RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Hendrik Hillesheimer

Ein skalierendes und selbstkonfigurierendes Monitoring System auf Linux-Clustern

Diplomarbeit

HD-KIP-03-09

KIRCHHOFF-INSTITUT FÜR PHYSIK

Fakultät für Physik und Astronomie
Ruprecht-Karls-Universität Heidelberg

Diplomarbeit
im Studiengang Physik
vorgelegt von
Hendrik Hillesheimer
aus Lörrach
2002

Ein skalierendes und selbstkonfigurierendes Monitoring System auf Linux-Clustern

Die Diplomarbeit wurde von Hendrik Hillesheimer **ausgeführt am**
Kirchhoff-Institut für Physik
unter Betreuung von
Herrn Prof. Dr. Volker Lindenstruth

Ein skalierendes und selbstkonfigurierendes Monitoring System auf Linux-Clustern

Durch Vernetzung einzelner Rechner zu einem Verbund kann deren Rechenleistung gebündelt werden, um zeitkritische, rechenintensive Anwendungen parallel auszuführen. Wichtig ist dabei eine Überwachung kritischer Parameter, um Fehlfunktionen zu erkennen und rechtzeitig adäquate Gegenmaßnahmen ergreifen zu können. Auch eine automatisierte Lastverteilung ist nur mit Kenntnis wichtiger Parameter aller Rechner des Verbunds möglich.

In dieser Diplomarbeit wird ein Programmpaket entwickelt, welches Systeminformationen einzelner beteiligter Rechner entgegennimmt und an geeigneten Punkten sammelt. Auf diese Weise kann jederzeit ein Bild über den Zustand des gesamten vernetzten Systems gewonnen werden.

Beliebige Überwachungseinheiten auf den einzelnen Rechnern sind einfach integrierbar und werden dynamisch verwaltet. Die Architektur des Überwachungssystems ist derart, daß es auch bei großen Verbunden skaliert. Selbst im Falle des Ausfalls einzelner Einheiten bleibt die Funktion des gesamten Systems durch eine automatische Rekonfiguration erhalten.

A scalable and self-configurating Cluster Monitoring System with the OS Linux

Much computational power can be achieved when linking computers via a fast network. For this purpose, it is important to measure critical parameters to avoid system crashes. Also load balancing is only possible with the knowledge of some important system parameters.

In this diploma work, programs are developped that can get locally measured system parameters. This offers the possibility to identify the state of the complete connected system.

Any parameters can be measured and integrated dynamically in the developped software. The architecture of the system also scales on large clusters. The functionality of the software is not influenced by system failures of a part of the cluster due to its automatic reconfiguration abilities.

Inhaltsverzeichnis

I Einleitung **6**

II Zielsetzungen und Motivation für die Entwicklung eines Cluster Monitoring Systems **8**

1 Motivation der Arbeit und wichtige Begriffe **8**
- 1.1 Definition eines Cluster Monitoring System 8
- 1.2 Notwendigkeit von Monitoring 9
  - 1.2.1 Monitoring auf lokaler Ebene 10
  - 1.2.2 Monitoring auf Cluster-Ebene 11

2 Cluster Monitoring: Zielsetzungen und Bestandsaufnahmen **11**
- 2.1 Anforderungen an ein Cluster Monitoring System 12
- 2.2 Existierende Programme 14
  - 2.2.1 PAM 14
  - 2.2.2 NGOP 15
  - 2.2.3 Ganglia 16
  - 2.2.4 Supermon 16
  - 2.2.5 VACM 18

3 Zusammenfassung **18**

III Datenbanken **20**

4 Datenbanken und Cluster Monitoring **20**
- 4.1 Definition einer Datenbank 20
- 4.2 Client-Server-Prinzip 21
- 4.3 Integration einer Datenbank in ein Monitoring System 21
- 4.4 Structured Query Language: SQL 21

5 Leistungsfähigkeit von Datenbanken beim Einfügen von Datensätzen **22**
- 5.1 Getestete Datenbanken 23
  - 5.1.1 PostgreSQL 23
  - 5.1.2 MySQL 24
- 5.2 Das Benchmark-Programm 24
- 5.3 Ergebnisse 28

5.4 Interpretation 37
5.5 Optimierungsmöglichkeiten 39

6 Zusammenfassung **40**

IV Grundlegende Konzepte des Monitoring Systems 42

7 Die Architektur des Cluster Monitoring System HCMS **42**
7.1 Hierarchisches Konzept 42
7.2 Datenübermittlung zwischen den Hierarchieebenen 45
7.2.1 XML 45
7.2.2 Sensordaten und XML 47
7.3 Schnittstellen 48
7.3.1 Sensorschnittstellen 48
7.3.2 Schnittstellen zu Datenbanken 49

8 Zusammenfassung **49**

V Die Implementierung des HCMS **52**

9 Allgemeines **52**
9.1 C, C++ und Linux-Systemprogrammierung 52
9.2 TCP/IP und BSD-Sockets 53
9.3 E/A-Multiplexing mit SELECT 54
9.4 Der XML-Parser TinyXML 54
9.5 Datenkompression mit der Bibliothek ZLIB 55
9.6 Datentransfer-Protokoll 56
9.7 Ermittlung der Serverhierarchie: die getserver()-Funktion . . . 57

10 Clientebene: der Sensormanager **59**
10.1 Einlesen der Konfiguration 59
10.2 Kommunikation der Sensoren mit dem Sensormanager 60
10.2.1 Dynamische Verwaltung der Sensoren 60
10.2.2 Schnittstelle zwischen Sensoren und Sensormanager . . 61
10.2.3 Aufbereitung der Datensätze 61
10.3 Konsolidierung der Daten 62
10.4 Eskalierung der Daten 63
10.5 Rekonfiguration 63

11 Serverebene: der Consolidator/Escalator **65**
11.1 Kommunikationsschnittstellen 65
11.2 Sammeln der Daten 65
11.3 Konsolidierung der Daten 66
11.4 Rekonfiguration und intelligente Datenweiterverarbeitung . . . 67
11.4.1 Eskalierung in nächste Hierarchieebene 68
11.4.2 Speicherung in lokaler Datenbank 68

12 Zusammenfassung **68**

VI Ausblick **71**

13 Bekannte Schwächen des HCMS **71**

14 Fehlende Funktionalität **73**

VII Zusammenfassung **74**

VIII Die CD **76**

IX Danksagung **79**

Teil I
Einleitung

Die Leistungsfähigkeit handelsüblicher Personal Computer nimmt stetig zu. Sie hat mittlerweile Dimensionen erreicht, die eine Konkurrenz zu teuren und oft auch weniger flexiblen Supercomputern möglich machen. Zur ernsthaften Konkurrenz werden Standardrechner aber erst im Verbund miteinander, sei es über lokale Netzwerke oder auch über das Internet. Vorrausetzung dabei ist allerdings, daß das zu lösende Problem gut parallelisiert abgearbeitet werden kann und der nötige Informationsaustausch Bandbreitenlimits, die durch die Architektur des verwendeten Netzwerks vorgegeben sind, nicht übersteigt. Im Falle lokal vernetzter Rechner spricht man auch von Clustern. Sind mehrere Cluster über das Internet miteinander im Verbund, wird der Begriff Grid verwendet.

Cluster und Grids haben jedoch einen schwerwiegenden Nachteil gegenüber Supercomputern: da sie nicht eine Einheit darstellen, sind sie deutlich schwerer in ihrer Gesamtheit zu Überwachen. Eine Überprüfung mancher Parameter des Systems ist aber unabdingbar, um zu gewährleisten, daß das System stabil läuft oder aber Anwendungen die notwendigen Ressourcen zur Verfügung haben. Im Fehlerfalle können zudem nur bei Kenntnis der Ursachen geeignete Maßnahmen eingeleitet werden.

Ein Programmpaket, welches ein Clusters oder ein Grid überwacht (im Folgenden wird hierfür der Begriff Cluster Monitoring verwendet) sind vor eine schwierige Aufgabe mit hohen Anforderungen gestellt. Es existieren nur wenige Programme, die diese Aufgabe erfüllen können, und diese haben oft nur einen eingeschränkten Funktionsumfang oder sind mit implementierungsspezifischen Problemen behaftet, die ihre Betriebsfähigkeit stark einschränken. Noch schwerer wiegt aber, daß bei vielen architekturelle Gründe verhindern, daß Monitoring auch bei einem großen Rechnerverbund möglich ist: die anfallenden Daten werden zentral zu einem Rechner geschickt, so daß die Leistungsfähigkeit des System durch diesen Rechner oder aber die Netzwerkbandbreite zu ihm beschränkt ist. Zudem verhält sich ein solches Design kritisch bei dem Ausfall des entsprechenden Rechners (single point of failure). Ein weiterer anzubringender Kritikpunkt ist die fehlende Flexibilität vieler Programme: erwünscht ist die Fähigkeit zur dynamischen Anbindung beliebiger Überwachungseinheiten auf jedem Knoten des Clusters statt deren Verbindung in statischer Form, da es im Prinzip beliebig viele mögliche Parameter gibt und je nach Anwendung die wichtigen Parameter des Gesamtclusters oder auch einzelner Rechner unterschiedlich sein können.

Aufgabe dieser Diplomarbeit war es also, ein Cluster Monitoring System mit dem Betriebssystem Linux zu entwickeln, wobei ein besonderes Augenmerk auf folgende Punkte gelegt wurde: es soll beliebig skalierbar, also auch auf großen Clustern verwendbar sein. Die Daten werden bestenfalls zentral abgelegt. Da dies nicht skalieren kann, muß das System aber bei einer starken Auslastung zu einer dezentralen Form der Datenspeicherung übergehen. Der Benutzer benötigt eine einfache Schnittstelle, um auf beliebige Daten zugreifen zu können. Das System soll einfach zu konfigurieren sein und bei Fehlfunktionen beteiligter Clusterknoten sich selbst rekonfigurieren können. Dabei muß auch gewährleistet sein, daß möglichst wenige Informationen verloren gehen. Die Datenübertragung soll in einem einfachen, gut lesbaren und verständlichen Format stattfinden. Um die Netzwerklast zu verringern, ist eine Komprimierung der Daten erwünscht. Auf den einzelnen Knoten, die von Sensoren überwacht werden, welche von dem Benutzer beliebig definiert und implementiert werden können, braucht man außerdem eine klar definierte Schnittstelle, um das Einlesen der Daten in das Monitoring System zu ermöglichen.

Die vorliegende Diplomarbeit gliedert sich wie folgt:

Teil II geht auf die Motivation für die Entwicklung eines Cluster Monitoring Systems ein, definiert Anforderungen und untersucht, inwieweit existierende Systeme diesen Anforderungen gerecht werden.

Teil III beschäftigt sich mit der Datenspeicherung der gewonnenen Systeminformationen. Aus Gründen der Notwendigkeit einer zentralen Verfügbarkeit der Datensätze für den Anwender wurden hierfür Datenbanken verwendet. Diese stellen einen kritischen Bestandteil des Monitoring Systems dar, was seine Skalierbarkeit betrifft. Folglich waren ausführliche Messungen notwendig, um die Leistungsfähigkeit der verwendeten Datenbanken zu ermitteln.

Teil IV zeigt die Entwicklung der Architektur des entwickelten Cluster Monitoring Systems auf. Diese wurde im Hinblick auf zuvor gestellte Anforderungen entworfen.

In Teil V wird die eigentliche Implementierung des Monitoring Systems erläutert. Dabei stehen nicht Details im Vordergrund, sondern es wird ein Überblick über die grundlegende Funktionsweise der Programme gegeben. Nur wichtige Details sind dabei ausführlicher beschrieben.

In Teil VI wird auf notwendige Verbesserungen und Schwächen des Programmpakets eingegangen.

Teil VII faßt die wichtigen Ergebnisse dieser Arbeit zusammen und dient als deren Abschluß.

In Teil VIII gibt Hinweise zur Benutzung der beigelegten CD.

Teil II
Zielsetzungen und Motivation für die Entwicklung eines Cluster Monitoring Systems

In diesem Teil werden die Zielsetzungen und die Motivation dieser Diplomarbeit erläutert. Hierbei ist die Definition zentraler Begriffe notwendig. Im Anschluß daran werden die grundlegenden Konzepte sowie generelle Anforderungen an dieses vorgestellt. Mögliche Architekturen werden anhand existierender Programme erklärt. Zugleich werden deren Unzulänglichkeiten in Bezug auf die existierenden Anforderungen beleuchtet.

1 Motivation der Arbeit und wichtige Begriffe

Kap. 1.1 definiert die zentralen Begriffe, die in dieser Diplomarbeit immer wieder Verwendung finden. Besonders wichtig ist dabei die Definition von Cluster Monitoring und eines Cluster Monitoring Systems. Warum Cluster Monitoring von so großer Bedeutung ist, zeigt Kap. 1.2.

1.1 Definition eines Cluster Monitoring System

Eine Ansammlung von Rechnern, die jeweils eindeutig durch ihren Namen und ihre Netzwerkadressen charakterisiert sind, bezeichnet man als **Cluster**, falls alle Rechner einen gemeinsamen Nutzen haben und somit einem Ziel dienen. Ein Cluster sollte soweit als möglich nach außen hin als eine Einheit auftreten (Single System View). Auf diese Weise kann eine leichte Administrierbarkeit sowie allgemein einfach geartete Schnittstellen zum Benutzer gewährleistet werden. Dies ist notwendig, um in akzeptabler Zeit die verfügbaren Ressourcen des Clusters, wie Speicherplatz und aggregierte Prozessorleistung, konstruktiv nutzen zu können, ohne daß sich der Anwender um Details wie Vernetzung, Verteilung der Prozesse[1] auf die Clusterknoten

[1] Ein **Prozeß** ist eine vom dem Betriebsystem verwaltete Einheit, innerhalb derer ein Programm ausgeführt wird. Ein Programm kann mehrfach in verschiedenen Prozessen ausgeführt werden: kommunizieren diese nicht miteinander, sind die Prozesse unabhängig voneinander.

(load balancing), Kommunikationsmechanismen usw. zu kümmern braucht. In diesem Zusammenhang könnte man auch von einem "Black-Box"-Prinzip sprechen: der Anwender teilt dem Cluster über einfache Schnittstellen mit, welches Problem er gelöst haben will. Über ähnlich geartete Schnittstellen liefert der Cluster seine Ergebnisse zurück. Wie die Arbeit im Detail erledigt wird, braucht der Benutzer dabei nicht zu wissen: die Aufteilung der Arbeit und Delegierung an verschiedene Knoten, das Abfangen von auftretenden Fehlern und das Zusammenführen von Teilergebnissen zu einem Ganzen wird durch auf dem Cluster laufende Programme eigenständig erledigt. Ist dies erreicht, kann der Cluster zum Beispiel als Ersatz für ein leistungsfähiges, aber teures Multiprozessorsystem dienen. Einen einzelnen Rechner innerhalb des Clusters bezeichnet man auch als **Knoten**. Er stellt eine alleine funktionierende Einheit des Gesamtclusters dar, soll sich diesem aber unterordnen, um gemeinsam mit den anderen Knoten als Multiprozessorsystem zu fungieren oder aber die Arbeit defekter Einheiten übernehmen zu können.

Jeder Computer hat eine Vielzahl sich zeitlich verändernder Parameter, die seinen Zustand charakterisieren, wie z.B. freier Speicherplatz, Anzahl der laufenden Prozesse und die Prozessorlast. Werden diese Daten abgefragt und gesammelt, spricht man von **Monitoring**. Ein Programm, welches diese Parameter registriert und dem Anwender auf zunächst nicht weiter spezifizierte Weise verfügbar macht, heißt auch **Monitoring System.** Ein Monitoring System verfügt über eine Anzahl von **Sensoren**, die dazu dienen, jeweils einen bestimmten Parameter auf jeweils einem Knoten zu bestimmen und dem Monitoring System zu übergeben. Bei einem **Cluster Monitoring System** werden die Sensordaten nicht nur eines einzelnen Rechners, sondern eines ganzen Clusters gesammelt und dem Anwender zur Verfügung gestellt.

1.2 Notwendigkeit von Monitoring

Wesentlich für die Motivation dieser Arbeit war, daß Monitoring und speziell ein Cluster Monitoring System eine hilfreiche oder sogar notwendige Funktionalität für eine Vielzahl von Anwendungen bereitstellt. Wie auch immer mit den von Monitoring Sensoren gewonnenen Daten verfahren wird, festzustellen bleibt, daß die eigentliche Aufgabe eines Monitoring System nur das Sammeln von Daten sowie die Verfügbarkeit dieser für den Anwender oder weiter Programme ist. Dabei muß dürfen möglichst keine Informationen verloren gehen.

1.2.1 Monitoring auf lokaler Ebene

Ein wichtiger Punkt, der für ein Monitoring spricht, ist das frühzeitige Erkennen von Problemen, welche die Stabilität des Betriebssystems gefährden können oder aber dafür Verantwortung tragen, daß die von laufenden Programmen benötigten Ressourcen nicht vorhanden sind. Die gewonnenen Informationen können durch das Monitoring System oder aber eine nachgeschaltete Instanz weiterverarbeitet werden. Folgende Punkte zeigen, wofür Monitoring hilfreich sein kann:

- **Systemzustand:** Als Beispiel hierfür kann das ps-Kommando bei UNIX-Systemen oder aber der Taskmanager bei Windows dienen. Diese zeigen die laufenden Prozesse mit ihren spezifischen Attributen an. Der Anwender bekommt einen unmittelbaren Einblick, welche Programme verwendet werden, in welchem Zustand sich die Prozesse befinden und welchem Benutzer der Prozeß gehört. Als Reaktion kann der Administrator Signale an die Prozesse schicken, um ihr Verhalten zu steuern.

  Weitere wichtige Systemparameter sind beispielsweise die Auslastung des Prozessors oder seine Temperatur.

- **Ausnahmebehandlung:** Beendet sich ein Programm in unvorhergesehener Weise oder schickt eine Anwendung bestimmte Fehlernachrichten, so können geeignete Maßnahmen eingeleitet werden, um das Problem zu beheben. So kann das Programm neu gestartet oder eine Nachricht an den Administrator geschickt werden.

- **Testen von Verfügbarkeiten:** Das Monitoring System überprüft hierbei, ob bestimmte Geräte oder Programme momentan aktiv sind. Als Bespiele seien Dienste[2] oder die Verfügbarkeit eines Netzwerks angeführt.

- **Überwachung vorhandener Kapazitäten:** Interessant sind auch verfügbare Ressourcen, wie freier Speicherplatz in einer bestimmten Partition[3] oder verfügbarer freier Hauptspeicher.

