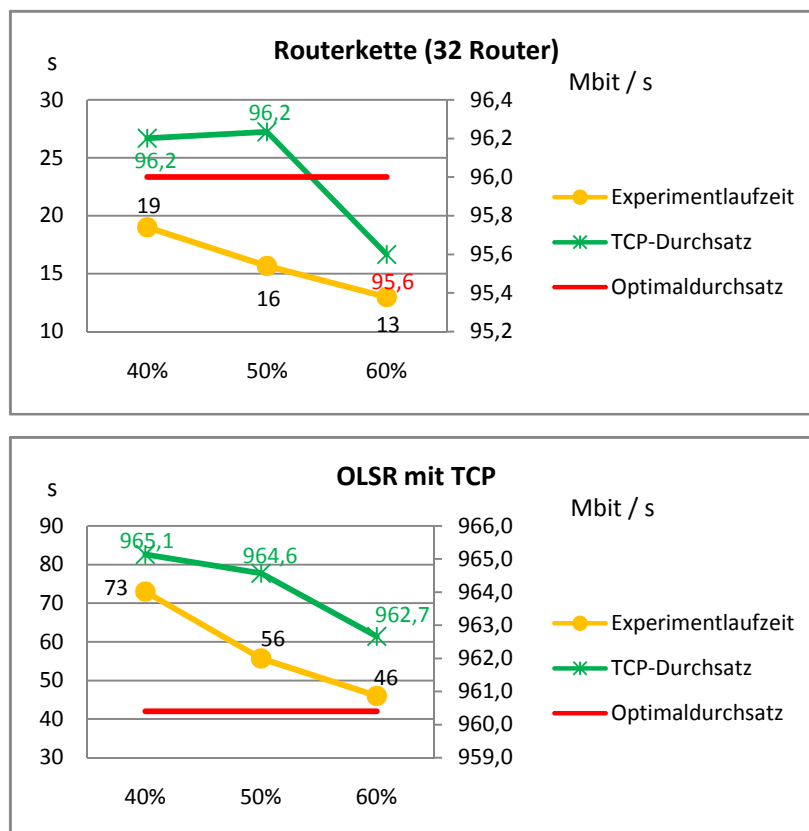


Abbildung 7.13: Vergleich der Korrektheit der Messergebnisse und der Experimentlaufzeit bei verschiedenen Optimallastwerten des Regler-basierten Ansatzes.

Man kann deutlich erkennen, dass ein niedriger Optimalwert zu korrekten Ergebnissen führt. Die Optimallastwerte 40% und 50% liefern korrekte Ergebnisse. Der Wert 60% liefert keine korrekten Ergebnisse mehr. Somit ist der beste hierfür geeignete Wert der Wert 50%. Er liefert die schnellste Ausführungszeit bei korrekten Ergebnissen.

### 7.3.6 Differentielle TDF-Sprünge bei Unterlast und Überlast

Bei der Adaption wird eine Unterlast nur halb so stark behandelt wie eine Überlast. Dies erweist sich als vorteilhaft, da eine zu starke Behandlung der Unterlast die Last zu schnell nach oben treibt, so dass schnell wieder Überlast entsteht. Wenn beispielsweise ein Prozess periodisch aktiv wird hat der Knoten immer abwechselnd hohe Last und niedrige Last. Durch die Adaption werden die Lastspitzen und Täler geglättet, so dass eine einheitliche Auslastung entsteht. Durch die Reaktionszeit der Adaption kann es allerdings passieren, dass die Unterlastbehandlung gerade dann aktiv wird, wenn der Prozess selbst wieder aktiv wird. Durch die auftretende höhere Last und die Unterlastbehandlung steigt die Last dadurch zu schnell an und es kommt zu Überlast. Durch die abgeschwächte Behandlung der Unterlast durch den halben TDF-Sprung und den doppelten Adaptionstimer kann diesem Problem entgegengewirkt werden. Tabelle 7.1 zeigt die Ergebnisse einer Messung mit und ohne differentielle Regelung im Vergleich. Als Regelalgorithmus kommt der Regler-basierte Ansatz mit Schwellenwert-basiertem Filter zum Einsatz.

	Experiment- lauzeit	TCP-Durchsatz	Gesamtaufwand / s
mit differentieller Regelung	13s	96,0 Mbit	94 Nachrichten
ohne differentielle Regelung	12s	95,1 Mbit	134 Nachrichten

Tabelle 7.1: Vergleich der Ausführung einer Routerkette mit 32 Routern mit und ohne differentielle Regelung

Man kann deutlich sehen, dass der TCP-Durchsatz ohne differentielle Regelung nicht mehr unseren Korrektheitsanforderungen genügt. Der Durchsatz liegt deutlich unter 96 Mbit. Aufgrund des bei der Unterlastbehandlung nun schnelleren Adaption-Timers werden bei dieser Variante auch wesentlich mehr Adaptionen getätigt. Auch die Anzahl der gesendeten Nachrichten steigt in die Höhe, so dass der Gesamtaufwand ohne differentielle Regelung wesentlich größer im Vergleich zur Variante mit differentieller Regelung ist. Die Laufzeit eines Experiments sinkt dadurch, dass schneller und härter auf Unterlast reagiert wird und so im Mittel eine höhere Durchschnittslast gefahren werden kann.

### 7.3.7 Festlegung der TDF-Sprünge

Da der Regler-basierte Ansatz den TDF-Sprung mittels einer Formel berechnet ist der TDF-Sprung dadurch schon begrenzt. Sollte die Last bei 100% sein, so muss sie maximal halbiert werden um auf den Optimallastwert von 50% zu gelangen, was einem TDF-Sprung von 100 entspricht. Bei Unterlast kann angenommen werden, dass die Last maximal von 5% auf 50% angehoben werden muss. Da der optimale TDF-Sprung bei Unterlast aber halbiert wird kommt in diesem Fall ein maximaler Sprung von 166 zum Zug.

$$TDFSprung = ((\log_2(50/5))/2) * 100 = 166$$

Da diese Werte durchaus noch in Ordnung scheinen, muss beim Regler-basierten Ansatz kein maximaler TDF-Sprung angegeben werden.

Beim TCP-basierten Ansatz müssen hingegen drei Sprünge angegeben werden. Die Sprünge müssen jeweils die Last vom entsprechenden Status in den Optimallastbereich befördern können. Folglich sind diese Sprünge abhängig von der Lage der Schwellenwerte. Die Sprünge werden so dimensioniert, dass sie die Last innerhalb eines Adaptionsschrittes vom Mittelwert eines Bereiches in den Optimallastbereich befördern. Sollte also der Überlastbereich bei 90% beginnen muss der entsprechende TDF-Sprung so bemessen sein, dass er die Last von 95% in den Optimallastbereich (50% Last) befördert. In diesem Fall wäre dies ein Sprung von ca. 93. Sollte der Schwellenwert für den potentiellen Überlastbereich bei 70% liegen so muss der TDF um 68 erhöht werden. Tabelle 7.2 zeigt die Sprünge für ausgewählte Schwellenwertkonfigurationen. Die Sprünge bei Unterlast sind nur halb so groß wie nötig um eine langsame, problemlose Reaktion auf Unterlast zu garantieren.