Wie bereits erwähnt, ist es sinnvoll, die gesammelten Informationen nach obigen Schema zu bewerten und definierte Maßnahmen automatisiert einzuleiten. Im Zusammenspiel mit dem Monitoring kann somit für eine große Fehlertoleranz gesorgt werden, d.h. kritische Situationen werden durch selbstständig

[2] **Dienste** sind Programme, die beim Hochfahren des Rechners vom Betriebssystem automatisch gestartet werden, wie z.B. der syslog Demon, welcher wichtige Nachrichten des Betriebssystems in einer Datei ablegt.

[3] eine **Partition** ist eine von dem Betriebssystem verwaltete Einheit eines beliebigen Massenspeichers

eingeleitete Reparaturmechanismen abgefangen. Fehlertoleranz ist aber nicht die Aufgabe dieser Arbeit, sondern ein eigenständiges Projekt [FT].

Auch ohne Fehlertoleranz kann eine Überwachung nützlich sein. So kann sie dazu dienen, sich einen Überblick über den Zustand des Rechners zu verschaffen. Weiterhin können auf diesem Wege Performancemessungen durchgeführt werden.. Ein Sensor nimmt die Messung der interessierenden Größe mit hinreichend großer Frequenz vor. Diese Daten werden gesammelt, abgelegt und können anschließend zur Auswertung abgeholt werden.

1.2.2 Monitoring auf Cluster-Ebene

Bei Clustern ist ein Monitoring noch weit wichtiger als bei einem Einzelrechner. Der Grund hierfür ist das Fehlen eines "Metabetriebssystems" auf Clusterebene, welches den Gesamtverbund von Rechnern verwaltet. Das bedeutet, das jeder Rechner zwar mit jedem anderen in Verbindung steht, über den Zustand der jeweils anderen oder des gesamten Clusters weiß er jedoch nichts. Dieser Mangel macht ein System notwendig, das auf dem Cluster für eine große Fehlertoleranz sorgt. So muß beispielsweise bei dem Ausfall eines Rechners dessen Arbeit unter den anderen aufgeteilt werden oder aber die Fehlfunktion eines Subnetzes durch gezielte Umgehung desselben ausgeglichen werden. Höchste Priorität hat dabei stets die gewährleistete volle Funktionsfähigkeit des Clusters, wenn auch eventuell mit geminderter Leistung. Die Fehlertoleranz wiederum muß aber ein Bild von der Situation haben, also den Zustand der Knoten und somit letztlich auch des Clusters insgesamt kennen. Diese Aufgabe wird durch das Cluster Monitoring System erledigt.

Komplexe Aufgaben kann der Cluster nur dann effizient und schnell bearbeiten, wenn Prozesse auf verschiedenen Knoten gestartet werden, die einen Teil zur Lösung beitragen. Zudem gibt es Prozesse, die nicht zwangsläufig an einen Rechner gebunden sind, also bei einer hohen Auslastung desselben besser auf einer anderen Maschine laufen würden. Um die Prozeßverteilung zu optimieren, also ein load-balancing durchzuführen, müssen die Zustandsinformationen aller Clusterknoten bekannt sein, welche das Cluster Monitoring zur Verfügung stellt.

2 Cluster Monitoring: Zielsetzungen und Bestandsaufnahmen

Dieses Kapitel zeigt, welchen generellen Anforderungen ein Cluster Monitoring System genügen muß (Kap 2.1). Im Anschluß daran werden existierende,

frei verfügbare Programme vorgestellt (Kap. 2.2). Es wird gezeigt, inwiefern diese die gestellten Zielsetzungen nicht erfüllen können.

2.1 Anforderungen an ein Cluster Monitoring System

Ein Cluster Monitoring System muß einigen wichtigen Anforderungen genügen, um auf beliebigen Linux-Clustern flexibel und fehlerfrei zu laufen und eine ausreichende Funtionalität bereitzustellen:

- **Einfache Konfiguration:** Das Cluster Monitoring System ist zwangsläufig ein verteiltes System, läuft also auf verschiedenen Knoten des Clusters. Dies werden im Allgemeinen verschiedene Programme sein, darunter die Sensoren auf den zu überwachenden Rechnern sowie weitere Systembestandteile, welche die Daten über Rechnergrenzen hinweg weiterleiten, bündeln und schließlich in adäquater Form archivieren. Wichtig ist hierbei eine Konfiguration, in der für das Gesamtsystem geltende Parameter in verständlicher Form spezifiziert werden können.

- **Dynamische Integration der Sensoren:** Die Zahl der verwendeten Sensoren, sowohl pro Knoten als auch insgesamt, stellt keine statische Größe dar. Das Monitoring System muß also beliebig viele Sensoren dynamisch verwalten können, d.h. neue Sensoren automatisch integrieren und Ressourcen für nicht mehr benötigte Sensoren wieder freigeben.

- **Beliebigkeit der Sensordaten:** Die Sensoren des Monitoring Systems können beliebiger Natur sein. Dies bedeutet, daß die Sensordaten von jeder Art sein können: so muß eine zahlenbehaftete Größe genauso wie ein Text als mögliches Datum zugelassen sein.

- **Einfache Schnittstelle für eigene Sensoren:** Aufgabe des Monitoring System darf es nicht sein, eine möglicherweise gar feste Anzahl von Sensoren zu implementieren. Umgekehrt sollte es also möglich sein, einen Sensor selbst zu programmieren. Damit diese Sensoren an das System ankoppeln können, ist eine Schnittstelle zu diesem wichtig, die einfach zu benutzen ist und die es erlaubt, beliebige Informationseinheiten an das Monitoring System zu übergeben.

- **Gute Skalierbarkeit:** Dies bedeutet, daß es möglich sein muß, das Monitoring System sowohl bei kleinen als auch bei großen Clustern ohne Einschränkung seiner Funktionalität zu betreiben. Konkret heißt dies, bei der Kommunikation über Rechnergrenzen hinweg die Netzlast möglichst gering zu halten. Auch die Last auf den Rechnern, die

einen großen Teil des Datenstroms bearbeiten müssen, ist zu minimieren. Einen wichtiger Punkt ist dabei die **Konsolidierung** der Daten. Konsolidierung bedeutet in diesem Zusammenhang eine Bündelung der Daten zu einem Datenstrom mit dem Ziel, Redundanzen in den Daten auf ein Minimum zu reduzieren und die Netzlast zusätzlich zu verringern, indem durch weniger Netzwerkoperationen protokollspezifischer Netzwerk-Overhead[4] im Mittel abgebaut wird. Desweiteren wird dabei die Integrität der Daten überprüft. Die Leistungsfähigkeit des Systems ist aber zusätzlich noch bestimmt durch die Instanz, welche für die Speicherung und Verfügbarmachung der Daten verantwortlich ist.

- **Sichere Eskalierung:** Werden die ursprünglich von Sensoren stammenden Daten weitergereicht, muß eine sichere Verbindung zwischen den beiden jeweils beteiligten Rechnern vorhanden sein. Das Monitoring System muß also alle gesammelten Daten temporär in geeigneten Strukturen speichern, bis diese dauerhaft gesichert oder aber weitergeschickt wurden. Eine gesicherte Eskalierung erfordert ein verbindungorientiertes Netzwerkprotokoll, welches die Ablieferung aller versandten Datenpakete garantiert.

- **Sicherheit der Datenspeicherung:** Eine besonders kritische Stelle ist der Teil des Systems, an dem die Daten gespeichert werden. Es muß sichergestellt werden, daß alle Informationen ohne Verluste abgelegt werden und wieder ausgelesen werden können. Zusätzlich muß auch der mögliche Ausfall des oder der beteiligten Knoten bedacht werden.

- **Einfache Verfügbarkeit beliebiger gespeicherter Daten:** Damit der Administrator des Clusters oder ein an das Monitoring gekoppeltes Programm die große Menge an gespeicherten Daten nutzen kann, ist es wichtig, gezielt auf eine klar zu spezifizierende Untermenge der Informationen zugreifen zu können. Die Schnittstelle hierfür sollte einfach zu handhaben und eine Eindeutigkeit der Anfrage zu definieren sein. Zusätzlich muß dies schnell stattfinden und parallel dazu müssen weiter Daten des Monitoring System gespeichert werden können.

- **Inhärente Fehlertoleranz:** Es kann prinzipiell nicht gewährleistet werden, daß ein Rechner des Clusters nicht ausfällt oder eine Kommu-

[4] **Overhead:** Ein Datenpaket enthält nicht nur die mit ihm übermittelten Informationen selbst, sondern auch Zusatzinformationen, die für eine korrekte Adressierung sorgen und die Netzwerkhardware ansprechen. Je größer der Overhead, desto kleiner ist das Verhältnis von der Größe der relevanten Information selbst zu der Größe des gesamten Datenpakets.

nikation mit ihm nicht mehr möglich ist. Da ein Verlust von Informationen unerwünscht ist, benötigt das Monitoring System eine inhärente Fehlertoleranz: die Daten müssen an der Schwachstelle vorbei weitergeleitet werden, um sie letztlich sichern zu können. Diese Aufgabe kann nicht der Fehlertoleranz überlassen werden, da sie selbst auf die Daten des Monitoring System angewiesen ist und somit bei fehlender Information nicht agieren kann. Das Cluster Monitoring System muß sich also selbst rekonfigurieren können, um seine Funktionstüchtigkeit unter allen Umständen zu erhalten.

- **Blockierungsfreies Lesen aus den Datenkanälen:** Jeder Rechner, der zu dem Cluster Monitoring System gehört, wird im allgemeinen Fall mehrere Datenkanäle offenhalten, auf denen jeweils Datenpakete eintreffen können. Wurden auf einem Kanal keine Daten geschickt, darf nicht auf das nächste Datenpaket in diesem gewartet werden, bevor die anderen Kanäle überprüft und eventuell ausgelesen werden (non-blocking). Andernfalls könnten Daten hoher Priorität nicht nahezu sofort gelesen werden; zudem würde in dem Fall, daß die Pakete auf jedem Kanal mit einer jeweils festen Frequenz übermittelt werden, der langsamste Kanal die Gesamtlesefrequenz vorgeben. Eine solche gegenseitige Behinderung der offenstehenden Verbindungen ist zu vermeiden.

2.2 Existierende Programme

Die in dieser Diplomarbeit entwickelte Software ist nicht das einzige existierende Cluster Monitoring System: Es gibt eine Menge verschiedener Programme mit unterschiedlichen Schwerpunkten und Architekturen.

Im Folgenden werden ein paar dieser Systeme exemplarisch vorgestellt. Obwohl manche dieser Programme ausgereift und sehr komplex aufgebaut sind, genügen sie nicht in allen Punkten den in Kap. 2.1 vorgestellten Anforderungen. Dies unterstreicht weiter die Notwendigkeit der Entwicklung eines neuen Cluster Monitoring System.

Ausschließlich NGOP und PAM wurden einer genaueren Betrachtung unterzogen, die anderen Programme schieden von vornherein aufgrund ihrer Konzeption aus.

2.2.1 PAM

Bei PAM [PAM] koppeln die Monitoring-Sensoren (MS) auf jedem Rechner jeweils an einen Monitoring Sensor Agent (MSA) an, der die Daten auf einen

zentralen Rechner schickt, auf welchem die Measurement Repository (MR) installiert ist. Die MR speichert die Daten in Texdateien.

PAM zeigt sich in mehrerer Hinsicht den gestellten Anforderungen nicht gewachsen:

- **Überflüssige Informationen:** Ein MSA besitzt stets einen Sensor, der diverse Systemparameter mißt. Diese Daten fallen auch an, wenn sie gar nicht benötigt werden.

  Speicherformat: Da die Daten in einfachen Textdateien abgelegt sind, ist das gezielte Auffinden von Datensätzen sehr uneffizient.

  Fehlende Skalierbarkeit: Da die Daten zentral gespeichert werden, kann PAM nicht beliebig skalieren.

2.2.2 NGOP

NGOP [NGO] stellt ein mächtiges Framework für eigene Sensoren dar. Der NGOP *Central Server* (NCS) nimmt die Daten verschiedener *Monitoring Agents* (MA), die dezentral Daten verschiedener Sensoren der Clusterknoten sammeln, an einer zentralen Stelle entgegen und speichert sie temporär. Der NCS verhält sich hierbei vollkommen passiv. MAs können selbst auf einem vorhandenen Grundgerüst basierend geschrieben werden, wodurch wiederum eigene Sensoren programmiert und die Verfahrensweisen mit den gesammelten Daten gesteuert werden können. Geeignete Anwendungsschnittstellen hierfür sind vorhanden. Ein MA kann registrierte Daten beurteilen und im nötigen Falle Warnungen oder andersweitige Hinweise an den NCS schicken. Falls die Daten dauerhaft abgelegt werden sollen, werden sie an den *Archive Server* weitergesandt, der die Daten in einer Datenbank sichert. Konfiguriert wird das System durch den *Configuration Server*, der alle Konfigurationsdateien zentral für alle NGOP-Komponenten bereithält. Der Anwender kann sich gewünschte Informationen in grafischer Form mit Hilfe des *NGOP MONITOR* anzeigen lassen und hierbei besonders den Status von Sensoren, der durch einen MA nach geeigneter Konfiguration festgesetzt wurde, erfragen.

NGOP erweist sich trotz durchdachter Konzeption als ungeeignet:

- **Installation:** Trotz langwieriger Versuche und auch mit Hilfe gelang es nicht, NGOP funktionsfähig in einer akzeptablen Zeit zu installieren.

- **Skalierbarkeit:** Da der NCS und der *Archive Server* nur auf jeweils einem Rechner installiert sind, ist NGOP aus prinzipiellen Gründen nur bedingt skalierbar.

- **externe Programme:** NGOP verlangt eine Vielzahl externer Bibliotheken und anderer Programme, die zum Teil nur in kommerzieller Form zu erwerben sind.

2.2.3 Ganglia

Ganglia [GAN] besteht im Wesentlichen aus zwei Programmen. Der **Ganglia Monitor Demon (gmond)** läuft auf jedem Knoten. Der **Ganglia Meta Demon (metad)** hingegen wird nur auf einem oder wenigen ausgezeichneten Knoten gestartet. Während gmond die Daten der Sensoren sammelt, faßt der metad diese Daten zusammen und sammelt diese in einer Round-Robin-Datenbank[5]. Vor allem in drei Punkten in der Architektur von Ganglia entspricht dieses nicht den gestellten Anforderungen:

- Um das System fehlertolerant zu gestalten, werden die Daten der einzelnen gmond-Programme abgeglichen: jeder gmond hat ein Bild von dem Zustand des gesamten Clusters. Die dahinterstehende Idee ist leicht zu erkennen: durch die durch dieses Konzept erzielte Redundanz führt der Ausfall eines einzelnen Knotens nicht zum Zusammenbruch des gesamten Systems. Ein großer Nachteil ist aber die immense Netzlast, denn der Abgleich findet durch Multicasts[6] statt. Auch wenn das Monitoring auf dieser Ebene ein verteiltes System darstellt, ist zu erwarten, daß dieser Faktor der Skalierbarkeit doch gewisse Grenzen setzt.

- Ganglia erlaubt nur zahlenbehaftete Sensorinformationen. Der Flexibilität eigener Sensoren sind somit enge Grenzen gesetzt.

2.2.4 Supermon

Supermon [SUP] ist eine Sammlung von Programmen und APIs[7], die zusammengenommen ein Cluster Monitoring System darstellen, welches auf hohe Geschwindigkeit (Abtastfrequenz der Sensoren) optimiert ist. Wiederum wird die angebotene Funktionalität durch zwei Programme ermöglicht: **mon** sammelt die Daten auf jedem Knoten. Diese werden einer höhergestellten Instanz namens **supermon** zugeführt. Ohne auf Details einzugehen, zeigen

[5]Eine **Round-Robin-Datenbank** legt die Datensätze in einer Ringstruktur fester Länge ab. Neue Datensätze überschreiben alte , nachdem der Ring vollständig gefüllt ist.

[6]Bei einem **Multicast** werden die Daten eines Rechners über das Netzwerk gleichzeitig zu vielen Zielrechnern geschickt

[7]**Application Programming Interface (API):** Schnittstelle, die ein Programmierer benutzen kann, um innerhalb seines Programmcodes auf bereits fertig implementierte Funktionen und Programme zugreifen zu können

nachfolgende Punkte die Unzulänglichkeiten von supermon in Bezug auf die verlangten Eigenschaften des Cluster Monitoring System:

- **Integrierte Sensoren:** Supermon hat bereits ein paar integrierte Sensoren. Auch wenn diese Informationen unnötig und nicht erwünscht sind, werden sie stets mitgeschickt und erzeugen einigen Overhead.

- **Schwer portables Datenformat:** Das von Supermon verwendete Datenaustauschprotokoll besteht aus symbolischen Ausdrücken, die rekursiv aufgebaut werden können: ein Ausdruck kann atomare Informationen beinhalten, aber auch aus weiteren Ausdrücken bestehen. Dies ermöglicht ein strukturiertes, hierarchisches Sichern aller Informationen. Problematisch ist aber der Aufbau dieses Protokolls: es ist zwar kurz und erzeugt somit wenig Last auf dem Netzwerk, dafür ist es jedoch nahezu unmöglich, die Daten ohne genaue Kenntnisse des Protokolls zu interpretieren. Besonders schwerwiegend ist, daß die Daten zum Teil binär vermittelt werden, was mit Hilfe von Bitmasken zum gezielten Filtern von Daten verwendet werden kann, jedoch die Portabilität stark einschränkt. Es sollte ein einfacheres, und portableres Format benutzt werden, welches unmittelbar verständlich ist[8] und ebenfalls hierarchische Konzeptionen direkt unterstützt.

- **Laden eines Kernel-Moduls notwendig:** Um Supermon verwenden zu können, muß ein Kernel-Modul[9] installiert werden. Auch wenn Kernel-Module bei vielen Anwendungen und Systemen notwendigerweise eingebunden werden, ist diese Maßnahme bei einem Monitoring System im Grunde vollkommen unötig.

- **Keine einfache Schnittstelle für weiterführende Programme** Eigene Sensoren können bei Supermon über die Schnittstelle **monhole** integriert werden. Bei der Entwicklung eines Sensors muß der Programmierer aber dafür Sorge tragen, daß die von ihm bereitgestellten Informationen genau dem von Supermon gewünschten, recht komplexen Protokoll entsprechen. Wünschenswert wäre dagegen eine deutlich einfacher geartete Schnittstelle, das Monitoring System kümmert sich dann selbstständig um die korrekte Aufbereitung der Daten.

[8] Man spricht dann auch von **"human readable"**

[9] Der **Kernel** ist der Hauptbestandteil des Betriebssystems Linux. Ein **Modul** ist ein dynamisch integrierbares Zusatzprogramm, welches eine erweiterte Funktionalität des Betriebssystems ermöglicht

2.2.5 VACM

VACM [VAC] ist nicht nur ein Monitoring System, sondern wertet gleichzeitig den Status der Knoten aus. So kann der Systemadministrator bei Fehlfunktionen informiert werden. In diesem Sinne hat VACM also auch Elemente einer Fehlertoleranz, ohne jedoch selbst aktiv werden zu können, um Problemfälle zu beheben. Zusätzlich dient das Programm auch dem Cluster Management: über die Software kann auf alle Knoten remote[10] zugegriffen werden. Dieses Design von VACM ist auch der Grund dafür, daß es nur beschränkt tauglich auf manchen Clusterumgebungen ist. So sehr es manchmal angezeigt ist, verschiedene Programme und Anwendungen aufeinander abzustimmen, so sehr kann es von Nachteil sein, diese zu eng zu koppeln. Der Hauptnachteil von VACM ist also, daß keine Entkopplung strukturell unterschiedlicher Aufgaben stattfindet. Dies kann bei der Fehlfunktion einer Komponente eine andere Komponente in Mitleidenschaft ziehen. Einziger Zweck eines Monitoring Systems mit den gegebenen Anforderungen sollte es sein, Daten zu sammeln und zuverlässig bereitzustellen. Die grundsätzliche Idee bei dieser Arbeit ist es also, die verschiedenen Aufgaben, die auf einem Cluster zu erledigen sind, um dieses konsistent und fehlertolerant zu verwalten, so weit wie möglich zu entkoppeln, d.h. unabhängige Programme zu entwickeln, die auf gemeinsame Schnittstellen zugreifen. Auch wenn VACM modular erweiterbar ist, müssen diese Module genau auf das System abgestimmt sein. Desweiteren ist es problematisch, eine eigene Fehlertoleranz auf dem Cluster zu verwenden, ohne daß sich diese und VACM bei manchen Aufgabengebieten überschneiden.

3 Zusammenfassung

Wie dieses Kapitel zeigt, gibt es angesichts diverser Schwachpunkte bereits verfügbarer Cluster Monitoring Software Grund und Motivation genug, ein eigenes Programmpaket zu entwickeln, das vollends den in Kap. 2.1 entwickelten Anforderungen genügen kann. Jedes der vorgestellten Programme hat seine Daseinsberechtigung und wird in speziellen Umgebungen gute Arbeit leisten, alle Forderungen erfüllen kann aber keines.

Es scheint also sinnvoll, ein skalierendes Monitoring System mit einfachen Schnittstellen zu entwickeln, das dynamisch Sensoren verwalten und sich selbst rekonfigurieren kann, falls dies notwendig sein sollte. Auch eine Beschränkung auf die eigentliche Kernaufgabe eines Monitoring Systems, also das Sammeln der Daten von Sensoren, deren Speicherung sowie Verfügbar-

[10] **remote:** ferngesteuert, d.h. auf einem Rechner wird nicht lokal, sondern über eine Netzwerkverbindung indirekt gearbeitet

machung, sowie eine strenge Entkopplung von allen anderen Anwendungen ist bei der Entwicklung der Software unabdingbar.

Teil III
Datenbanken

Das Cluster Monitoring System benötigt eine Komponente, welche die Daten der Sensoren dauerhaft ablegt, wobei die Integrität der Datensätze zu gewährleisten ist. Diese Daten müssen aus Skalierungsgründen dezentral gespeichert werden können und über eine Schnittstelle zentral verfügbar sein (single system view). Dabei sollte auch die Möglichkeit bestehen, gezielt nur auf benötigte Datensätze zuzugreifen. Geeignete Datenbanken haben diese Funktionalitäten bereits implementiert. Kap. 4 erläutert wichtige Begriffe in Bezug auf Datenbanken und zeigt auf, wie eine Datenbank in ein Cluster Monitoring System integriert werden kann. Kap. 5 in diesem Teil der Arbeit beschäftigt sich mit der Leistungsfähigkeit von frei verfügbaren Datenbanken. Wichtig sind dabei nur die Aspekte, die den Betrieb der Datenbank im Rahmen eines zu entwickelnden Monitoring Systems betreffen.

4 Datenbanken und Cluster Monitoring

In diesem Kapitel wird der Begriff Datenbank definiert und erläutert, wie eine Verbindung zu ihr aufgenommen werden kann. Desweiteren wird gezeigt, wie eine Datenbank mit einer Anwendung wie dem Cluster Monitoring System verknüpft werden kann und welche Möglichkeiten und Schnittstellen bereitstehen, um die Datenbank anwendungsspezifisch einzusetzen.

4.1 Definition einer Datenbank

Eine **Datenbank** ist eine strukturelle Sammlung von Daten. Wie diese Daten abgelegt werden und auf welche Weise und mit welchen Operationen mit ihnen verfahren werden kann, ist zunächst nicht weiter spezifiziert. Erst durch solche Möglichkeiten kann aber konstruktiv mit den Datenmengen verfahren werden. Ein Programmpaket, welches Datenbanken verwaltet, ihre Konsistenz gewährleistet und Schnittstellen zum Anwender und anderen Programmen zur Verfügung stellt, bezeichnet man als **Datenbank Management System (DBMS)**.

Ein DBMS hat verschiedene Aufgaben. Es muß alle Datenbanken und Einträge in diese verwalten und dafür sorgen, daß eingefügte Datenmengen sicher und konsistent abgelegt werden **(Integrität der Daten)**. Weiterhin trägt es Sorge dafür, daß auf eine angelegte Datenbank über eine geeignete Schnittstelle einfach zugegriffen werden kann und beliebige Datenunter-

mengen nach bestimmten Kriterien extrahiert werden können. Mithilfe von Programmierschnittstellen ist eine Einbindung einer Datenbank in eigene Anwendungen möglich. Das DBMS regelt auch den Lese- und Schreibzugriff auf eine Datenbank, falls weitere Programme parallel Zugang zu dieser wünschen.

4.2 Client-Server-Prinzip

Es gibt eine Menge verschiedener DBMS, die grundlegende Unterschiede in ihrer Funktionsweise und Implementierung zeigen. Gemein ist aber den meisten, daß sie nach dem Client-Server-Prinzip aufgebaut sind. Dieses Prinzip findet oft Verwendung, wenn es nötig ist, Daten zwischen verschiedenen Prozessen auszutauschen **(Inter Process Communication, IPC)**. Ein **Server** ist ein Prozeß, der passiv auf eine Verbindungsanfrage eines anderen Prozesses wartet. Ein **Client** hingegen schickt aktiv zu einem von ihm gewählten Zeitpunkt diese Anfrage. Wird der Client von einem Server akzeptiert, kann eine dauerhafte Verbindung zwischen beiden hergestellt werden, welche uni- oder auch bidirektional sein kann.

4.3 Integration einer Datenbank in ein Monitoring System

Viele DBMS bieten Programmierschnittstellen an, die es erlauben, Programme zu schreiben, welche mit dem DBMS kommunizieren und gleichzeitig in vollem Umfang Datensätze in einer Datenbank manipulieren können. Diese sind meist als Programmbibliotheken realisiert. Der Programmierer kann das gesamte Datenbankmanagement vollständig dem DBMS überlassen und braucht sich nur um das Ein- und Auslesen von Datensätzen zu kümmern, was meist dennoch gewisse Kenntnisse der DBMS-spezifischen Sprache erfordert. Besonders vorteilhaft ist die Tatsache, daß das Rückgabeformat von Datensätzen an das aufrufende Programm über geeignete Bibliotheksfunktionen oft programmiersprachenspezifisch gewählt werden kann, was die Weiterverarbeitung im selbstgeschriebenen Programmcode sehr erleichtert. Besitzt das DBMS also entsprechende Schnittstellen in derselben Sprache, in der das Monitoring System geschrieben ist, ist eine Kopplung beider Programme ohne weiteres möglich.

4.4 Structured Query Language: SQL

Zur Administration des DBMS, zum Anlegen von Datenbanken und zum Manipulieren von Datensätzen braucht es geeignete sprachliche Mittel, die

unabhängig von der Implementierung des DBMS erst ein sinnvolles Arbeiten mit den gespeicherten Informationen ermöglichen. **SQL (Structured Query Language)** ist eine solche Datenbanksprache. Für SQL spricht ihre Standardisierung, welche auch bei unterschiedlichen standard-konformen DBMS identisches Verhalten und Ergebnisse verspricht. Diese Sprache wird bei **relationalen** DBMS benutzt. Diese speichern ihre Daten nicht notwendigerweise alle in einem einzigen Speicherraum, die Daten können gleichwohl in unterschiedliche **Tabellen** eingefügt werden, welche ihrerseits als **Spalten** unterschiedliche Datentypen ermöglichen. Durch eindeutige Beziehungen zwischen verschiedenen Tabellen **(Relationen)** können diese bei Anfragen miteinander verknüpft werden und Datensätze nach beliebig einschränkenden Kriterien zurückliefern. SQL zeigt eine einfach strukturierte Grammatik, was das Erlernen dieser Sprache erleichtert und syntaktische Fehler, welche zu nicht erwünschten Ergebnissen führen, zu verhindern hilft.

Für ein Cluster Monitoring sind nur wenige Möglichkeiten von SQL und relationalen DBMS von Belang. Die Daten und die Beziehungen zwischen ihnen sind recht einfach geartet, so daß es in erster Ordnung ausreicht, alle Informationen in nur einer Tabelle zu speichern. Viele sprachliche Feinheiten von SQL dienen zudem der gezielten Übermittlung von definierten Teilinformationen an das aufrufende Programm, während beim Monitoring System zunächst das Ablegen der Daten in der Datenbank im Mittelpunkt steht. Eine eingehendere Behandlung von SQL würde den Rahmen dieser Arbeit sprengen und ist zudem nicht ihr thematischer Schwerpunkt, findet folglich an dieser Stelle nicht statt. Zur Vertiefung sei auf [Pet99] verwiesen.

5 Leistungsfähigkeit von Datenbanken beim Einfügen von Datensätzen

Nach Kap. 4 erweist es sich als sinnvoll, ein DBMS in das Cluster Monitoring System zu integrieren. Da unter Umständen eine Menge Daten auf den Knoten anfallen, welche zu sammeln sind und die das DBMS an geeigneter Stelle abzulegen hat, stellt sich die Frage, wie leistungsfähig ein DBMS bei dem Einfügen vieler Datensätze innerhalb einer kurzen Zeitspanne ist. Diese Größe stellt also die Rate dar, mit der gegebene Datensätze in einer Datenbank gespeichert werden können. Gemessen wird sie in [MByte/s], oder alternativ bei fester Größe eines Datensatzes in [Hz].

Die maximale Füllrate ist ein besonders wichtiger Faktor für ein Monitoring System, da dessen Skalierbarkeit wesentlich von diesem Parameter abhängt: werden mehr Daten geliefert, als das DBMS einfügen kann, kommt

es zu einem Stau im Datenfluß. Dies führt zu einer Fehlfunktion des Monitoring Systems, falls keine geeigneten Gegenmaßnahmen getroffen werden. Zumindest ist Monitoring in Echtzeit nicht mehr möglich.

Ein Testen der Füllrate unter verschiedenen Bedingungen ist also unbedingt notwendig, da dieser Parameter möglicherweise zu einem kritischen Verhalten des Cluster Monitoring Systems führen kann. Eine geeignete Architektur des Systems kann zudem nur entwickelt werden, wenn diese Größe bekannt ist und Schwachstellen unter bestimmten Bedingungen erkannt wurden.

5.1 Getestete Datenbanken

Zwei relationale DBMS, die als Sprache SQL verwenden, wurden getestet: PostgreSQL [PGS] und MySQL [MYS]. Beide sind Open Source Software und unterliegen der GPL (General Public Licence). Die Wahl fiel auf diese beiden Systeme, da sie die gebräuchlichsten freien DBMS sind, die zur Verfügung stehen. Zudem sind sie die einzigen Datenbanken, welche die in Kap. 4 genannten Eigenschaften und Schnittstellen in ihrem gesamten Umfang in ausgereifter Form zur Verfügung stellen. Wichtig ist auch, daß sowohl PostgreSQL als auch MySQL den SQL-Standard (SQL92) weitestgehend erfüllen, was die Datenbank als Modul in dem Cluster Monitoring System leicht austauschbar macht. Beide DBMS gelten gemeinhin als sehr stabil und zuverlässig, was nicht zuletzt der starken Entwicklergemeinde, die an beiden Projekten beteiligt ist, zu verdanken ist. So ist davon auszugehen, daß mögliche Implementierungsfehler rasch behoben werden.

In diesem Kapitel wird die Leistungsfähigkeit von MySQL und PostgreSQL bei dem Einfügen verschiedener Datensätze fester Größe ermittelt. Verschiedene, teils auf unterschiedlichen Clusterknoten laufende Prozesse senden diese Daten bei fester Frequenz zu einem zentralen Datenbankserver. Sowohl die Zahl der einfügenden Prozesse als auch die Frequenz, mit der jeweils ein Prozeß Daten versendet, werden hierbei variiert.

5.1.1 PostgreSQL

PostgreSQL ist der Nachkömmling eines Projekts, welches 1986 an der Universität Berkeley in den USA begonnen wurde. Release 1.0 erschien 1989. 1994 schrieben Andrew Yu und Jolly Chen einen SQL Interpreter; zuvor wurde eine proprietäre Datenbanksprache verwendet. Nach einer Portierung des Codes nach ANSI-C wurde das Paket Postgres95 genannt. 1996 erfolgte die Umbenennung in PostgreSQL.

Bei den Tests wurde PostgreSQL 7.0.3 verwendet. PostgreSQL ist SQL92-konform und bietet verschiedene Programmierschnittstellen an. Trigger[11], Stored Procedures[12] und transaktionssichere Tabellentypen[13] sind ebenfalls generisch vorhanden. Nach dem Start des Programms läuft ein Master-Prozeß, welcher für jede eingehende Verbindung einen Tochterprozeß generiert und diesen verwaltet. Die Clientprozesse, welche eine Verbindungsanfrage schicken, kommunizieren über eine Programmbibliothek oder eine einfache shell-artige Benutzerschnittstelle (psql) mit den DBMS-Prozessen.

5.1.2 MySQL

MySQL entstand 1995. Entwickelt wurde es von Michael Widenius bei dem schwedischen Unternehmen TcX. MySQL baut auf dem vom selben Autor geschriebenen Datenbankverwaltungswerkzeug UNIREG (1979) auf.

Es wurde Version 3.23.37 verwendet. MySQL ist nicht vollständig SQL92-konform, dafür sind viele Module und externe Werkzeuge zu finden, die seine Funktionalität erweitern können. MySQL bietet keine Trigger, Stored Procedures und generische transaktionssichere Tabellentypen an, diese können jedoch modular eingebunden und verwendet werden. Der Master-Prozeß von MySQL erzeugt für jede eingehende Verbindung einen neuen Thread[14], der mit dem anfragenden Client kommuniziert. Der Client kann dabei wie bei PostgreSQL über Programmbibliotheken oder eine shell-artige Benutzerschnittstelle (mysql) Zugriff auf die Datenbanken erhalten.

5.2 Das Benchmark-Programm

Um zu auswertbaren Resultaten zu gelangen, mußte zunächst ein Benchmark-Programmpaket geschrieben werden, welches die notwendigen Tests auf einem Cluster möglichst autark auf allen beteiligten Knoten durchführt und die Ergebnisse in einer Datei speichert. Als Grundlage wurde hierbei der iX SQLBench 2.1 [Mei02] herangezogen. Der Benchmark mußte dennoch zum großen Teil selbst geschrieben werden, denn der iX Benchmark läuft nur auf einem einzelnen Rechner und testet primär die Leistung des DBMS beim

[11] **Trigger** lösen unter definierten Umständen gezielte Aktionen aus

[12] **Stored Procedures** sind Programme, die in einer Datenbank des DBMS selbst gespeichert sind und beispielsweise durch Trigger ausgelöst werden können

[13] Transaktionssicherheit bedeutet in diesem Fall, daß atomare Manipulationen an einer Tabelle nur ganz oder aber gar nicht ausgeführt werden, um die Konsistenz der Daten zu sichern

[14] Ein **Thread** ähnelt stark einem Prozeß. Im Unterschied zu diesem greifen Threads auf den selben Programmspeicher zurück. Auch globale Variablen gelten für alle Threads eines Programms.

Auslesen von Datensätzen. Desweiteren können gewünschte variable Parameter nicht als Argument angegeben werden; die Resultate werden ebenfalls nicht in einem Format abgelegt, das von einem Programm direkt ausgewertet werden kann, sondern in einer Textdatei gespeichert, welches für Anwendungen nicht auslesbar ist.

Zum Testen der Datenbankperformance wurden 27 Knoten des TI-Clusters-[CLU] verwendet. Da die Ergebnisse auch von der verwendeten Hard- und Software stark abhängen, folgt eine Auflistung der wichtigsten Kenndaten des Clusters:

- **Betriebssystem:** SuSE Linux 7.2
- **Kernel:** 2.4.18
- **Prozessor:** 2 x Intel Pentium III (Coppermine) mit je 800 MHz
- **Filesystem:** Reiserfs
- **Hauptspeicher:** 512 MB PC-133 SDRAM
- **Netzwerk:** 100 Mbit Fast Ethernet

Das getesteste DBMS wurde auf einem der Knoten installiert. Auf den anderen Knoten hingegen mußte nur die jeweilige shell-artige Benutzerschnittstelle vorhanden sein, welche als DBMS-Client fungiert und mit der direkt Kommandos in SQL an den Server abgesetzt werden können. Durch Verwendung von Pipes kann das Benchmarkprogramm notwendige Kommandos an diese Tools weiterreichen.

Der Benchmark selbst besteht aus einer Reihe von Programmen, die teils in C geschrieben sind und teils die BASH-Shell als Interpreter verwenden. Abb. 1 zeigt, wie der Ablauf eines Tests mit Hilfe dieses Programmpakets schematisch funktioniert.