Schwellenwert-konfiguration	Unterlast-schwellenwert	potentieller Überlast-schwellenwert	Überlast-schwellenwert
30 / 60 / 70	87	18	77
30 / 60 / 80	87	49	85
30 / 60 / 90	87	58	93
40 / 70 / 80	66	58	85
40 / 70 / 90	66	68	93

Tabelle 7.2: Sinnvolle TDF-Sprünge des TCP-basierten Ansatzes bei ausgewählten Schwellenwertkonfigurationen

### 7.3.8 Vergleich der Filter

#### 7.3.8.1 Ohne Filter

Bei Verwendung der Variante ohne Filter wird jeder ermittelte Lastwert direkt zum Koordinator gesendet. Dies bedeutet, dass in Summe eine immense Anzahl an Lastsignalisierungsnachrichten zum Koordinator gesendet werden. Dies bedeutet einen enormen Aufwand für

den Koordinator, der jede Nachricht verarbeiten muss. Bei dieser Evaluation werden nur zwei Computenodes eingesetzt. Allein diese beiden Nodes schicken so viele Nachrichten, dass die Last des Koordinators ansteigt. Wenn nun noch mehr Knoten am Experiment beteiligt sind und demnach mehr Nachrichten durch den Koordinator zu verarbeiten sind, kommt dieser in einen Bereich, indem er keine zuverlässigen Berechnungen mehr durchführen kann. Somit scheidet dieses Konzept für große Systeme in der Praxis aus. Deshalb sollte dort dann definitiv ein Filter verwendet werden.

Die Tabelle 7.3 zeigt die Anzahl der während der Experimente von den Knoten gesendeten Lastnachrichten, sowie die Anzahl der TDF-Anpassungen bei den verschiedenen Szenarien.

Evaluierungsszenario	gesendete Lastnachrichten Node 1 pro Sekunde	gesendete Lastnachrichten Node 2 pro Sekunde	TDF-Anpassungen pro Sekunde
Routerkette (32 Router)	134	133	128
OLSR mit TCP-Verbindung	256	254	264

Tabelle 7.3: Lastnachrichten und TDF-Anpassungen ohne Filter

### 7.3.8.2 Differentieller Filter

Der Differentielle Filter filtert die getriggerten Lastwerte anhand der Differenz zum vorher gesendeten Wert. Je größer diese Differenz definiert wird, um so besser wird gefiltert. Abbildung 7.14 zeigt die Anzahl der gesendeten Lastnachrichten und die Korrektheit der Messergebnisse bei verschiedenen Lastdifferenzen. Als Trigger kommt der timerge-steuerte Trigger zum Einsatz. Als Regelalgorithmus wird der Regler-basierte Ansatz mit Optimallastwert 50% verwendet.

Deutlich zu erkennen ist, dass der Filter gute Eigenschaften besitzt. Es werden im besten Fall (Differenz = 8%) ca. 2/3 an Aufwand im Vergleich zur Variante ohne Filter eingespart. Die Ergebnisse sind dennoch korrekt. Zu groß darf die Differenz allerdings nicht sein, da sonst die großen Lastschwankungen zu spät signalisiert werden und so eine Überlast entsteht. Dies ist deutlich bei den Differenzen 10% und 15% zu erkennen. Korrekte Ergebnisse liefern die Werte bis 8%. Somit ist für diesen Filter der Differenzwert 8% optimal.

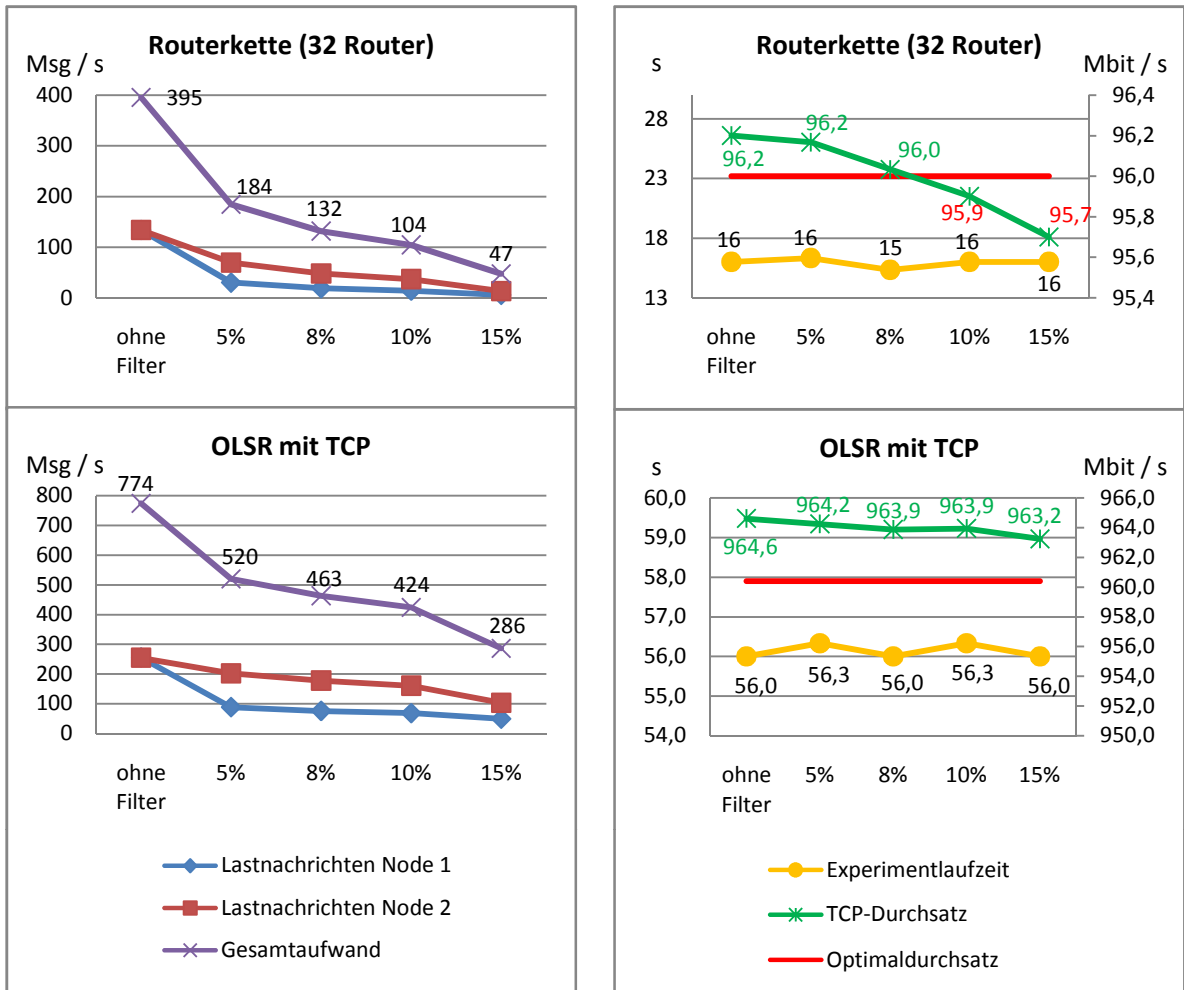


Abbildung 7.14: Vergleich der Filtereigenschaft des Differentiellen Filters bei der Routerkette mit 32 Routern und dem OLSR-Szenario mit TCP-Verbindungen mit verschiedenen Differenzwerten.

### 7.3.8.3 Schwellenwert-basierter Filter

Der schwellenwertbasierte Filter filtert die getriggerten Lastwerte anhand der Änderungen des zugehörigen Status. Er filtert dadurch sehr gut und sorgt hierbei für wesentlich weniger Nachrichten die der Koordinator verarbeiten muss. Abbildung 7.15 zeigt die anfallenden Nachrichten und die Korrektheit der Messergebnisse bei verschiedenen Schwellenwertkonfigurationen. Als Trigger kommt der timergesteuerte Trigger zum Einsatz. Als Adaptionalgorithmus wird der Regler-basierte Ansatz verwendet. Der Optimalwert wird auf 50% gesetzt, da dieser Wert der optimalste ist (siehe 7.3.5 „Festlegung des Optimalwertes beim Regler-basierten Ansatz“).

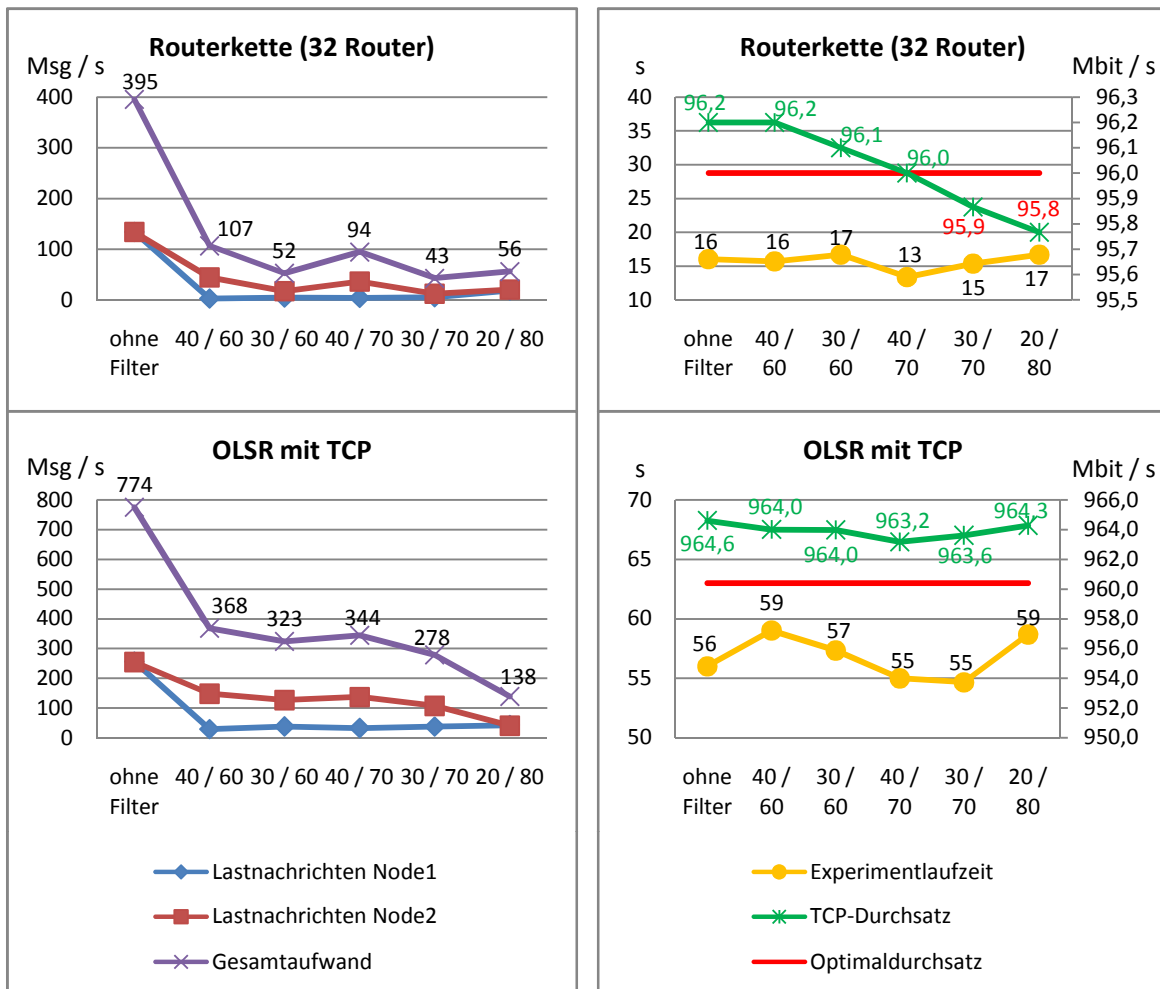


Abbildung 7.15: Vergleich der Filtereigenschaft des schwellenwertbasierten Filters bei der Routerkette mit 32 Routern und dem OLSR-Szenario mit TCP-Verbindungen mit verschiedenen Schwellenwertkonfigurationen.

Man kann deutlich erkennen, dass je weiter die Schwellenwerte von einander entfernt sind um so weniger Anpassungen nötig sind. Dies liegt daran, dass die Last der Knoten mehr im Optimalbereich liegt. Des Weiteren ist zu sehen, dass der Knoten mit wenig Last (Node 1) erst dann viele Nachrichten schicken muss, wenn der Unterlastschwelenwert entsprechend tief liegt. Dabei pendelt die Idle-Last zwischen Unterlastbereich und Optimallastbereich. Umgekehrtes gilt für den Knoten mit der Routerkette / TCP-Verbindung (Node 2). Dieser sendet dann weniger Nachrichten, wenn der Schwellenwert für die Überlast höher liegt. Denn dann überschreitet seine Last den Überlastschwelenwert seltener.

Im Vergleich zur Variante ohne Filter kann man sehen, dass die Eigenschaften dieses Filters ausgesprochen gut sind. Man spart hier bis zu 4/5 an Aufwand im Vergleich zur Variante ohne Filter. Als empfehlenswerte Schwellenwertkonfiguration bieten sich zwei

Einstellungen an. Die Konfiguration 30 / 60 liefert korrekte Ergebnisse und erzeugt wenige Nachrichten. Die Konfiguration 40 / 70 liefert ebenfalls korrekte Ergebnisse, beschleunigt die Experimentlaufzeit, benötigt aber mehr Nachrichten. Somit sollte man für große Systeme die Konfiguration 30 / 60 verwenden, kann aber bei kleinen Systemen, also wenigen Computenodes, die „schnellere“ Konfiguration 40 / 70 verwenden um Zeit zu sparen.

**7.3.8.4 Fazit**

Abbildung 7.16 zeigt den Vergleich der Filtereigenschaften der drei vorgestellten Möglichkeiten. Hierbei wurde beim Differentiellen Filter wie auch beim Schwellenwertbasierten Filter die Konfiguration genommen, die am Besten filtert und dabei dennoch korrekte Ergebnisse liefert.