Die Aufgabe des Benchmarkprogramms ist es, die Füllrate als Funktion der schreibenden Prozesse bei fester Datensatzgröße und fester Frequenz, mit der jeder Prozess diese Daten zu schreiben versucht, zu bestimmen. Ideal wäre es gewesen, die Füllrate als Funktion der Knotenzahl zu ermitteln, wobei auf jedem dieser Knoten nur ein Prozess läuft. Aufgrund der beschränkten Anzahl zur Verfügung stehender Clusterknoten war dies jedoch nur mit bis zu 26 Knoten möglich. Abhilfe konnte jedoch geschaffen werden, indem mehrere Prozesse pro Clusterknoten (n_P) liefen. Dies sollte die Ergebnisse der Tests nicht verfälschen, da bei beiden DBMS der Server zusätzliche Kommunikationsinstanzen für jeden sich verbindenden Client erzeugt, was unabhängig davon sein sollte, an welcher Stelle im Cluster der Client läuft.

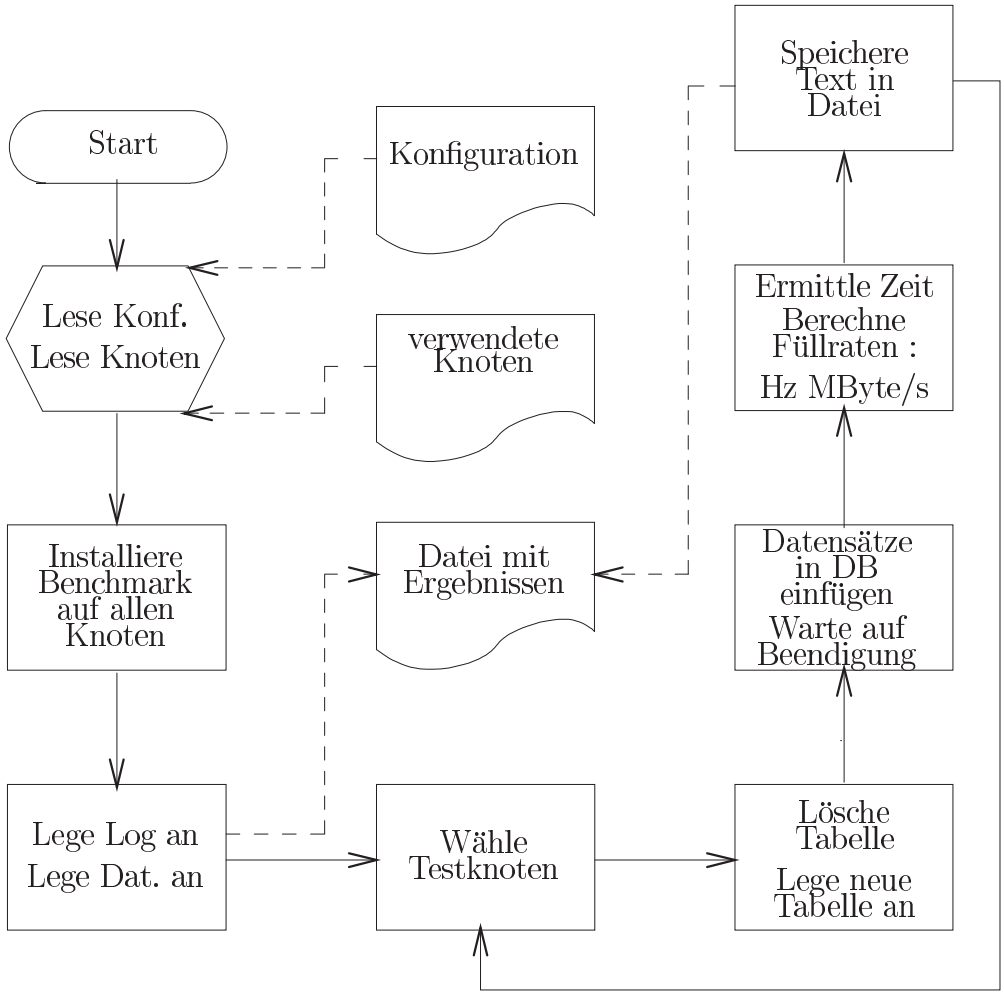

Abbildung 1: Schematischer Ablauf des Benchmarks

Bei dieser Annahme wurde davon ausgegangen, daß nur der DBMS-Server selbst oder die Netzwerkbandbreite auf dem Serverrechner verantwortlich für Füllratenlimits sein kann. Diese Annahme mußte verifiziert werden.

Die Anzahl der schreibenden Prozesse (N_P: Anzahl Client-Prozesse) entspricht also der Zahl der beteiligten Knoten (N_P: Anzahl beteiligter Knoten) multipliziert mit den Prozessen auf jedem Knoten:

$$N_P = N_K \cdot n_P$$

Die Zahl der insgesamt zur Verfügung stehenden Knoten ist bekannt: sie sind in einer Konfigurationdatei angegeben. Eine zweite Konfigurationsdatei enthält Informationen, welche der Benutzerschnittstelle erlaubt, eine

Verbindung zu dem DBMS-Server aufzubauen. An diese Schnittstelle selbst werden wiederum über Pipes[15] SQL-Kommandos abgesetzt. Nachdem der Benchmark auf allen Knoten automatisiert installiert wurde, wird eine Datei angelegt, in der wichtige Schritte protokolliert werden. Eine zweite erzeugte Datei dient der Ablage der Meßergebnisse in einer für weiterverarbeitende Programme lesbaren Form.

Der erste Durchlauf erfolgt nur auf einem Knoten. Alle folgenden Schritte werden solange durchlaufen, bis bei schrittweiser Erhöhung der Knotenanzahl alle Knoten benutzt wurden und parallel Daten in die Datenbank eingefügt haben.

Die Tabelle, welche die Daten fasst, wird gelöscht und gleich wieder erzeugt, enthält also keine Datensätze der je nach Test gewünschten Form und Zusammensetzung. Anschließend werden auf allen für den jeweiligen Meßpunkt aktiven Knoten ($1 \leq N_K \leq$ Zahl der Gesamtknoten) n_K Prozesse gestartet, welche über die Benutzerschnittstelle an das DBMS Daten übermitteln. Sind alle diese Prozesse beendet, wird die dafür benötigte Zeit T ermittelt. Bei konstanter Datensatzgröße ergibt sich dann die Füllrate f_{fill} mit Hilfe der Formel

$$f_{fill} = \frac{N_P \cdot N_{rows}}{T}$$

N_{rows} ist hierbei die Anzahl der Datensätze, die ein Prozess einfügt. f_{fill} stellt dabei eine Effektivfrequenz dar, mit welcher der DBMS-Server maximal Datensätze entgegennehmen und speichern kann.

Gemessen in [MB/s] ergibt sich die Füllrate R_{fill} einfach durch

$$R_{fill} = f_{fill} \cdot S_{data}$$

S_{data} ist die Größe eines jeden konstant großen eingefügten Datensatzes, gemessen in [MB].

Die gewonnenen Daten werden in der entsprechenden Datei abgelegt. Das Programm wird schließlich von neuem ausgeführt, nachdem N_K um eins erhöht wurde. Entspricht N_K der Anzahl der Gesamtknoten, wird das Programm beendet.

Wie an den Formeln abgelesen werden kann, ist die Füllrate proportional zur Zahl der einfügenden Prozesse, falls es diesen gelingt, die Datensätze parallel mit der vorgegebenen festen Frequenz pro Prozess f_P in die Tabelle einzufügen. Es ist zu erwarten, daß dieser lineare Bereich aufgrund von Limits bei einer größeren N_P verlassen wird und die Gesamtfüllrate gegen einen konstanten Wert geht.

[15] Eine **Pipe** erlaubt eine unidirektionale Kommunikation zwischen zwei Prozessen

5.3 Ergebnisse

Beide getesteten DBMS haben diverse Konfigurationsparameter, mit denen Einstellungen an den Servern vorgenommen werden können. Die DBMS wurden mit jeweils folgenden Parametern gestartet:

- **MySQL:** Standardeinstellungen, kein Logging
- **PostgreSQL:** -B600 (600 KB Shared Memory Puffer zur IPC zwischen dem Master-Prozeß und seinen Tochterprozessen statt 64 KB (Standard)). Diese Einstellung ist notwendig, da die Zahl der möglichen Clients mit der Option -N300 auf 300 Clients erhöht wurde (Standard:32) und zum fehlerfreien Betrieb die Puffergröße doppelt so groß sein muß wie die Zahl der möglichen Backends.

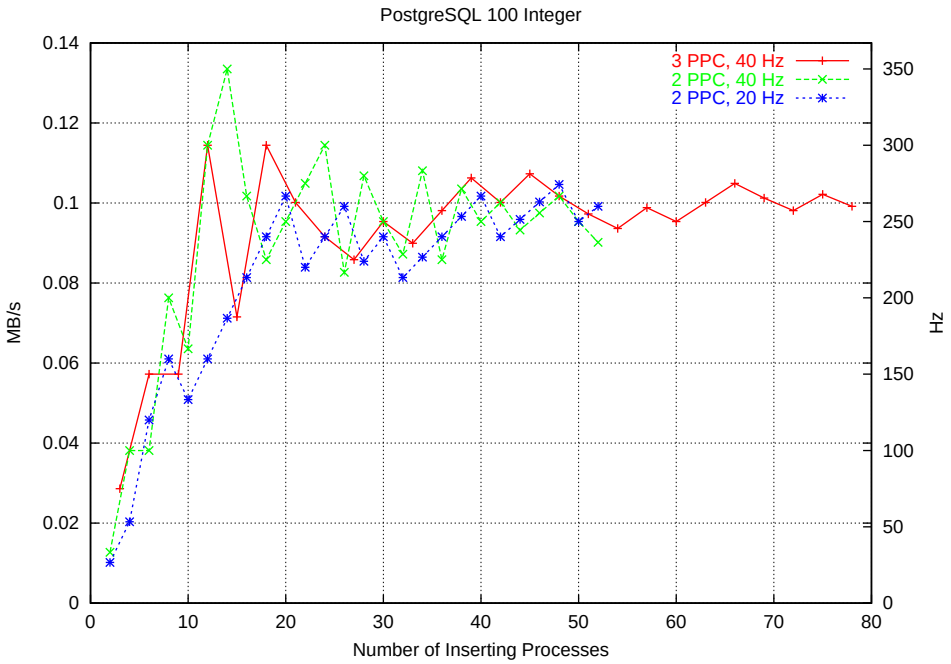

Abbildung 2: PostgreSQL, 100 Integer

Ein genaues Bild kann weiterhin nur gewonnen werden, wenn verschiedene Spaltentypen in unterschiedlichen Zusammensetzungen und Anzahlen innerhalb einer Tabelle getestet werden. Die Zahl der Möglichkeiten ist dabei beliebig groß, es lassen sich aber repräsentative Zusammenstellungen dieser Typen finden, aus denen sich in ausreichendem Maße auf die Leistungsfähigkeit der DBMS als Bestandteil eines Monitoring Systems schließen läßt.

Um Diese Datensätze wurden getestet:

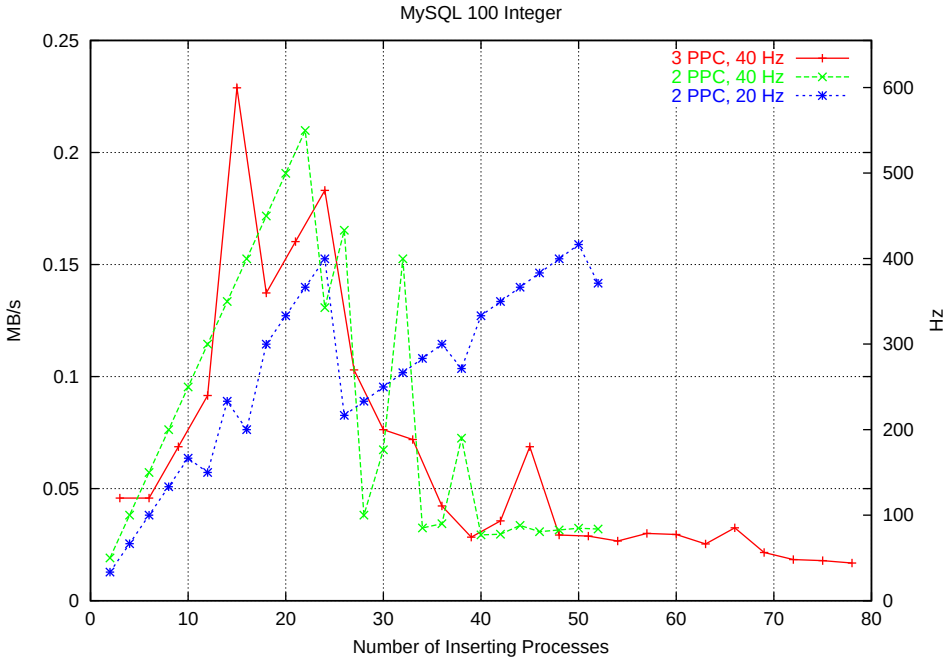

Abbildung 3: MySQL, 100 Integer

- **Text mit 1024 Bytes:** Nur eine Spalte vom Typ TEXT (Variante eines Binary Large Object (BLOB)). 1024 zufallsbasierte Zeichen (1 KB) werden pro Datensatz eingefügt.

  Dieser Datentyp ist besonders geeignet für ein Monitoring System, da Informationen aller Art durch Zeichenketten repräsentiert werden können. Die Größe wurde auf 1 KB festgelegt, da dies eine kanonische Größe darstellt.

- **Text 6144 Bytes:** Wie Text mit 1024 Bytes, nur 6 KB statt 1 KB pro Datensatz.

  Auf diese Weise kann festgestellt werden, wie sich die DBMS bei größeren Datensätzen vom Typ TEXT verhalten. 6 KB ist die maximale Größe eines Text-Tabellentyps, die PostgreSQL verwalten kann.

- **100 Integer:** 100 Spalten vom Typ Integer mit jeweils 4 Byte Größe. Zahlenbehaftete Größen können durch diesen Datentyp repräsentiert werden, es muß dann aber entsprechend viele Spalten dieses Typs geben, um Informationen verschiedener Sensoren sichern zu können.

- **1000 Integer:** 1000 Spalten vom Typ Integer mit jeweils 4 Byte Größe.

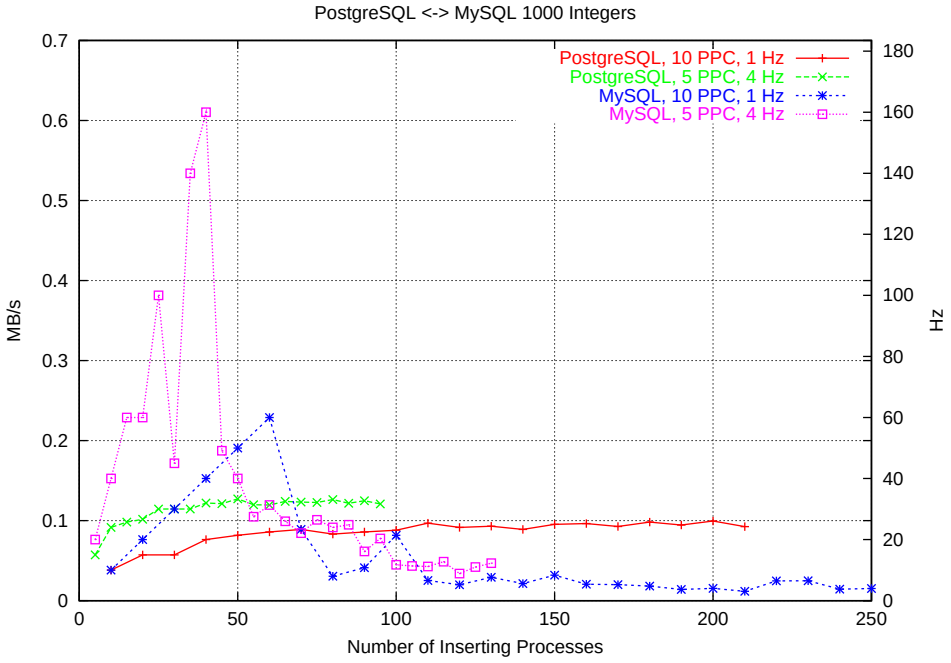

Abbildung 4: MySQL und PostgreSQL, 1000 Integer

Ermöglicht einen Vergleich mit dem Datensatz von 100 Integerwerten.

- **Mixed:** Gemischte Datentypen. 1 x Integer mit 4 Bytes, 1 x Timestamp (Zeitmarke, durch das DBMS selbst erzeugt) mit 4 Bytes, 1 x char(255) (Bis zu 255 Zeichen können in diesem statisch allokierten Datenraum abgelegt werden, der immer 255 Bytes Speicherplatz benötigt), 1 x Text mit 6144 Bytes, 1 x Text mit 1024 Bytes.

  Diese Tabelle zeigt eine Möglichkeit auf, wie die Datenstruktur gewählt werden kann, um abhängig von der zu wählenden Architektur ein Monitoring System repräsentieren.

Die Ergebnisse sind in Diagrammform in Abb. 2-11 zu finden. n_P ist hierbei als PPC (Prozesse pro Client) bezeichnet. Die gewollte Frequenz einer Meßkurve in Hz ist als weiterer Parameter angegeben.

Es ist sofort erkennbar, daß sich PostgreSQL bei vielen parallelen dateneinfügenden Prozessen deutlich performanter zeigt als MySQL. MySQL erreicht zwar stets die höhere maximale Füllrate (f_{max}), bricht jedoch danach dramatisch ein (Integer, Abb. 3, 4) oder zeigt eine starke Streuung der Füllrate, bis diese schließlich in beiden Fällen mit der wachsenden Zahl von Clients weiter fällt. PostgreSQL hat zwar ein f_{max} , die bis um den Faktor 14 kleiner

ist als die von MySQL, zeigt aber ein Verlaufsbild, das den eigentlichen Erwartungen entspricht: die Füllrate steigt zunächst linear mit der Anzahl an Prozessen, flacht dann ab und geht schließlich gegen einen konstanten Wert, der deutlich über dem von MySQL nach dem Einbruch liegt.

Die Benchmarkdurchgänge zeigen auch, daß PostgreSQL Datensätze von dem Typ Text beim Einfügen effektiver bearbeitet als MySQL: Mit wachsender Textgröße steigt die maximale Füllrate bei PostgreSQL, bei MySQL hingegen sinkt sie (Abb. 5-8). Ein umgekehrtes Verhalten ist bei den Integerdurchläufen zu beobachten. Das Einfügen der gemischten Datentypen (mixed) ist offensichtlich durch die Tabellenstandteile vom Type Text dominiert (Abb. 9-11).