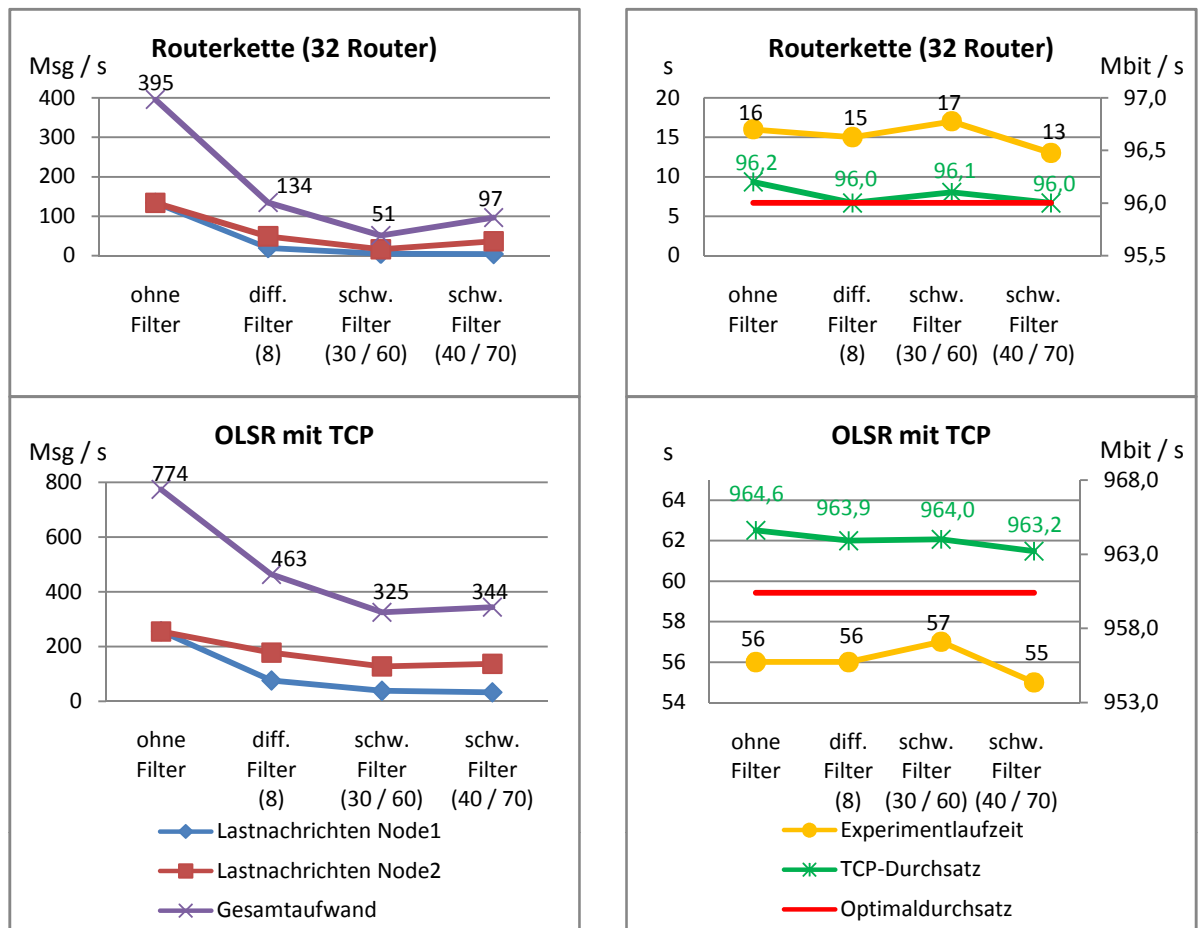


Abbildung 7.16: Vergleich der Filtereigenschaft der 3 vorgestellten Filter bei deren jeweiliger bester Konfiguration.

Zu sehen ist, dass die Variante ohne Filter zwar die korrektesten Ergebnisse liefert, allerdings auch am meisten Aufwand bedeutet. Der Differentielle Filter hingegen benötigt in seiner besten Konfiguration bis zu  $2/3$  weniger Aufwand als die Variante ohne Filter. Der Schwellenwert-basierte Filter ist in beiden Konfigurationen der am Besten filternde. Er bespart bis zu  $4/5$  weniger Aufwand im Vergleich zur Variante ohne Filter. Hierbei kann man zwischen einer Konfiguration für kleine Systeme und einer für größere Systeme wählen. Die Konfiguration (30 / 60) für größere Systeme dauert etwas länger als die Konfiguration (40 / 70) für kleinere Systeme, benötigt aber auch weniger Aufwand.

In dieser Evaluation hat sich somit der Schwellenwert-basierte Filter als die beste Variante herausgestellt. Dies ermöglicht auch die Verwendung beider Adaptionalgorithmen.

### 7.3.9 Festlegung des Überlastschwellenwert beim TCP-basierten Ansatz

Der Unterlast- und Überlast-Schwellenwert des TCP-basierten Ansatzes sind bereits durch den Filter auf einen guten Wert festgelegt worden. Für den TCP-basierten Ansatz benötigen wir jedoch noch einen dritten Schwellenwert. Der durch den Filter festgelegte Überlast-Schwellenwert wird hierbei zum Schwellenwert für potentielle Überlast und ein neuer Schwellenwert kommt für die Überlast hinzu. Abbildung 7.17 zeigt die gesendeten Lastnachrichten und die Anzahl der Adaptionen beim TCP-basierten Ansatz bei verschiedenen Schwellenwertkonfigurationen. Als TDF-Sprünge werden die in Kapitel 7.3.7 beschriebenen TDF-Sprünge gesetzt.

Es ist deutlich zu erkennen, dass für den Überlastschwellenwert der höchste Wert anzusetzen ist. 90% ist bei beiden Konfigurationen der Wert, bei dem verhältnismäßig wenige Nachrichten geschickt werden. Die Konfiguration 40 / 70 / 90 ist in diesem Fall die Beste und sollte bei Verwendung des TCP-basierten Adaptionalgorithmus verwendet werden. Diese Werte liefern korrekte Ergebnisse und eine schnelle Ausführung bei niedrigem Overhead.

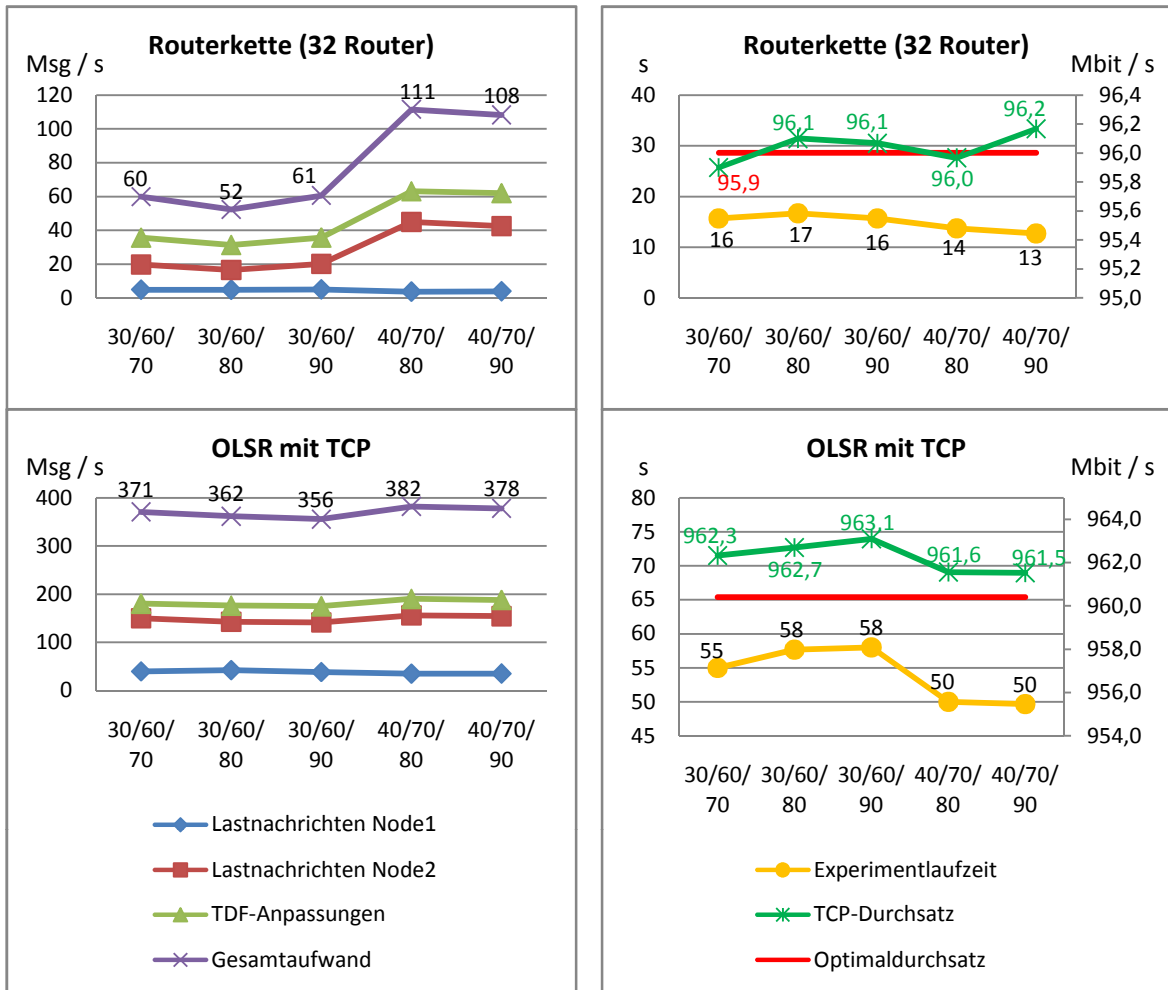


Abbildung 7.17: Vergleich der Adaption des TCP-basierten Ansatzes bei ausgewählten Schwellenwertkonfigurationen.

### 7.3.10 Vergleich der Adaptionskonzepte

In dieser Diplomarbeit wurden zwei alternative Ansätze für die eigentliche Adaption vorgestellt. Diese sollen nun miteinander verglichen werden. Es werden beide Alternativen mit einer Routerkette mit einer Länge von 32 Routern und mit dem OLSR-Szenario mit TCP-Verbindungen untersucht. Dabei werden die bereits vorgestellten Optimalwerte eingestellt. Abbildung 7.18 zeigt die Messergebnisse zum Vergleich der Konzepte. Zudem wurde das OLSR-Szenario mit beiden Adaptionskonzepten evaluiert. Tabelle 7.4 zeigt die gemessenen Werte.

Beim Betrachten der Ergebnisse kann man feststellen, dass beide Adaptionskonzepte relativ gleichwertig sind. Es gibt keine deutlichen Unterschiede. Beide Konzepte liefern

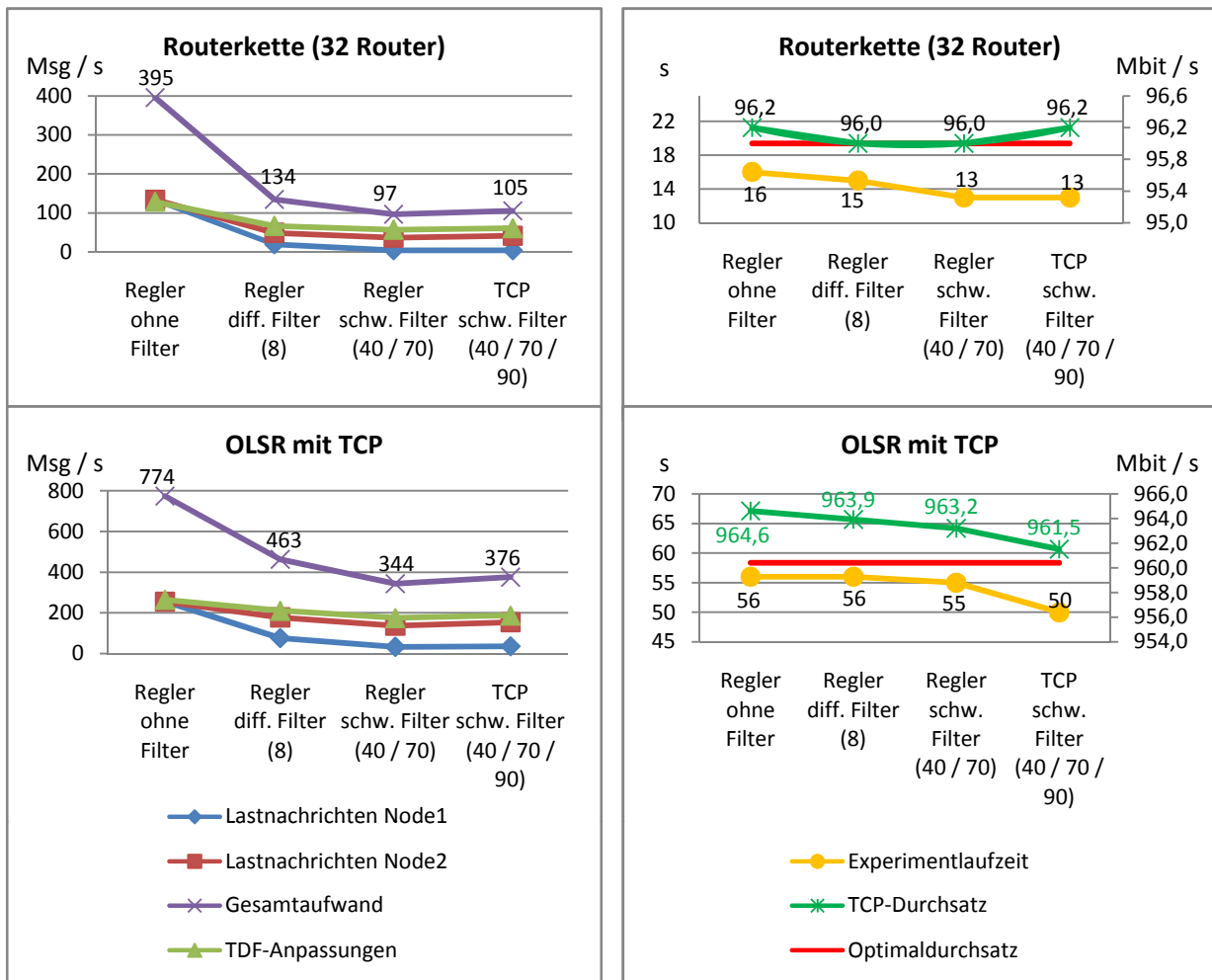


Abbildung 7.18: Vergleich der Adaptionkonzepte.