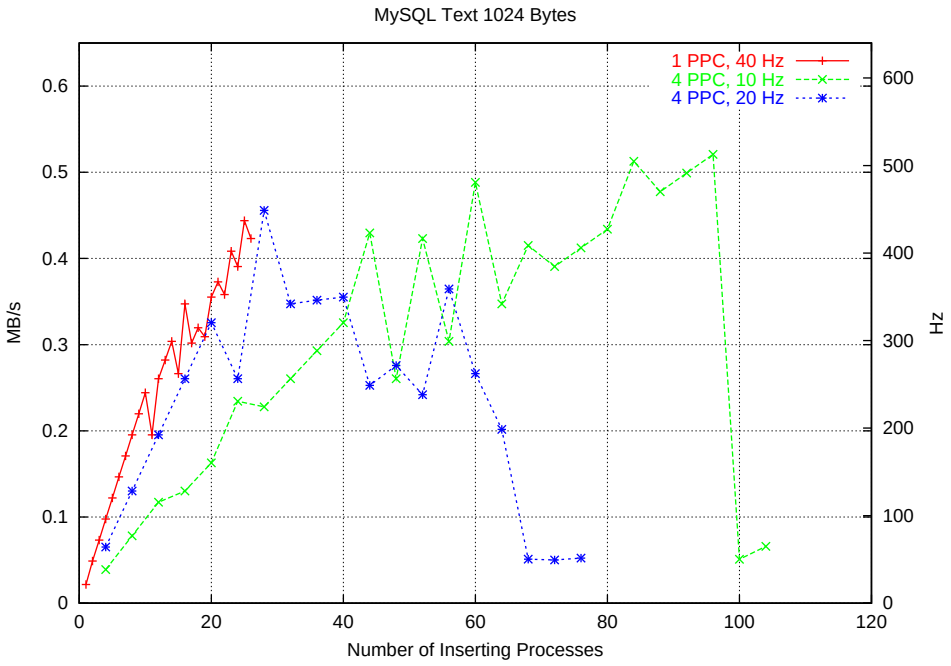

Abbildung 5: MySQL, 1 KB Text

Ziel ist es nun, die Abhängigkeiten der Füllrate von der Anzahl der Client-Prozesse (N_P), den Prozessen pro Knoten (n_P) und der gewollten Frequenz eines Prozesses (f_P) zu ermitteln.

Wie bereits erwähnt und auch erwartet, gibt es keinen Nachweis für einen Einfluß des Parameters n_P auf die Resultate. Halbiert man die gewollte Frequenz f_P eines Prozesses um den Faktor M und erhöht zum Ausgleich die auf einem Knoten laufenden Prozesse n_P um diesen Faktor, zeigt sich ein identisches Verlaufsbild. Eine Ausnahme: in Abb. 4 ist eine leichte Abhängigkeit

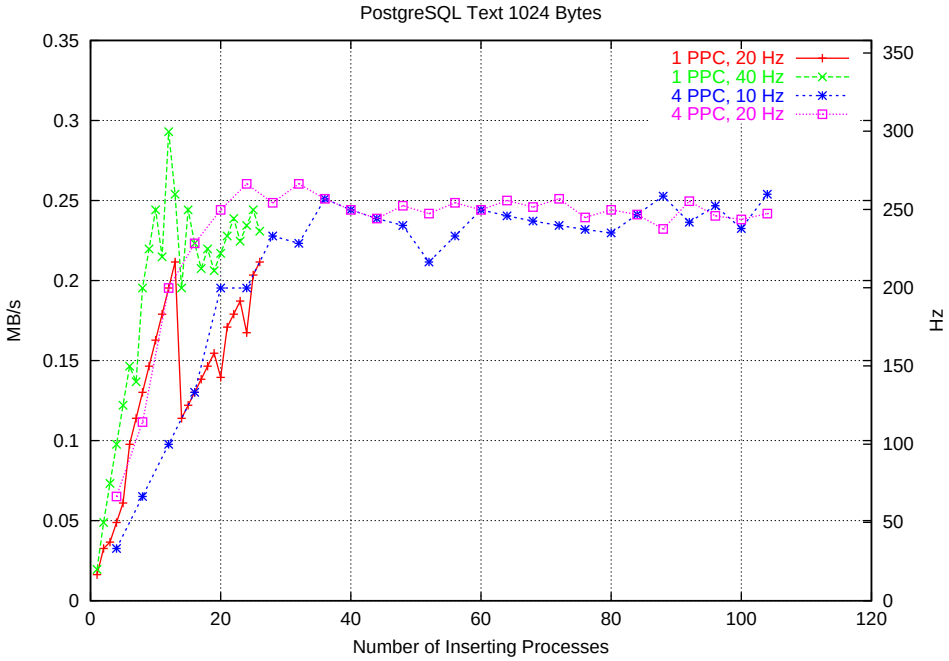

Abbildung 6: PostgreSQL, 1 KB Text

von f_{max} bei dem Einfügen von Integern in eine PostgreSQL-Datenbank zu entdecken, welche ohne genauere Messungen nicht zu erklären ist.

Für beide DBMS läßt sich die Abhängigkeit der effektiven Einfügefrequenz f_{fill} von der Anzahl der insgesamt schreibenden Prozesse N_P folgendermaßen parametrisieren:

$$f_{fill} = A \cdot N_P \tag{5.1}$$

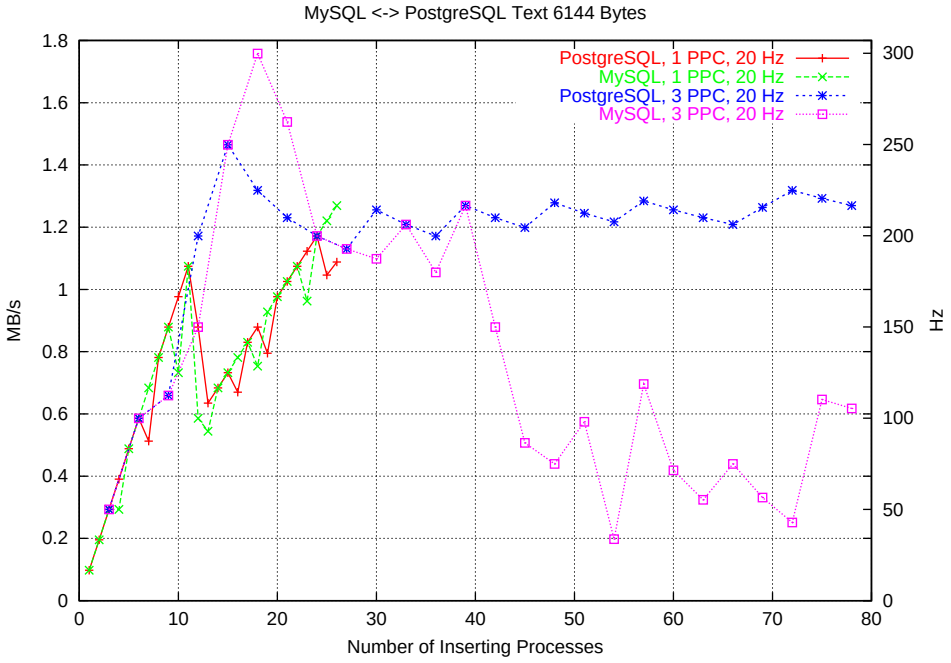

Abbildung 7: MySQL und PostgreSQL, 6 KB Text

Diese Parametrisierung ist so gewählt, daß in dem Falle, daß das DBMS gut skaliert, A konstant ist bei konstantem f_P. Allgemein ist A eine Funktion von N_P und f_P. Auch wenn die Datenlage aufgrund des großen Parameterraums der Benchmarkdurchgänge und zu knapper Meßzeit zu schlecht ist, die Abhängigkeiten des Parameters A zu bestimmen, ist eine solche Parametrisierung jedoch stets möglich. Zudem kann mit ihrer Hilfe eine Erklärung für das beobachtete Verhalten gefunden werden.

Bei MySQL erweist sich A tatsächlich als konstant bis auf nicht erklärbare gelegentliche Schwankungen, bis f_{max} erreicht wird. Anschließend bricht A aber dramatisch ein und überkompensiert dabei sogar das Anwachsen von N_P, so daß die Füllrate f_{fill} stark abfällt und kontinuierlich weiter sinkt.

PostgreSQL zeigt ein anderes und zudem einfacheres Verhalten. f_{max} ist bei diesem DBMS vollständig unabhängig von N_P, n_P und auch f_P, hängt also auschließlich von der Größe der Tabelle und den gewählten Spaltentypen ab. Die Füllrate steigt zunächst linear, was einem konstanten Parameter A in obiger Formel entspricht. Schließlich geht die Füllrate gegen einen konstanten Wert: A ist somit antiproportional zu N_P.

Bei beiden DBMS wurde zudem die Auslastung der CPU und die Netzlast gemessen. Unglücklicherweise sind diese Daten bei einem Systemabsturz verloren gegangen. Es läßt sich aber sagen, daß eine Messung der Netzlast

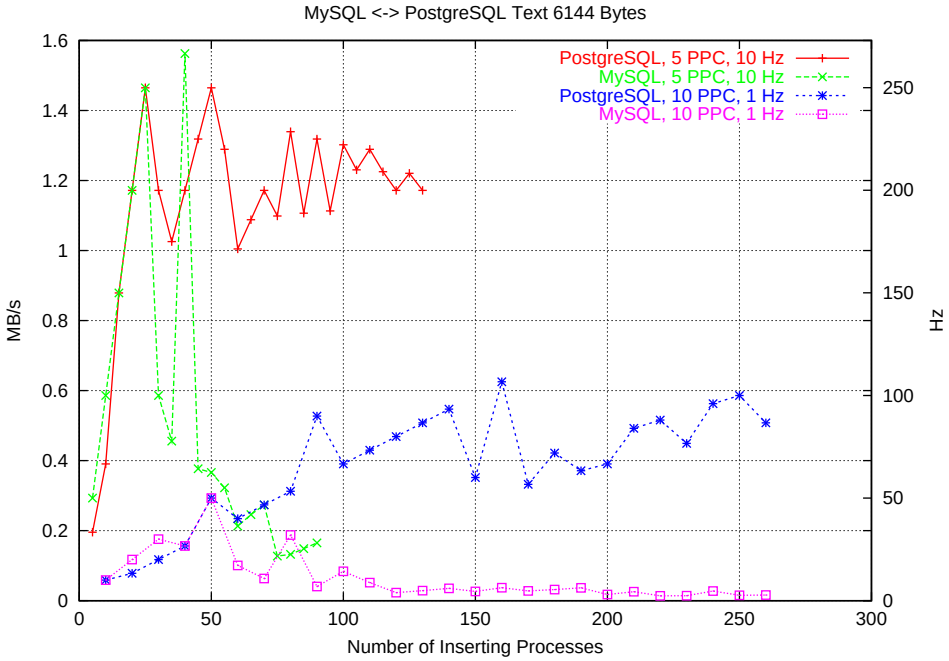

Abbildung 8: MySQL und PostgreSQL, 6 KB Text

keinerlei Aufschluß über die beobachteten Limitierungen liefert. Ein Engpaß der Netzwerkbandbreite ist also nicht der Grund für das Sättigungsverhalten von PostgreSQL und den Einbruch in der Füllrate bei MySQL, was auch durch die Werte von f_{max} bestätigt wird: dieser entspricht stets einer maximalen Füllrate von unter 5 MB/s, während über eine 100 MBit Ethernet Verbindung auch 10 MB/s und mehr möglich sein sollten.

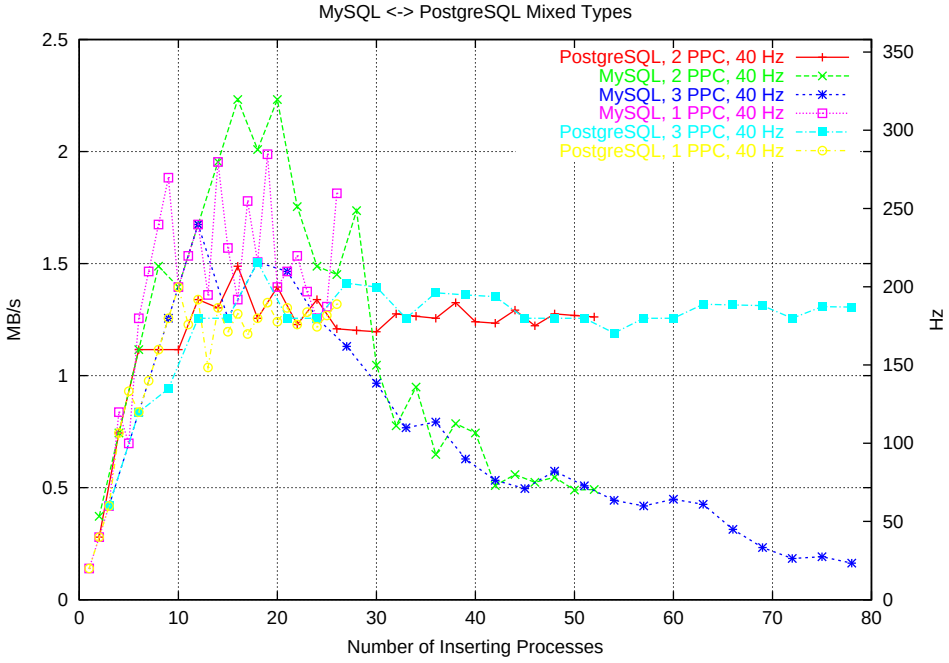

Abbildung 9: MySQL und PostgreSQL, gemischte Datensätze

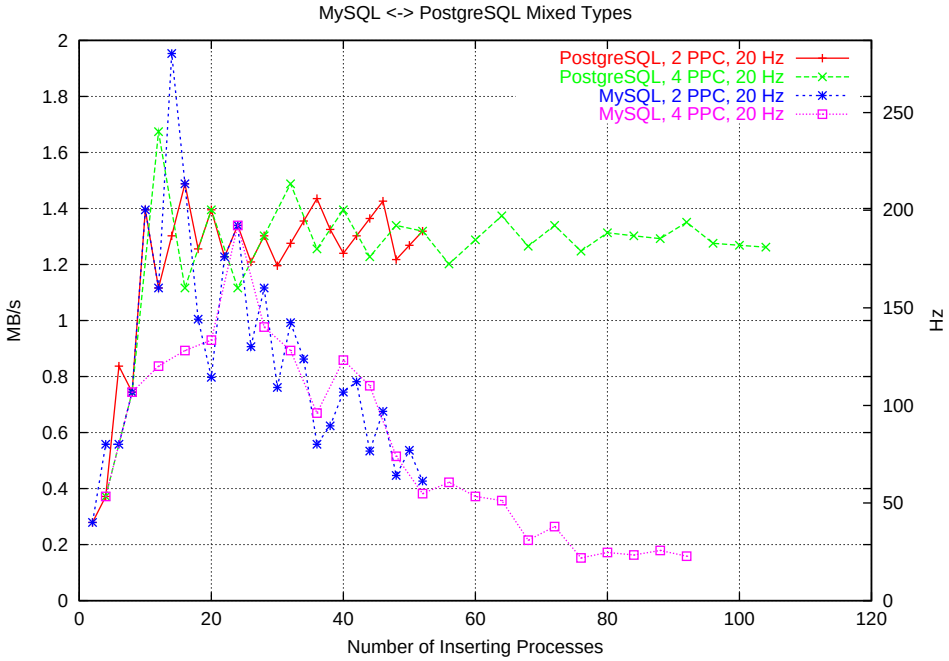

Abbildung 10: MySQL und PostgreSQL, gemischte Datensätze

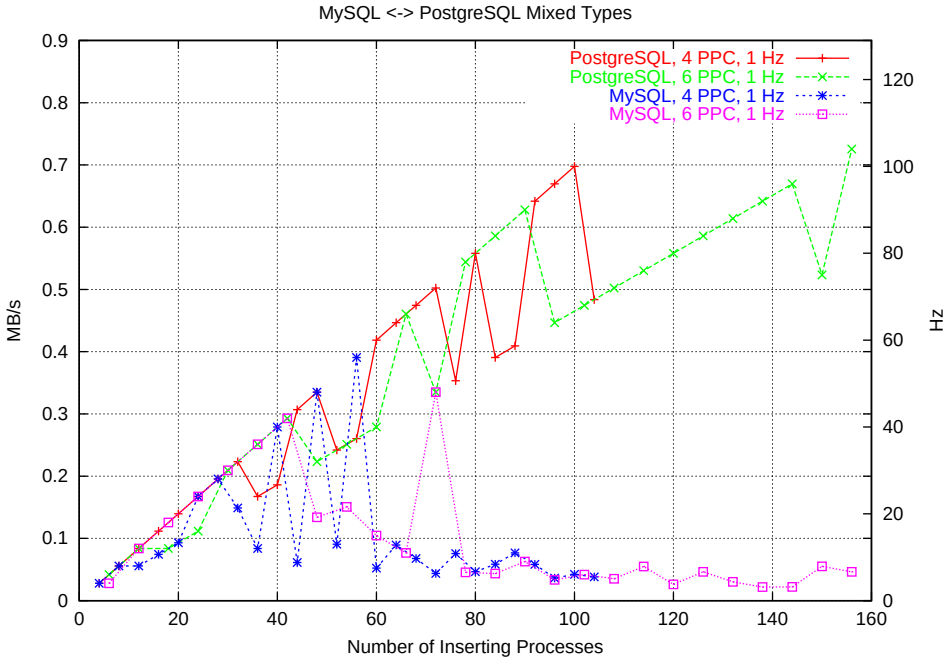

Abbildung 11: MySQL und PostgreSQL, gemischte Datensätze

Erreichen die DBMS ihre Limits, ist aber eine vollständige Auslastung der CPU zu verzeichnen. Bei der gegebenen Hardware ist also der Prozessor und nicht das Netzwerk die limitierende Größe.

Neben den Messungen mit einer variablen Anzahl an schreibenden Prozessen wurden Benchmarks durchgeführt, bei denen nur ein Prozeß lokal in die Datenbank schreibt. Dessen Schreibfrequenz wurde dabei nicht vorgegeben, sie wird also durch das DBMS selbst limitiert.

Folgende Tabelle zeigt die Ergebnisse. Die Fehler waren bei allen Datensätzen gleich groß. Sie ergeben sich aus der Standardabweichung. Bei MySQL betrug der Fehler 10 Prozent, bei PostgreSQL (abgekürzt durch PgSQL) hingegen nur 2 Prozent.