Adaptionsansatz	gesendete OLSR-Nachrichten
ohne Adaption (konstanter TDF = 0)	10.389
Regler-basierter Ansatz	10.428
TCP-basierter Ansatz	10.499

Tabelle 7.4: Vergleich der Ausführung des OLSR-Szenarios.

korrekte Ergebnisse in ungefähr der selben Zeit mit ungefähr dem selben Aufwand. Es kristallisiert sich hier also kein Ansatz als besser heraus, beide sind einsatzfähig und erfüllen die Anforderungen an die Adaption. Auch beim OLSR-Szenario ist die Anzahl der gemessenen OLSR-Pakete relativ identisch. Kleine Differenzen entstehen durch die unterschiedliche gemessene Laufzeit. Durch Bruchteile von Sekunden kann bereits ein

merklicher Unterschied entstehen.

Aus subjektiver Sicht kann man dem Regler-basierten Ansatz einen kleinen Vorzug geben, da er das flexiblere Konzept ist. Man kann jede hier vorgestellte Filtervariante mit ihm verwenden. Zudem besitzt er nicht so viele Parameter, was die Konfiguration deutlich erleichtert. Tabelle 7.5 zeigt die benötigten Parameter der beiden Konzepte im Vergleich.

	Regler-basierter Ansatz	TCP-basierter Ansatz
Adaptions-Timer	benötigt	benötigt
maximaler Adaptions-Timer	benötigt	benötigt
maximaler TDF-Sprung	-	benötigt
TDF-Sprung bei Potentieller Überlast	-	benötigt
TDF-Sprung bei Unterlast	-	benötigt
optimale Last	benötigt	-
Extremwerte des TDF	(benötigt)	(benötigt)
Filter-Logik	(benötigt)	(-)
Trigger-Timer	benötigt	benötigt
maximaler Trigger-Timer	benötigt	benötigt
Unterlast Schwellenwert	(benötigt)	benötigt
Potentielle Überlast Schwellenwert	-	benötigt
Überlast Schwellenwert	(benötigt)	benötigt
Lastdifferenz im Filter	(benötigt)	-

Tabelle 7.5: Vergleich der benötigten Parameter der Adaptionskonzepte.

Wenn man die Einstellung der Filter vernachlässigt, so muss man beim Regler-basierten Ansatz gerade einmal drei Parameter bestimmen. Beim TCP-basierten Ansatz sind es immerhin schon fünf Parameter. Aufgrund dieses kleinen Unterschiedes wird dem Regler-basierten Adaptionsalgorithmus der Vorzug gegeben, in der Praxis sind aber beide Konzepte als gleichwertig anzusehen.

## 7.4 Zusammenfassung

In dieser Evaluation wurde gezeigt, dass die entwickelten Konzepte allen Anforderungen an die Adaption des TDF genügen und auch korrekt sind. Dabei wurden einzelne Konzepte als unbrauchbar oder überflüssig festgestellt. Diese sind die Initiale Adaption bzw. der eingeschränkte Reglerbereich, welche/r nicht notwendig sind und somit nur bremsend auf die Adaption einwirken sowie der eventgesteuerte Trigger, der sich als potentielle Fehlerquelle und somit nicht praxistauglich herausstellte. Desweiteren wurden optimale Werte für die Parameter der einzelnen Konzepte sowie die optimale Kombination der einzelnen Konzepte herausgefunden. Es wurde zudem festgestellt, dass beide aufgezeigten

Adaptionskonzepte gleichwertig sind und somit beide in der Praxis angewandt werden können.

Tabelle 7.6 zeigt die optimale Konfiguration des Regler-basierten Ansatzes mit dem Schwellenwert-basierten Filter.

Parameter	Wert
Adaptions-Timer	3
maximaler Adaptions-Timer	768
optimale Last	50
Extremwerte des TDF	(-400 / 1000)
Trigger-Timer	2
maximaler Trigger-Timer	512
Unterlast Schwellenwert	40
Überlast Schwellenwert	70

Tabelle 7.6: optimale Konfiguration des Regler-basierten Adaptionsansatzes mit Schwellenwert-basiertem Filter.

Tabelle 7.7 zeigt die optimale Konfiguration des TCP-basierten Ansatzes mit dem Schwellenwert-basierten Filter.

Parameter	Wert
Adaptions-Timer	3
maximaler Adaptions-Timer	768
TDF-Sprung bei Unterlast	66
TDF-Sprung bei potentieller Überlast	68
TDF-Sprung bei Überlast	93
Extremwerte des TDF	(-400 / 1000)
Trigger-Timer	2
maximaler Trigger-Timer	512
Schwellenwert für Unterlast	40
Schwellenwert für potentielle Überlast	70
Schwellenwert für Überlast	90

Tabelle 7.7: optimale Konfiguration des Regler-basierten Adaptionsansatzes mit Schwellenwert-basiertem Filter.



# Zusammenfassung und Ausblick

---

## 8.1 Zusammenfassung

In dieser Diplomarbeit wurde der TDF-Adapter für die TVEE entwickelt. Hierzu wurde im Kapitel 2 auf das Projekt TVEE eingegangen und die Anforderungen an eine erfolgreiche Adaption herausgearbeitet und vorgestellt.

Im anschließenden Kapitel 3 „Related Work“ wurden in angrenzenden Fachgebieten Möglichkeiten und Ideen zum Lösen der gestellten Aufgabe gesucht. Dabei wurden aus dem Bereich der Regelungstechnik und aus dem Anwendungsfall des TCP-Protokolls gute und brauchbare Ansätze gefunden und erarbeitet.

Im Kapitel 4 „Entwurfskriterien“ wurden einige Probleme bei der Aufgabenstellung dieser Diplomarbeit erläutert und wichtige Überlegungen zur Adaption angestellt und aufgeführt.

Im darauf folgenden Kapitel 5 „Architektur des TDF-Adapters“ wurde zunächst ein allgemeiner Entwurf des verteilten TDF-Adapters entwickelt bevor Konzepte für die einzelnen Module im Detail erstellt wurden. Hierbei entwickelte man Konzepte für das Relevanz-Modul, das die Aufgabe hat, die Anzahl der für die Adaption so wichtigen Lastsignalisierungsnachrichten zu minimieren, für die Logik im Global-State-Modul das aus den gesendeten Lastinformationen den Global-State bestimmt und für die Adaptionskonzepte, die im Adaptions-Modul für die erfolgreiche Adaption des TDF sorgen.