	MySQL[MB/s]	MySQL[Hz]	PgSQL[MB/s]	PgSQL[Hz]
100 Integer	1419,7	0,55	255,5	0,10
1000 Integer	203,6	0,78	15,8	0,06
Text 1024 B	2857,1	2,79	368,1	0,36
Text 6144 B	439,7	2,57	131,9	0,77
Mixed	381,7	2,62	145,0	0,74

5.4 Interpretation

Die Benchmarks zeigten ein sehr unterschiedliches Verhalten der beiden DBMS MySQL und PostgreSQL bei dem Einfügen von Datensätzen mit parallel auf die Datenbank zugreifenden, über das Cluster verstreuten Prozessen. Um die Ergebnisse verstehen zu können, wäre eine genaue Untersuchung des Source Codes beider Datenbanken unumgänglich. Aufgrund begrenzter Zeitressourcen war dies leider nicht möglich, dennoch lassen sich die Resultate beider DBMS in einem einfachen, konsistenten Modellrahmen verstehen. Inwieweit dieses Modell der Realität entspricht, ließe sich nur durch weit aufwendigere und längere Testreihen eruieren. Da der Datenbanktest aber nicht Hauptschwerpunkt dieser Arbeit ist, konnten diese nicht durchgeführt werden und sind zudem für das Design eines Monitoring System von keiner Bedeutung, nachdem das wesentliche Verhalten bereits untersucht wurde.

Es wird angenommen, daß die Prozesse (PostgreSQL) oder Threads (MySQL, im folgenden der einfachkeit halber auch Prozeß genannt) der DBMS die einzigen Prozesse auf dem Serverrechner sind, die nennenswert CPU-Zeit verlangen. Die Auslastung des Prozessors für jeden Prozeß, der Daten annimmt und in die Tabelle einfügt, wird als A bezeichnet. A ist eine Funktion von N_P und f_P. Die Prozessorlast durch alle laufenden Prozesse ist insgesamt A * N_P, wobei angenommen wird, daß jeder Prozess die CPU in gleichem Maße belastet, was aber durch ein gutes Scheduling[16] gegeben ist. Dies wird durch Abb. 12 bewiesen. 300 Prozesse arbeiten parallel einen identischen Code ab. Jeder Prozeß ist durch eine eindeutige Nummer gekennzeichnet (y-Achse). Nach seiner Beendigung gibt jeder Prozeß jeweils die benötigte Zeit in Microsekunden zurück (x-Achse). Es ist zu erkennen, daß alle Prozesse die gleiche Zeit benötigen, wobei die Abweichung vom Mittelwert im Promille-Bereich liegt.

Weiterhin muß aber berücksichtigt werden, daß sich die Prozesse durch eine IPC oder Locking-Mechanismen gegenseitig behindern können, was den Prozessor zusätzlich belastet. Der diesem Faktor zugeordnete Parameter bekommt die Bezeichnung B. B * N_P ist die Zusatzbelastung durch die Interaktion der Prozesse untereinander. B kann dabei noch Abhängigkeiten von N_P und f_P aufweisen. Nennt man die maximal mögliche Prozessorlast C, so gilt:

$$Prozessorlast = A \cdot N_P + B \cdot N_P \leq C \tag{5.2}$$

Die Zeit, die der Prozessor relativ zur Gesamtzeit verwendet, um Daten

[16] der **Scheduler** ist der Teil eines Betriebsystems, der für die Zuteilung der CPU-Zyklen an die Prozesse zuständig ist.

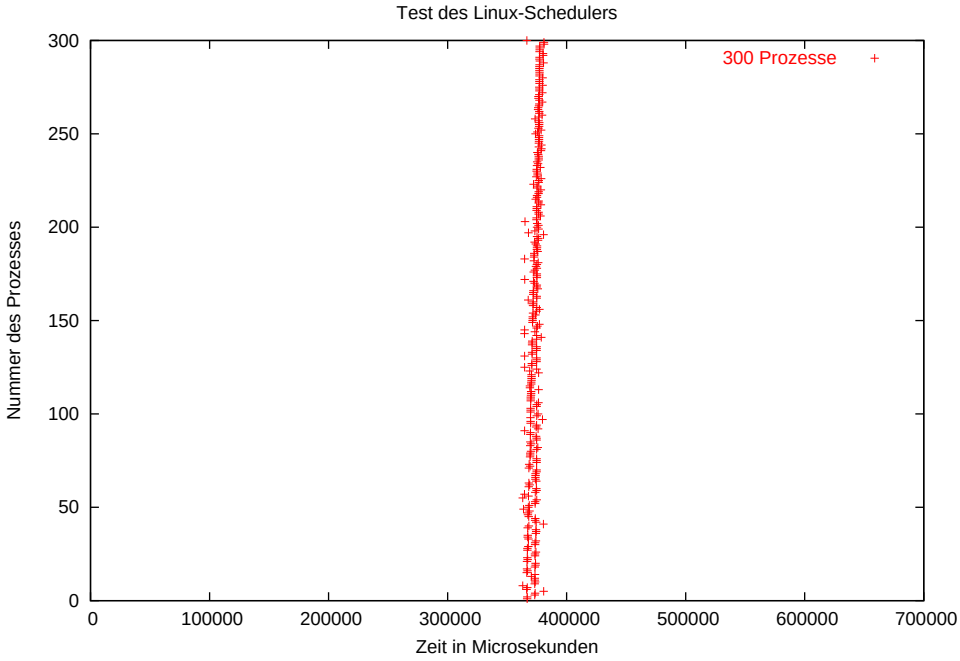

Abbildung 12: Test des Schedulings von 300 unabhängigen Prozessen

einzufügen, ist:

$$\frac{T_{einf.}}{T_{ges.}} = \frac{A \cdot N_P}{A \cdot N_P + B \cdot N_P} \tag{5.3}$$

Die CPU-Leistung, die also effektiv zum Einfügen von Datensätzen gebraucht wird, ergibt sich durch Multiplikation dieser beiden Formeln. Nimmt man an, daß f_{fill} proportional zu dieser Größe ist, erhält man

$$f_{fill} \sim eff.CPU - Last = (A \cdot N_P + B \cdot N_P)\frac{A \cdot N_P}{A \cdot N_P + B \cdot N_P} \tag{5.4}$$

also gerade A * N_P, was bekannter Formel entspricht.

Hat die CPU noch genügend Leistungsreserven, so kann A als konstant angenommen werden. f_{fill} steigt also linear mit N_P, was bei beiden DBMS zunächst tatsächlich der Fall ist. Sind aber genügend Prozesse vorhanden, daß bei Gl 5.2 das Gleichheitszeichen gilt, so läßt sich diese Gleichung nach A auflösen, was in Gl 5.1 eingesetzt

$$f_{fill} \sim C - B \cdot N_P \tag{5.5}$$

ergibt. C und B sind hierbei konstant.

Gl. 5.5 erlaubt es nun, die Resultate zu deuten.

Bei PostgreSQL scheint es zu keiner gegenseitigen Behinderung der Prozesse zu kommen. Dies hat zur Folge, daß der Parameter B faktisch identisch Null ist. Nach Gl. 5.5 heißt dies, daß in dem Falle, daß die Auslastung des Prozessors 100 Prozent beträgt, die Füllrate proportional zu diesem Limit, also konstant ist. Anders ausgedrückt: jeder beteiligte Prozeß bekommt weniger CPU-Zeit durch den Scheduler zugeteilt, über alle Prozesse summiert bleibt dies verfügbare Gesamtzeit aber konstant, was zu einer konstanten Füllrate führt.

MySQL hingegen hat einen endlichen und mit A vergleichbaren Parameter B. Dies führt nach Gl 5.5 dazu, daß bei einer Auslastung des Prozessors die Füllrate mit steigender N_P sinkt. Anzumerken ist dabei, daß der Wert natürlich niemals kleiner als 0 werden kann; dies passiert nur aufgrund vereinfachender Annahmen in diesem Modellrahmen, falls B nicht ebenfalls genügend stark mit N_P fällt. Über die Gründe der gegenseitigen Behinderung der Threads kann an dieser Stelle nur gemutmaßt werden. Wie in Abb. 12 zu sehen ist, kann der Scheduler des Betriebssystems bei unabhängigen Threads nicht verantwortlich dafür sein: alle Threads bekommen bis auf kaum meßbare Abweichungen die gleiche CPU-Zeit zugeteilt. Diese sind bei dem getesteten Programm allerdings unabhängig voneinander: die Threads bei MySQL werden aber über geeignete Mechanismen miteinander und dem DBMS kommunizieren müssen. Die Implementierung dieser Kommunikation bei gleichzeitiger Gewährleistung der Datenintegrität bei parallel auf die Datenbank zugreifenden Threads ist offenbar derart, daß sich die Threads gegenseitig behindern. Diese Behinderung wird wahrscheinlich durch exklusive Rechte eines Threads, aus der Tabelle zu lesen oder aber in sie zu schreiben, verursacht (table locking). Aufgrund dieses Sachverhalts gehen CPU-Zyklen verloren, da alle anderen Threads warten müssen.

5.5 Optimierungsmöglichkeiten

Es stellt sich zuletzt die Frage, ob die Füllraten nicht durch zusätzliche Maßnahmen und Herangehensweisen weiter gesteigert werden kann.

Eine Idee war es, die Datenbank selbst auf einer Ramdisk statt einer Festplatte anzulegen. Dies führte aber zu keiner Verbesserung der Ergebnisse, da die Limitierung nicht durch einen zu langsamen Plattenzugriff, sondern die CPU-Last bei dem Einfügen der Daten gegeben ist. Die eingesetzten Festplatten erlauben zudem Schreibraten von mehr als 20 MB/s, die maximalen Füllraten der DBMS liegen aber um den Faktor 5 und mehr darunter.

Beide DBMS bieten zudem die Möglichkeit, Datensätze aus geeignet formatierten Dateien einzulesen. Diese Maßnahme kann zwar die Füllrate stark

erhöhen, wird aber im Rahmen des Monitoring System nicht verwendet. Ein Grund ist die Tatsache, daß die entsprechenden Kommandos bei MySQL und PostgreSQL verschieden sind und somit im Unterschied zur Verwendung von SQL keine Verwendung generischer Schnittstellenfunktionen erlauben. Ein weiterer Grund ist, daß viele Datensätze zunächst in einer Datei abgelegt werden müssen und schließlich der Datenbank zugeführt werden. Ist die Datei fehlerhaft, kann dies zu dem Verlust aller Informationen oder gar zu dem Einfügen falscher Daten führen, was die Datenintegritätsmechanismen der DBMS zunichte macht. Die gewonnene Geschwindigkeit ist aber beachtlich, was aufgrund von Zeitmangel nur angetestet werden konnte.

Die DBMS haben zudem diverse Puffer und Speicherzuteilungsparameter, die die Füllrate potentiell beeinflussen können. Es konnten jedoch keinerlei Verbesserungen durch eine Veränderung der Ausgangswerte festgestellt werden.

MySQL bietet einen erweiterten INSERT-Befehl, der mehrere Datensätze mit einem Kommando in die Tabelle schreibt. Dieser Befehl kann die Leistungsfähigkeit verbessern, ist aber nicht SQL-kompatibel und zudem empfindlich gegenüber Pufferüberläufen, kann also schlecht bei dynamisch großen Datensätzen verwendet werden, was für ein Monitoring System von großer Bedeutung ist.

PostgreSQL kann das Einfügen mehrerer Datensätze zu einer Transaktion zusammenfassen. Desweiteren kann die fsync-Option, die als Standardparameter aktiviert ist, abgeschaltet werden. Dies führt zwar zu einer stark verbesserten Füllrate, die Datenintegrität ist aber gefährdet, da nicht jede ausgeführte Transaktion auch auf Festplatte gespeichert wird. Dennoch ist besonders die Transaktion vielversprechend: auch hier war aus Zeitgründen keine Messung mehr möglich.

6 Zusammenfassung

Zwei DBMS, MySQL und PostgreSQL, wurden auf ihre Leistungsfähigkeit bei dem Einlesen verschiedener Datensätze konstanter Größe in eine Datenbank getestet. Ein besonderer Augenmerk lag dabei auf der Abhängigkeit der gesamten Füllrate von der Zahl der Prozesse, welche die Daten zu dem DBMS-Server schicken.

MySQL und PostgreSQL zeigen bei den Benchmarks ein sehr unterschiedliches Verhalten. Schreibt nur ein Prozess lokal in die Datenbank, so ist MySQL stets deutlich schneller als PostgreSQL. PostgreSQL skaliert deutlich besser mit einer wachsenden Zahl an verbundenen Prozessen, die Gesamtfüllrate erreicht einen akzeptablen, konstanten Wert, der stabil ist und nur von

der Art eines Datensatzes abhängt. MySQL hingegen skaliert sehr schlecht mit einer wachsenden Zahl von Clients: nach dem erreichen eines Maximums fällt die Füllrate dramatisch ab und liegt weit unter dem von PostgreSQL.

Festzustellen ist in jedem Fall, daß die Füllraten beider DBMS weit unter den Geschwindigkeiten bleiben, die erreicht werden können, wenn die Datensätze einfach in eine Datei auf Festplatte geschrieben werden: hierbei sind gut 20 MB/s möglich.

Die erhaltenen Resultate können mithilfe eines einfachen Modells verstanden werden. Die Datengrundlage reicht aber nicht aus, die Richtigkeit des Modells oder gar seine Parameter zu bestimmen, dennoch bleibt festzustellen, daß auf diese Weise eine plausible Erklärung der Benchmarkdurchläufe möglich wird. Genauere Ursachen können nur durch eine Überprüfung von Programmaufrufen oder einen genauen Blick auf den verfügbaren Source Code beider DBMS gefunden werden.

Auch wenn die Füllraten zunächst gering erscheinen, die DBMS eröffnen durch ihre Schnittstellen, eine gewährleistete Datenintegrität und die einfache, schnelle Selektion beliebiger Teildaten Möglichkeiten, die andersweitig kaum in ein Monitoring System zu integrieren sind. Bei der Architektur desselben ist aber zu beachten, daß bei dem Ablegen der Daten in die Datenbank möglichst wenige Prozesse auf die Datenbank zugreifen sollten, um ein Optimum in der Füllrate zu erreichen. Dies gilt in speziellem Maße für MySQL. Diesem Umstand muß bei dem Design des Monitoring Systems Sorge getragen werden.

Es ist zuletzt noch zu bemerken, daß das kommerzielle Datenbankprodukt Oracle [ORA] zum Vergleich mit den beiden DBMS herangezogen werden sollte. Leider ließ sich Oracle aber nicht in der verfügbaren Zeit installieren, da die Entwicklung des Cluster Monitoring Systems selbst Vorrang hatte.

Teil IV
Grundlegende Konzepte des Monitoring Systems

Der Hauptteil dieser Arbeit bestand in der Entwicklung eines Hierarchischen Cluster Monitoring System (HCMS). Bevor dieses aber implementiert werden kann, müssen noch einige grundsätzlichen Entscheidungen getroffen werden, die das Design des HCMS betreffen. Da die entwickelten Konzepte letztlich gleichbedeutend mit der Architektur des Cluster Monitoring Systems sind, besteht dieser Teil nur aus Kap. 7, welches sich mit derselben beschäftigt..

7 Die Architektur des Cluster Monitoring System HCMS

Ein Cluster Monitoring System verlangt nach einem durchdachten Aufbau, um die Aufgaben, die in seinem Zuständigkeitsbereich liegen, sicher und zuverlässig erledigen zu können. Zu beachten sind bei der Entwicklung des Designs besonders die in Kap. 2.1 aufgeführten Anforderungen, berücksichtigt werden muß aber auch die in Teil III untersuchte Leistungsfähigkeit der DBMS, welche in das HCMS integriert werden sollen. Die Architektur, durch welche eine gute Skalierbarkeit erreicht wird, wird in Kap. 7.1 beschrieben. Kap. 7.2.1 zeigt, welche Punkte bei der Datenübermittlung zwischen verschiedenen Knoten eine Rolle spielen und mit welchen Methoden und Übertragungsprotokollen eine geeignete Kommunikation stattfindet. Da das HCMS weitestgehend von anderen clusterverwaltenden Programmen entkoppelt sein soll, stellen auch die Schnittstellen einen besonders zentralen Punkt dar (Kap. 7.3). Das Ergebnis der Fragen, die das Design des HCMS betreffen wird in Kap. 8 skizziert.

7.1 Hierarchisches Konzept

Konzeptionell ist es ein wichtiges Ziel, daß das HCMS gut skaliert. Dies bedeutet, daß im Grunde beliebig viele Knoten in einem beliebig großen Cluster überwachbar sein müssen. Als direkte Folge hiervon ist es unmöglich, nur einen zentralen Server zu verwenden, an den alle mit Sensoren versehenen Clusterknoten ihre Daten schicken: der maximale Datenfluß, den der Server akzeptieren kann, ist durch seine Leistungsfähigkeit oder Netzwerkbandbrei-

ten beschränkt; bei der Verwendung eines DBMS zur Datenspeicherung gibt es sogar noch eine stärkere Limitierung durch das DBMS selbst (siehe Teil III). Wird als DBMS MySQL oder PostgreSQL verwendet, kann ein Server abhängig von der gewählten Datenstruktur nicht viel mehr als 2 MB/s an Daten annehmen

Ein geeigneter Ansatz, dieses Dilemma zu lösen, besteht in der Verwendung einer Serverhierarchie. Hierbei werden die Datenströme aufgeteilt, mehrere Server nehmen jeweils nur einen Teil der gesamten Daten entgegen. Diese Informationen werden so weit wie möglich gebündelt und an eine höhere Serverinstanz weitergesandt, auf der sich weniger Server befinden. Diese Server reichen die Daten wiederum an die nächste Hierarchieebene weiter. Das Ganze findet so lange statt, bis die höchste Hierarchieebene erreicht ist oder aber aufgrund von besagten Ressourcenlimits die Daten nicht weitergeschickt (**eskaliert**) werden können. Erkennt das HCMS, daß einer der beiden Fälle eingetroffen ist, so werden die Daten in einer auf jedem Server lokal verfügbaren Datenbank gesichert. Dieses Design bietet also auch den Vorteil, daß sinkende Füllraten bei der Datensicherung mithilfe eines DBMS bei mehreren schreibenden Prozessen keine Rolle spielen (siehe Teil III): nur ein einziger Prozeß schreibt jeweils auf einen Datenbankserver. In Abb. 13 ist der Aufbau einer solchen Hierarchie schematisch dargestellt.

Die grünen Knoten am unteren Ende der Abbildung sind Clusterrechner, auf denen ausschließlich Monitoring-Sensoren laufen. Die Daten dieser Clients werden in die unterste Ebene der Serverhierarchie (rot) eingespeist (Level 0), nachdem sie mit Zusatzattributen wie einer Timestamp[17] versehen wurden. Diese Daten müssen in jeder Ebene auf ihre Konsistenz überprüft und zusammengefaßt werden, wodurch unnötige Redundanzen vermieden werden (**Konsolidierung**). Jeder Knoten innerhalb der Hierarchie kann selbst auch Sensoren besitzen. Diese lokal gewonnenen Daten werden von dem Rechner unabhängig von den konsolidierten Daten einer hierarchisch tiefergelegenen Schicht weiter nach oben gereicht, wo sie bei der Konsolidierung zusammengefaßt werden. Diese Art der Eskalierung ermöglicht es, die Sensoren und deren Verwaltung von dem Programm zu entkoppeln, das für die Konsolidierung und Eskalierung zuständig ist. Dieses Programm ist auch dafür zuständig, die Datensätze in einer Datenbank zu speichern, falls eine Weitereskalierung unmöglich oder aber die höchste Ebene der Hierarchie erreicht ist. Jeder Rechner innerhalb der Serverhierarchie ist zugleich Client und Server. Eine Ausnahme ist die oberste Ebene, deren Rechner nur als Server dienen (sie sind aber auch Clients des DBMS, das sie zur Datenspeicherung verwenden).

Die Aufbaustrategie der Serverhierarchie sieht folgendermaßen aus:

[17] Eine **Timestamp** gibt den Zeitpunkt der Messung eines Sensors an.

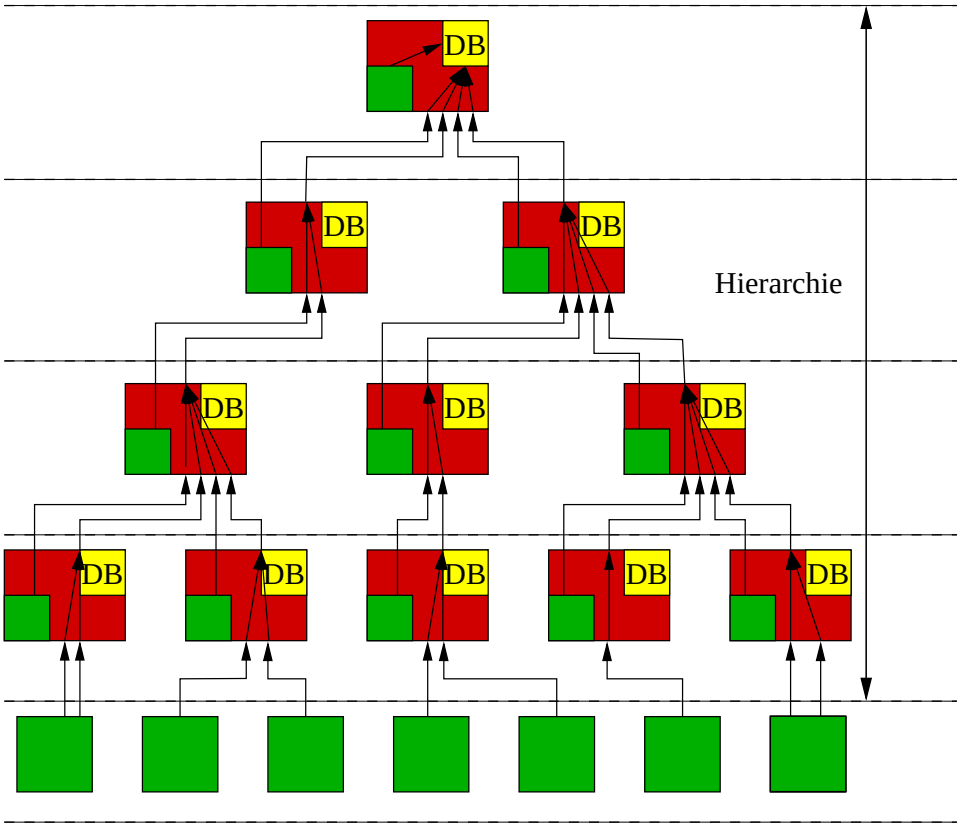

Abbildung 13: Hierarchie beim HCMS. Grün: Clients mit lokalen Sensoren. Rot: Server

- Jeder Rechner im Cluster liest die Level der Serverhierarchie aus einer Konfigurationsdatei ein.
- Die Hierarchie wird von oben nach unten aufgebaut. Erkennt ein Rechner, daß er zur Serverhierarchie gehört (sein Name ist in einem der Level in der Konfigurationsdatei enthalten), so versucht er, eine Verbindung zu einem Server der nächsthöheren Ebene aufzubauen. Jeder Server in der Hierarchie, abgesehen von der höchsten, ist gleichzeitig auch Client.
- Der Server, mit der sich ein Client verbindet, ist nicht festgelegt. Der Client weiß nur, welche Server ihm potentiell zur Verfügung stehen. Er versucht, alle Server der nächsten Ebene zu erreichen. Jeder Server, der ihm antwortet, also tatsächlich verfügbar ist, gibt dem Client Informationen über seine momentane Auslastung und die bereits mit ihm verbundenen Clients, so daß dieser weiß, welcher Server der best-

geeignetste (d.h. derjenige mit der geringsten Auslastung) ist. Mit dem gefundenen Server verbindet sich der Client letztendlich.

- Taucht der Name eines Rechners nicht in der Konfigurationsdatei auf, so bedeutet dies, daß er nicht zur Serverhierarchie gehört. Der Computer befindet also in der Clientebene Auf dem Computer laufen somit ausschließlich Sensoren. Er sucht einen Server in der untersten Hierarchieebene, wobei der verwendete Suchmechanismus dem im vorigen Punkt beschriebenen entspricht.

7.2 Datenübermittlung zwischen den Hierarchieebenen

Ist die Serverhierarchie des Monitoring System erst einmal aufgebaut, müssen die von den Sensoren ermittelten Informationen in geeigneter Weise von Rechner zu Rechner transportiert werden. Es zeigt sich, daß XML hierfür eine gute Wahl darstellt (Kap. 7.2.1). Desweiteren ist es notwendig zu definieren, welche Informationseinheiten ein Sensor beinhaltet und wie diese sprachlich mit XML auszudrücken sind (Kap. 7.2.2).

7.2.1 XML

Das Datenübermittlungsformat des HCMS soll folgende Eigenschaften besitzen:

- **Prägnanz:** Keine überflüssigen Informationen und unnötige Redundanzen, da dies zu erhöhter Netzlast führt. Das Format sollte zudem nur einen geringen Overhead haben.

- **Erweiterbarkeit:** Das Austauschformat darf nicht statisch sein, da beliebige Sensoren in beliebiger Anzahl gewonnene Daten übermitteln können müssen. Wichtig ist, daß jeder Sensor mehrere Datensätze in einem Austauschobjekt ablegen können muß, denn es muß möglich sein, viele Datensätze verschiedener Sensoren in geeignet konsolidierter Form in einem Durchlauf zu eskalieren, um viele Schreib- und Lesezugriffe auf die Daten zu vermeiden: Dies würde zu einer starken CPU-Belastung führen, zudem würde sich der Netzwerk-Overhead erhöhen.

- **Kein binäres Format:** Hierdurch ließe sich die Datengröße zwar verringern, jedoch drohen implementierungsspezifische Probleme, die eine Portierung auf andere Systeme erschweren können. Desweiteren ist es schwieriger, dynamisch beliebige Sensorinformationen zu übermitteln.

- **Durchsichtigkeit:** Der Inhalt der Datenpakete sollte klar strukturiert sein, am Besten in hierarchischer Form. Zudem wäre es wünschenswert, die Struktur leicht für eine beliebige Anwendung lesbar machen zu können, insbesonders aber auch für Menschen ("human readable").

- **Standardisierung:** Ideal ist es, wenn das verwendete Format einem Standard genügt, der prinzipiell die Weiterverarbeitung durch Programme beliebiger Art erlaubt, die dieses Protokoll verstehen.

XML zeigt sich diesen Anforderungen gewachsen. XML [Ray01] (eXtensible Markup Language) ist eine Metasprache, mit der beliebige Dokumenttypen definiert werden können. Sie wurde 1998 standarisiert. Ein Dokumenttyp ist gekennzeichnet durch ein ihm eigenes Vokabular und eine Grammatik. Auf diese Weise können Inhalte in hierarchisch geschachtelter Form ausgedrückt werden. Da XML eine Metasprache ist, müssen das Vokabular und die Grammatik eines jeden Dokumenttyps erst definiert werden: dies geschieht über eine DTD (Document Type Definition). Eine DTD ist nicht notwendig, um ein XML-Dokument zu schreiben, erlaubt es aber, dieses auf syntaktische Richtigkeit zu überprüfen.

Ein jedes sprachliches Element (entspricht einem Wort des Vokabulars) eines XML-Dokumentes (Tag) beinhaltet selbst einen einfachen ASCII-String als Inhalt oder aber auch weitere Elemente, was eine hierarchische Ordnung ermöglicht. Um die Schachtelung eindeutig zu machen, muß jedes Element abgeschlossen werden.

Durch die Möglichkeit der Definition eines beliebigen, einer definierten Struktur genügenden Dokumenttyps eignet sich XML als Austauschformat für Datensätze. Zu beachten ist hierbei, daß nur geprüft werden kann, ob die Struktur des Dokumentes richtig ist. Zwei Fälle werden unterschieden:

- **Wohlgeformtheit:** Die Standards sind erfüllt. Es gibt genau ein abgeschlossenes Element, das alle anderen Elemente umfaßt (Wurzelelement). Die Elemente sind korrekt geschachtelt.

- **Gültigkeit:** Zusätzlich zur Wohlgeformtheit genügt das Dokument einer DTD, benutzt also nur bestimmte sprachliche Elemente und erlaubte Schachtelungen. Eine verlangte Mindestinformation ist ebenfalls vorhanden.

Jedes Element kann zudem mit zusätzlichen Informationen gekennzeichnet sein, die Attribute genannt werden.

XML trennt strikt den Inhalt, die Struktur und die Darstellung eines XML-Dokumenttyps. Bekommt eine Anwendung ein XML-Objekt, so muß

sie darauf vorbereitet sein: ohne geeignete Mittel kann sie die Daten nicht interpretieren, denn das Objekt ist zunächst nur ein ASCII-Text. Erst durch einen Parser können die Inhalte ihrer Struktur entsprechend aufbereitet werden.

Ein Parser liest das XML-Dokument und bereitet die Inhalte in geeigneter Form für die Anwendung auf. Ist das XML nicht wohlgeformt, so kann eine korrekte Funktion des Parsers nicht garantiert werden. Das Dokument muß zudem gültige Argumente besitzen, die der Parser als Schlüssel benutzt, um bestimmte Inhalte auszulesen und korrekt identifizieren zu können, was deren Weiterverarbeitung erst ermöglicht.

7.2.2 Sensordaten und XML

Die Sensoren speisen mit jedem Datensatz mehrere Informationseinheiten in das HCMS ein:

- **Bezeichnung:** Der eindeutige Name des Sensors, welche seine Aufgabenstellung repräsentiert. Nur ein Sensor gleichen Namens sollte auf einem Knoten laufen.
- **Inhalt:** Die Information, die der Sensor spezifisch zu ermitteln hat. Diese ist prinzipiell beliebiger Natur, entspricht aber bei der Verwendung von XML einem einfachen ASCII-Text.
- **Zeitpunkt der Messung:** Die Zeit, zu der der Inhalt durch den Sensor auf einem Knoten ermittelt wurde (Timestamp). Dient der zeitlichen Verlaufskontrolle von Parametern, die das System zu überwachen hat.
- **Rechnername:** Rechner, auf dem der Sensor läuft. Wichtig zur nachträglichen Zuordnung der gesammelten Daten.

Mit Hilfe von XML läßt sich dies so formulieren:

```
<?xml version=''1.0''?>
<MONDAT>
   <RECHNERNAME = ''Name''>
      <Sensorname TIMESTAMP = ''Zeit''>
         Inhalt des Sensors
      </Sensorname>
   </RECHNERNAME>
</MONDAT>
```

Das Wurzelelement MONDAT umschließt alle anderen Elemente. Es enthält eine Reihe von Elementen des Typs RECHNERNAME (hier nur eines), welche als Attribut den Namen des Rechners haben, auf dem der Sensor installiert ist. Dieses Element hat ebenfalls eine beliebige Anzahl an Tochterelementen, deren Name identisch mit der Bezeichnung des entsprechenden Sensors ist. Als Attribut führt jedes dieser Elemente die Timestamp mit. Der Inhalt der Tags, welche den Namen eines Sensors tragen, besteht aus den gesammelten Daten des Sensors selbst in ASCII-Form und enthält keine weiteren Unterelemente.

7.3 Schnittstellen

Die Aufteilung der Clusterknoten in Rechner, die zur Serverhierarchie gehören und solche, die mit Sensoren behaftet sind, aber nicht in eine Ebene der Hierarchie eingetragen sind, legt die Idee nahe, das Grundgerüst des HCMS aus zwei Programmen bestehen zu lassen: ein Programm (**Escalator/Consolidator**) läuft auf jedem Rechner der Hierarchie, das andere auf den übrigen Knoten (**Sensormanager**). Der Sensormanager beinhaltet eine Schnittstelle, welche die Sensoren zur Kommunikation mit dem Monitoring System benutzen (Kap 7.3.1. Der Escalator/Consolidator (im folgenden kurz Escalator) besitzt ein Modul, mit welchem er die Daten für eine SQL Datenbank vorbereiten und nach Wahl des DBMS einfügen kann, falls dies aufgrund fehlender Eskalierungsmöglichkeiten notwendig sein sollte. Ohne eine Wechslwirkung mit dem HCMS selbst können diese Informationen aus der Datenbank über eine einfache Schnittstelle abgeholt werden (Kap. 7.3.2). Ziel des Ganzen ist es, das HCMS sowohl nach oben hin möglichst weit zu entkoppeln, um es unabhängig von Programmen zu machen, welche die Sensordaten weiterverarbeiten. Eine leichte Integration beliebiger Sensoren mit einer möglichst einfachen Schnittstelle ist zudem erforderlich, um bestmögliche Flexibilität zu erreichen und den Administrator, der einen Sensor programmiert, zu entlasten.

7.3.1 Sensorschnittstellen

Die Sensoren sind eigenständige Prozesse, was es leicht möglich macht, neue Sensoren zu schreiben. Jeder Sensor baut eine Verbindung zu dem als Server fungierenden, auf dem selben Rechner wie der Sensor selbst laufenden Sensormanager auf. Der Sensormanager sammelt die Daten aller Sensoren und speichert sie bis zu der Eskalierung in die Hierarchie temporär; zudem verwaltet er alle mit ihm verbundenen Sensorprozesse, die bei diesem Konzept vollständig unabhängig voneinander sind.

Nach dem Aufbau einer Verbindung zu dem Sensormanager muß ein Sensorprozeß seine Informationen nur noch in geeigneter, für den Sensormanager lesbarer Form schicken. Dieses Format kann sehr einfach sein:

SENSORNAME = SENSORDATEN

Um die weitere Aufbereitung der Datensätze kümmert sich der Sensormanager: er versieht die Daten der Sensoren mit einer Timestamp und schickt alle gesammelten Informationen in regelmäßigen Abständen zu einem sich in der untersten Hierarchieebene befindlichen Server, nachdem sie konsolidiert und in ein XML-Austauschobjekt verpackt wurden. Diesen Server sucht der Consolidator nach dem Verfahren, das in Kap. 7.1 beschrieben ist.

7.3.2 Schnittstellen zu Datenbanken

Erkennt ein Escalator auf einem Rechner innerhalb der Serverhierarchie, daß eine Weitereskalierung der Daten augrund von Fehlfunktionen oder Limits unmöglich ist, so muß er die Daten seiner Clients in einer Datenbank speichern. Der Zugriff auf diese Daten muß in adäquater und leichter Form möglich sein. Manche DBMS bieten eine solche Funktionalität bereits an (Teil III). Auch wenn Datenbanken dieselbe Sprache verwenden (hier: SQL), so sind doch die Schnittstellen, mit denen Zugriff auf die Daten erreicht werden kann, unterschiedlich, was insbesondere für Programmierschnittstellen gilt. Es erweist sich also als notwendig, eine erweiterte Schnittstelle zu konstruieren, die nach Wahl modular das gewünschte DBMS verwendet, wobei aber die Sprachsyntax jeweils gleich bleibt. Werden künftig andere DBMS verwendet, so reicht es aus, Zusatzmodule zu schreiben, was zumindest bei SQL-Datenbanken leicht zu bewerkstelligen ist.

8 Zusammenfassung

Nachdem alle grundsätzlichen Designfragen diskutiert sind, kann das HCMS konstruiert werden. Abb. 14 zeigt, wie die Architektur des HCMS endgültig aussieht.

An dieser Stelle ist es wichtig zu überprüfen, ob die Architektur allen in Kap. 2.1 Anforderungen genügt.

Das Konzept erlaubt es, Sensoren dynamisch einzubinden. Eine einfache Schnittstelle ist vorhanden, die es erlaubt, Sensoren unabhängig voneinander in eigenen Prozessen ablaufen zu lassen. Diese Schnittstelle koppelt alle Sensoren eines Rechners lokal an einen Server (Sensormanager), der als wichtiges

Zusatzattribut alle Daten mit Timestamps versieht, um die genaue Rückverfolgung eines Ereignisses zu ermöglichen. Eine gut strukturierte Eskalierung wird durch die Verwendung von XML erreicht, wobei das Ankommen aller Daten auf dem Server der nächsten Ebene durch ein sicheres Netzwerkprotokoll erfolgen sollte. Da der Sensormanager die gleiche Methode wie der Escalator verwendet, um einen Server zu finden, werden seine Daten stets eine Ebene höher geschickt. So wird verhindert, daß diese Daten im Kreis fließen, weil sie erst einen Rechner in der untersten Hierarchieebene erreichen müssen. Durch Verwendung von DBMS wird eine Möglichkeit geschaffen, Daten dauerhaft und sicher abzulegen. Zugleich erlauben es die DBMS, beliebig selektierte Datensätze durch eine bereits vorhandene Schnittstelle einer anderen Anwendung zu übermitteln. Die DBMS stellen innerhalb der Architektur verteilte Datenspeicher dar. Nur auf diese Weise kann ein Monitoring System skalieren, da eine zentrale Datenspeicherung zwangsläufig ab einer bestimmten Clustergröße nicht mehr möglich ist. Dezentrale Datenspeicherung ist aber kein Nachteil, denn die verteilten Daten sind dennoch zentral verfügbar: die verteilten Datenbanken können durch eine gemeinsame Schnittstelle angesprochen werden. Obwohl Daten auf allen Rechnern der Serverhierarchie in einer Datenbank abgelegt werden können, sollte das HCMS versuchen, die Datensätze so zentral und so weit oben wie möglich in der Hierarchie abzulegen. Weiterhin ist noch dafür zu sorgen, daß Limits definiert werden können, welche Trigger für eine eigenständige Rekonfiguration und das load-balancing des HCMS darstellen. Dieser Aspekt der Skalierbarkeit des Gesamtsystems muß bei der Implementierung des Escalators bedacht werden.