Hierbei entstanden viele verschiedene Varianten und Optionen die im Kapitel 6 „Implementierung“ in realen Code umgesetzt wurden. Es wurde auch detailliert herausgestellt welche Parameter für eine optimale Adaption benötigt werden.

In der anschließenden Evaluation (Kapitel 7) wurden zunächst drei Szenarien vorgestellt und die Evaluationskriterien beschrieben, anhand derer die Bewertung während der Evaluation stattgefunden hat.

Diese sind an erster Stelle die Korrektheit der gemessenen Ergebnisse, dicht gefolgt vom Aufwand der durch die Adaption entsteht und der Experimentlaufzeit die durch die Adaption beeinflusst wird. In der dann erfolgten Evaluation wurden alle Konzepte getestet und

bei Verfügbarkeit verschiedener Varianten miteinander verglichen. Dabei wurden Konzepte als nicht sinnvoll befunden, andere hingegen als unerlässlich, wie z.B. die Berücksichtigung des aktuellen Zustandes. Während der Evaluation wurden für alle im Entwurf und der Implementierung vorgestellten Parameter gute und sinnvolle Werte herausgefunden, so dass nun eine Konfiguration zur Verfügung steht, die bei allen drei vorgestellten Evaluierungsszenarien korrekte Ergebnisse bei der schnellstmöglichen Ausführung mit wenig Overhead ermöglichen. Zudem konnte herausgefunden werden, dass im Prinzip beide Adaptionkonzepte gleichwertig sind und somit beide zum Einsatz kommen können. Hierbei sei jedoch angemerkt, dass dem Regler-basierten Ansatz durch seine vereinfachte Konfiguration und die Flexibilität bei der Auswahl der Filter ein leichter Vorzug eingeräumt werden kann, obwohl beide technisch nahezu dieselben Ergebnisse produzieren. Die Aufgabe dieser Diplomarbeit wurde somit erfolgreich umgesetzt und alle Anforderungen an eine Diplomarbeit und auch an die Adaption des TDF erfüllt.

### 8.2 Ausblick

Bei der Evaluation zu Tage gekommene neue Fakten, konnten in diese Arbeit nicht mehr einfließen und müssen zu einem späteren Zeitpunkt bearbeitet und gelöst werden.

Bisher wurden noch keine Tests an größeren Systemen durchgeführt. Man sollte die Adaption mit Szenarien testen, die mehr als zwei Computenodes benötigen. Hierbei kann man gleich ein reales Szenario, wie in der Einführung beschrieben, verwenden. Es könnten wirklich 1.000 Knoten ein Protokoll fahren, so dass die Adaption an einem realen großen Szenario mit vielen Computenodes getestet wird ([24] [20]).

Ein Punkt, der bisher noch nicht untersucht wurde, ist das Synchronisationsproblem, auf das bei den TDF-Sprüngen hingewiesen wurde. Bei jedem Epochenwechsel laufen die einzelnen Computenodes minimal auseinander. Hier gilt es zu untersuchen inwieweit sich die hier entstandenen TDF-Sprünge als problematisch herausstellen. Evtl. können Synchronisationsprotokolle entwickelt werden, die dieses Problem beheben.

Ein weiterer Punkt, der in zukünftiger Arbeit untersucht werden kann, ist die Lösung des Problems der Ausführung eines Experiments schneller als Echtzeit. Hierbei gibt es aktuell die Problematik, dass die Überwachung der Last und die darauffolgende Reaktion nur in Echtzeit ablaufen kann. Wenn die Virtuelle Zeit wesentlich schneller läuft, kann die Überwachung und Adaption zu langsam sein, so dass es beim Experiment zu Überlast kommen kann. In dieser Diplomarbeit wurde die Adaption zwar mit negativem TDF getestet, allerdings wurde diese Problematik nicht tiefgründig genug bearbeitet, da die Adaption bei positivem TDF vorrangig war.

Generell muss man die Adaption bei weitaus größeren und anders gearteten Szenarien testen, um eine allgemeine Adaptionlösung für eine Vielzahl von Anwendungen bereit zu stellen. Die hier betrachteten Szenarien sind zwar charakteristisch für viele Anwendungen, sichern aber noch keine Allgemeingültigkeit zu.

# Abbildungsverzeichnis

---

2.1	Architektur des TVEE - vNodes mittels Virtual Routing innerhalb einer Virtuellen Maschine . . . . .	14
2.2	Zusammenspiel der Komponenten der TVEE . . . . .	16
2.3	Regelkreis des TVEE - Regelkreis, der während der gesamten Experimentdauer immer wieder durchlaufen wird . . . . .	17
2.4	64 Knoten Cluster der Univerität Stuttgart auf dem der Prototyp der TVEE läuft ( <a href="http://net.informatik.uni-stuttgart.de/">http://net.informatik.uni-stuttgart.de/</a> ) . . . . .	18
3.1	TCP - Flusskontrolle . . . . .	22
3.2	Blockschaltbild eines Standardregelkreises . . . . .	23
3.3	Klassifizierung der verschiedenen Reglerarten . . . . .	24
3.4	Beispiel eines Temperaturverlaufes eines Kühlschranks mit zweipunktgeregeltem Kühlaggregat . . . . .	25
4.1	Übersicht über die Lastverteilung eines Prozessors während einiger Takte des Schedulers. . . . .	30
4.2	Übersicht über die Lastverteilung eines Prozessors während einiger Millisekunden . . . . .	30
5.1	Architektur des TDF-Adapters . . . . .	38
5.2	Architektur des Relevanz-Moduls . . . . .	40
5.3	Übersicht über den eingeteilten Lastbereich beim TCP-basierten Ansatz . . . . .	45
5.4	Übersicht über den Lastbereich beim Regler-basierten Ansatz . . . . .	47
7.1	Anordnung der Knoten beim OLSR-Szenario . . . . .	72
7.2	Beispiel für einen eher wechselhaften Lastverlauf wie er beim OLSR-Szenario entsteht . . . . .	72
7.3	Anordnung der Routerkette und der Knoten beim Routerketten-Szenario . . . . .	74
7.4	Beispiel für einen eher glatten Lastverlauf wie er bei der Routerkette entsteht . . . . .	74
7.5	maximal erreichbarer TCP-Durchsatz des Routerketten-Szenarios bei unterschiedlich langen Routerketten. . . . .	75
7.6	Anordnung der Knoten beim OLSR-Szenario in Kombination mit den parallel verlaufenden TCP-Verbindungen . . . . .	76

7.7	Minimaler TDF im Vergleich . . . . .	78
7.8	Deadlock des eventgesteuerten Triggers . . . . .	80
7.9	Vergleich der Korrektheit der ermittelten Messergebnisse bei verschiedenen Timer-Intervallen des Triggers. . . . .	81
7.10	Verlauf des Timerintervalls bei der Berücksichtigung des aktuellen Zustandes. . . . .	81
7.11	Vergleich der Ausführung von Routerketten verschiedener Länge mit Berücksichtigung des aktuellen Zustandes und ohne Berücksichtigung des aktuellen Zustandes. . . . .	82
7.12	Vergleich der Adaption mit Berücksichtigung des aktuellen Zustandes und ohne Berücksichtigung des aktuellen Zustandes. . . . .	83
7.13	Vergleich der Korrektheit der Messergebnisse und der Experimentlaufzeit bei verschiedenen Optimallastwerten des Regler-basierten Ansatzes. . . . .	84
7.14	Vergleich der Filtereigenschaft des Differentiellen Filters bei der Router- kette mit 32 Routern und dem OLSR-Szenario mit TCP-Verbindungen mit verschiedenen Differenzwerten. . . . .	88
7.15	Vergleich der Filtereigenschaft des schwellenwertbasierten Filters bei der Routerkette mit 32 Routern und dem OLSR-Szenario mit TCP-Verbindungen mit verschiedenen Schwellenwertkonfigurationen. . . . .	89
7.16	Vergleich der Filtereigenschaft der 3 vorgestellten Filter bei deren jeweiliger bester Konfiguration. . . . .	90
7.17	Vergleich der Adaption des TCP-basierten Ansatzes bei ausgewählten Schwellen- wertkonfigurationen. . . . .	92
7.18	Vergleich der Adaptionskonzepte. . . . .	93

# Tabellenverzeichnis

---

7.1	Vergleich der Ausführung einer Routerkette mit 32 Routern mit und ohne differenzielle Regelung . . . . .	85
7.2	Sinnvolle TDF-Sprünge des TCP-basierten Ansatzes bei ausgewählten Schwellenwertkonfigurationen . . . . .	86
7.3	Lastnachrichten und TDF-Anpassungen ohne Filter . . . . .	87
7.4	Vergleich der Ausführung des OLSR-Szenarios. . . . .	93
7.5	Vergleich der benötigten Parameter der Adaptionskonzepte. . . . .	94
7.6	optimale Konfiguration des Regler-basierten Adaptionsansatzes mit Schwellenwert-basiertem Filter. . . . .	95
7.7	optimale Konfiguration des Regler-basierten Adaptionsansatzes mit Schwellenwert-basiertem Filter. . . . .	95



# Verzeichnis der Listings

---

6.1	Algorithmus für den timergesteuerten Trigger . . . . .	50
6.2	Algorithmus für den timergesteuerten Trigger mit Berücksichtigung des aktuellen Zustandes . . . . .	51
6.3	Algorithmus für den eventgesteuerten Trigger . . . . .	52
6.4	Algorithmus für den eventgesteuerten Trigger mit Berücksichtigung des aktuellen Zustandes . . . . .	53
6.5	Algorithmus für den Differentiellen Filter . . . . .	54
6.6	Algorithmus für den Schwellenwert-basierten Filter . . . . .	55
6.7	Algorithmus für die Aktualisierung der Knotenlast im Global-State-Modul .	56
6.8	Algorithmus für die Bildung des Global-State im Global-State-Modul . . . .	56
6.9	Initialisierungsalgorithmus beim TCP-basierten Ansatz . . . . .	58
6.10	Algorithmus zur Unterlastbehandlung beim TCP-basierten Ansatz . . . . .	59
6.11	Algorithmus zum Feintuning im Optimallastbereiches beim TCP-basierten Ansatz . . . . .	60
6.12	Algorithmus zur potentiellen Überlastbehandlung beim TCP-basierten Ansatz	61
6.13	Algorithmus zur Überlastbehandlung beim TCP-basierten Ansatz . . . . .	61
6.14	Basisalgorithmus des TCP-basierten Ansatzes . . . . .	62
6.15	Initialisierungsalgorithmus beim Regler-basierten Ansatz . . . . .	64
6.16	Algorithmus zur Unterlastbehandlung beim Regler-basierten Ansatz . . . . .	66
6.17	Algorithmus zum Feintuning im Optimallastbereiches beim Regler-basierten Ansatz . . . . .	67
6.18	Algorithmus zur Überlastbehandlung beim Regler-basierten Ansatz . . . . .	68
6.19	Basisalgorithmus des Regler-basierten Ansatzes . . . . .	68



# Literaturverzeichnis

---

- [1] <http://openVZ.org/>.
- [2] <http://www.netperf.org/>.
- [3] <http://www.olsr.org/>.
- [4] Optimized link state routing protocol (olsr), 2003.
- [5] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP congestion control, April 1999.
- [6] Frank Meyer Andrej Radonic. *Xen 3*. Franzis' Verlag, 2006.
- [7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Adrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, USA, October 2003. ACM.
- [8] Daniel J. Barrett. *Linux kurz & gut*. O'Reilly, 2004.
- [9] Timo Benk, Henning Spranga, Jaroslaw Zdrzalek, and Ralph Dehner. *Xen – Virtualisierung unter Linux*. Open Source Press, 2007.
- [10] Peter Busch. *Elementare Regelungstechnik: allgemeingültige Darstellung ohne höhere Mathematik*. Vogel, 2002.
- [11] Alexander Egorenkov. Protocol for epoch switching in a distributed time virtualized emulation environment. Master's thesis, Universität Stuttgart, Studiengang Softwaretechnik, 2008.
- [12] Stephen Figgins Ellen Siever, Stephen Spainhour. *Linux in a Nutshell*. O'Reilly, 2005.
- [13] Otto Föllinger. *Regelungstechnik*. Hüthig Verlag.
- [14] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley and Sons, 2000.
- [15] Andreas Grau, Steffen Maier, Klaus Herrmann, and Kurt Rothermel. Time jails: A hybrid approach to scalable network emulation. In *In Proceedings of the 22nd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2008)*. IEEE Computer Society; ACM, 2008.

- [16] Diwaker Gupta, Ken Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To infinity and beyond: Time-warped network emulation. In *NSDI*. USENIX, 2006.
- [17] P. Jacquet, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot. Optimized link state routing protocol for ad hoc networks, December 14 2001.
- [18] Steffen Maier, Andreas Grau, Harald Weinschrott, and Kurt Rothermel. Scalable network emulation: A comparison of virtual routing and virtual machines. In *ISCC*, pages 395–402. IEEE, 2007.
- [19] Jon Postel. Transmission control protocol. RFC 793, ISI, September 1981.
- [20] Y. Rekhter and T. Li. RFC 1771: A Border Gateway Protocol 4 (BGP-4), March 1995. Obsoletes RFC1654 [?]. Status: DRAFT STANDARD.
- [21] George F. Riley, Richard Fujimoto, and Mostafa H. Ammar. A generic framework for parallelization of network simulations. In *MASCOTS*, page 128. IEEE Computer Society, 1999.
- [22] David Sellers. An overview of proportional plus integral plus derivative control and suggestions for its successful application and implementation. Technical report, Portland Energy Conservation Inc.
- [23] Rob Simmonds and Brian Unger. Towards scalable network emulation. *Computer Communications*, 26(3):264–277, 2003.
- [24] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.
- [25] A. S. Tannenbaum. *Computer Networks*. Prentice Hall, 2003.

Alle URLs wurden zuletzt am 17.12.2008 geprüft.

## **Erklärung**

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

---

(Frederik Pakai)