Jeder Kasten in Abb. 14 entspricht einer funktionellen Einheit des Cluster Monitoring Systems. Auf jedem Knoten, der einen Escalator laufen hat, kann zusätzlich ein Sensormanager installiert sein, um auch auf Rechnern der Serverhierarchie Sensoren zu betreiben. Hier zeigt sich auch ein Problem der Architektur: da der Sensormanager die Daten stets eskaliert und nicht lokal sichert, sind diese Datensätze beispielsweise für eine lokale Fehlertoleranz nicht verfügbar. Dies kann nur umgangen werden, indem jeder Sensor Daten selbst lokal abspeichert und zugänglich macht. Es kann auch gewartet werden, bis diese Daten durch einen Escalator gespeichert wurden und somit global verfügbar sind. Hat die Serverhierarchie aber die Tiefe N und ist die Zeitscheibe T das Zeitintervall, nachdem die Daten auf jeder Ebene weitergeschickt werden, so kann es $N \cdot T$ dauern, bis diese Informationen abgefragt werden können.

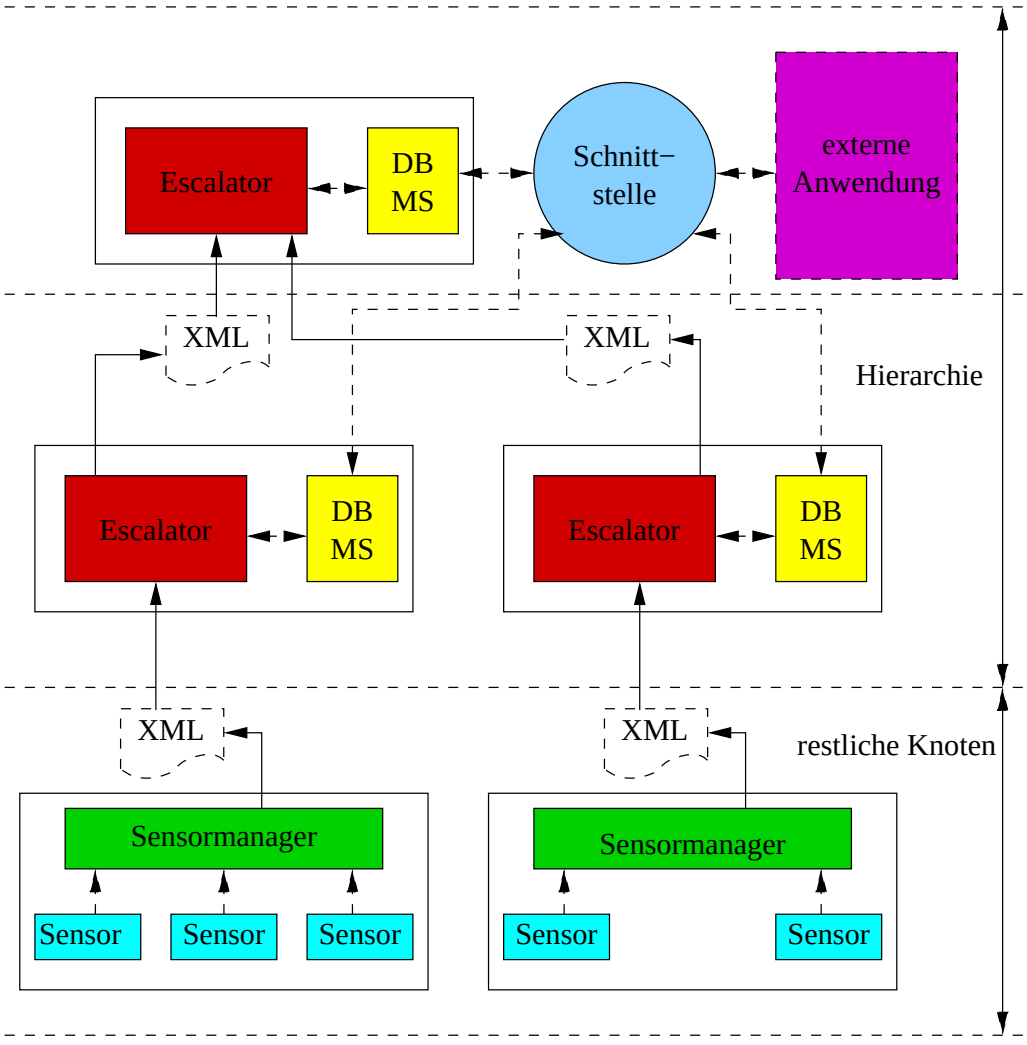

Abbildung 14: Ausschnitt eines Clusters mit installiertem HCMS

Teil V
Die Implementierung des HCMS

Nachdem alle konzeptionellen Designfragen geklärt waren (Teil IV), konnte das HCMS implementiert werden. Die hierfür verwendeten Bibliotheken, Tools und Kommunikationsmechanismen werden in Kap. 9 vorgestellt.

Das HCMS besteht im Wesentlichen aus zwei Programmen. Kap. 10 beschäftigt sich mit dem Programm, welches auf allen zu überwachenden Knoten läuft, Kap. 11 hingegen erläutert den Aufbau der Anwendung, welche auf jedem Knoten in der in Kap. 7.1 konstruierten Serverhierarchie installiert ist. Kap. 12 faßt kurz zusammen, wie das fertige HCMS aussieht und was es zu leisten vermag.

9 Allgemeines

Bei der Implementierung des Monitoring Systems stehen an erster Stelle die Fragen, welche Programmiersprache gewählt wird, wie die Kommunikation zwischen den Rechnern realisiert wird und welche Bibliotheken und externen Programme verwendet werden. Diese allgemeineren implemetierungsspezifischen Faktoren werden in diesem Kapitel besprochen.

9.1 C, C++ und Linux-Systemprogrammierung

Als Programmiersprache wird C/C++ verwendet. Für diese Wahl sprechen mehrere Gründe: sowohl C [Ker90] als auch C++ [Str98] sind sehr ausgereifte Sprachen, die bei Verwendung eines guten Compilers sowie sauberer Programmierung eine sichere und eindeutige Ausführung des compilierten Programms erlauben. Weiterhin ist das Programm deutlich schneller, als dies mit einer Sprache der Fall wäre, die ein ausführbares Programm mithilfe eines Interpreters zur Laufzeit erzeugt (z.B. Perl [PER]). Speziell mit C++ kann der Programmcode sehr effizient und prägnant gestaltet werden: durch den objektorientierten Ansatz (Klassenkonzept) und viele bereits vorhandene Standarddatentypen, die bereits mit geeigneten Methoden zu ihrer Bearbeitung ausgestattet sind und leicht erweitert bzw. abgeändert werden können, wird der Programmierer bei vielen Implementierungsfragen stark entlastet und kann sich somit mehr um anwendungsspezifische Probleme kümmern. Von großer Bedeutung ist zudem, daß der Source-Code von Linux selbst

größtenteils in C verfaßt ist. Entsprechend bietet das Betriebssystem (der Kernel) mächtige, ebenfalls in C geschriebene Programmbibliotheken [Her99] an, mit denen der Programmierer über eine direkte Schnittstelle mit dem Betriebssystem kommunizieren kann und somit leistungsfähige Anwendungen schreiben kann, die nur durch Verwendung dieser Kernelroutinen effizient oder überhaupt erst möglich sind.

9.2 TCP/IP und BSD-Sockets

Bei der Implementierung des HCMS ist es wichtig zu entscheiden, wie man die Kommunikation der verschiedenen Clusterknoten nach dem Client-Server-Prinzip realisiert.

Ein geeignetes und gut standarisiertes, auf jedem Linux-Rechner verfügbares Netzwerk-Protokoll stellt TCP/IP dar. IP [Bad01] (**Internet Protocol**) ist ein Datentransferprotokoll, welches auf einer Vielzahl von darunterliegenden Netzwerkhardware eine Netwerkverbindung aufbauen kann, aber unabhängig von dieser Hardware ist, was eine Netzwerkprogrammierung nur auf dieser Ebene erlaubt: Darunterliegende Protokollschichten werden mit Hilfe geeigneter Treiber durch das Betriebssystem selbst verwaltet. Mit IP ist ein bidirektionaler Datentranfer möglich. TCP/IP (**TCP: Transmission Control Protocol**) ist die auf IP aufbauende Netzwerkprotokollschicht, die im Unterschied zu UDP/IP (**UDP: User Datagram Protocol**) verbindungsorientiert ist. Dies bedeutet, daß der Empfänger von Datenpaketen deren Erhalt dem Sender bestätigt, so daß eine sichere Ablieferung aller Daten stets gewährleistet ist. Auch wenn dies mit einigem Overhead einhergeht, kommt für ein Monitoring System somit nur TCP/IP in Frage.

Um eine TCP/IP-Verbindung zwischen zwei Rechnern herzustellen, müssen die beiden Endpunkte der Verbindung bekannt sein. Diese sind jeweils eindeutig durch die **IP-Adresse** des Rechners sowie eine Zahl (**Port**) gekennzeichnet.

Der Linux-Kernel bietet mit BSD-Sockets eine Schnittstelle an, mit denen eine Interprozeß-Kommunikation zwischen Programmen auf verschiedenen Rechnern unter Verwendung von TCP/IP möglich ist. Ein Socket ist ein durch das Betriebssystem verwaltetes Objekt, das nach seiner Initialisierung durch eine Kennung (**Deskriptor**) eindeutig charakterisiert ist und beliebige Eingabe-/Ausgabeoperationen (**E/A-Operationen**) auf dem mit ihm verbundenen Gerät erlaubt, was in diesem Falle die Netzwerkkarte ist.

Der Aufbau einer Verbindung mit Sockets funktioniert folgendermaßen: Der Server erzeugt einen Socket mit dem Systemkommando *socket*(), wobei zugleich über angegebene Parameter festgelegt wird, daß dieser Socket eine Netzwerkverbindung über das TCP/IP-Protokoll beschreibt. Ein fester Port

wird für eingehende Verbindungsanforderungen mittels des Befehls *bind*() reserviert. Auf diesem Port wird mit *listen*() auf eine solche Anforderung gewartet. Meldet sich ein Client, so liefert *accept*() einen neuen Socketdeskriptor[18] zurück, über den die weitere Kommunikation stattfindet.

Auch der Client muß einen Socket mit dem Befehl *socket*() erzeugen. Als Nächstes schickt er aber eine Verbindungsanfrage an den Server, wofür das Kommando *connect*() Verwendung findet. Akzeptiert der Server das Ansuchen, liefert *connect*() einen Deskriptor zurück: die Kommunikationsendpunkte sind etabliert, ein Datenaustausch kann stattfinden.

9.3 E/A-Multiplexing mit SELECT

Im Allgemeinen ist davon auszugehen, daß jeder Server mehrere Clients besitzt. Das ist im Grunde unproblematisch: jeder Client meldet sich bei dem Server auf dem Port, auf dem dieser mit einem *listen*() lauscht, woraufhin sowohl der Server als auch der Client einen Socketdeskriptor erzeugen, über welche beide kommunizieren können. Der Server besitzt nun aber im Falle mehrerer Clients eine entsprechende Anzahl an Deskriptoren, die alle jeweils einen Datenkanal repräsentieren, über den potentiell Datenpakete eintreffen können. Er muß also eine Möglichkeit haben, alle Deskriptoren nach neuen angekommenen Datenpaketen zu befragen und diese gegebenenfalls jeweils mit einer Leseoperation abholen.

Will man dies innerhalb eines Prozesses machen und nicht mehrere Prozesse verwenden, die mittels des $fork()$-Mechanismus je einen Client bedienen[19], so bietet sich die Systemfunktion *select*() an. Diese Funktion legt ein Bitmuster an, wobei ein Bit jeweils genau einen Client repräsentiert. Nach dem Aufruf von *select*() kann am Zustand eines Bits abgelesen werden, ob von dem entsprechenden Client Daten gesandt wurden, so daß alle Informationen abgegriffen und gleichzeitig jede Teilinformation eindeutig einem der Clients zugeordnet werden kann. Besonders wichtig ist, daß *select*() es erlaubt, Sockets ohne deren Blockierung abzufragen (non-blocking), falls der entsprechende Kanal keine Daten enthält.

9.4 Der XML-Parser TinyXML

Die entwickelten Programme müssen die Fähigkeit besitzen, Informationen aus empfangenen XML-Objekten auszulesen, um diese in geeigneter Form

[18]Dieser ist ein abstraktes Objekt, das durch elementare Funktionen angesprochen werden kann. Er repräsentiert die Verbindung.

[19]Dieses Konzept ist schwierig zu realisieren, da mindestens ein Prozess alle Daten kennen muß, also eine Interprozeß-Kommunikation notwendig wird.

weiterverarbeiten zu können. Dazu ist ein XML-Parser notwendig (siehe Kap. 7.2.1).

Das HCMS verwendet den Parser TinyXML [TIN]. TinyXML kann nicht überprüfen, ob ein XML-Dokument einer DTD genügt, es muß also dafür gesorgt werden, daß das zu parsende Dokument passende Tags in einer korrekten hierarchischen Gliederung enthält. Da die XML-Dokumente aber nur von dem HCMS selbst erzeugt werden, kann man neben der Wohlgeformtheit auch von einer Gültigkeit der empfangenen Objekte ausgehen.

Damit auf das Dokument zugegriffen werden kann, muß es in einer Datei gespeichert sein. Nachdem es durch eine Parserfunktion geöffnet wurde, können Elemente beliebiger Hierarchieebene innerhalb der XML-Struktur mittels geeigneter Bibliotheksfunktionen angesprochen werden, wobei eine exakte Filterung über den optional verwendbaren Namen des anzusprechenden Tags ebenfalls möglich ist. Desweiteren kann aber auch iterativ in einen Zweig der Hierarchie hinabgestiegen und nach dem Auffinden des passenden Elements an eine andere Stelle in dem Zweig zurückgekehrt werden. Auch das Auslesen von Attributen stellt kein Problem dar, sei es zur Filterung, zum Auffinden von bestimmten Informationen oder aber zum Auslesen einer Informationseinheit selbst.

Durch den Parser ist es zusätzlich möglich, die Konfiguration des HCMS, d.h. wichtige Betriebsparameter und die Struktur der Serverhierarchie, in Form von XML-Dateien durchzuführen, was einen guten Überblick und die leichte, auch automatisierte Änderung dieser Parameter erlaubt.

9.5 Datenkompression mit der Bibliothek ZLIB

Mit einem Malus ist die Verwendung eines XML-Dokuments als Datenaustauschformat immer noch behaftet: es ist relativ groß, denn jedes Zeichen des ASCII-Textes fordert ein Byte Speicherplatz. Der Overhead ist auch nach der Konsolidierung der Daten relativ groß. Es ist aber festzustellen, daß das XML-Objekt in konsolidierter Form viele ähnliche Zeichenfolgen aufweist, die zwar keine Redundanzen sind (sonst wäre eine weitere Konsolidierung möglich), dennoch aber die Verwendung eines Kompressionsverfahrens zur Verringerung der Datenmenge während des Transports über das Netzwerk nahelegen.

Für das HCMS wird die Kompressionsbibliothek zlib [ZLI] benutzt. Sie basiert auf einem Kompressionsverfahren, welches im Falle ähnlicher Strukturen innerhalb eines beliebigen Dokuments eine starke Verkleinerung desselben erlaubt. Welcher Art dieses Kompressionsverfahren ist, konnte nicht herausgefunden werden. Besonders effektiv zeigt sich die Kompression, wenn schon die Daten vieler Sensoren angefallen sind, also weiter oben in der Ser-

verhierarchie: die XML-Dokumente werden um 90 Prozent und mehr in ihrer Größe reduziert. Das ist eine ideale Ausgangslage, denn gerade an diesen Stellen ist die Netzlast besonders kritisch, da schon eine starke Bündelung vieler kleiner Datenströme stattgefunden hat: große Datenmengen werden von den Clients zu ihren Servern geschoben. Es bleibt aber zu beachten, daß hierdurch das HCMS keineswegs beliebig skaliert, denn einerseits stellt ein Einfügen der Datensätze in die Datenbank hohe Anforderungen an den Server, andererseits müssen die Daten zudem auf jeder Hierarchieebene vor ihrer Weiterverarbeitung erst dekomprimiert und bearbeitet werden. Die Last auf dem Netzwerk wird also teilweise auf die beteiligten Rechner der Serverhierarchie selbst übertragen. Dies bietet aber den Vorteil, daß Rechner außerhalb der Serverhierarchie, die Netzwerkbandbreite zur Lösung verteilter Aufgaben benötigen, nicht von einer erhöhten Netzlast betroffen sind, was ihre Effektivität stark einschränken würde.

9.6 Datentransfer-Protokoll

Werden die komprimierten XML-Daten von einem Server empfangen, so muß dieser Speicherplatz für sie zur Verfügung stellen, zusätzlich aber auch für die dekomprimierten Daten, welche mit Funktionen der *zlib*-Bibliothek entpackt werden. Es wäre äußerst ungünstig, dies in statischer Weise zu tun: sind die empfangenen oder entpackten Daten größer als der ihnen bereitgestellte Puffer, kommt es zu undefinierten Fehlfunktionen. Die Puffer sehr groß zu machen, ist ebenfalls nicht sinnvoll, denn dies käme in den meisten Fällen einer Speicherplatzverschwendung gleich und würde prinzipbedingt auch nicht beliebig skalieren. Problematisch ist auch, daß es passieren kann, daß der Server mit einem *read*()-Aufruf mehrere durch Clients mit einem *write*()-Aufruf abgesetze XML-Dokumente entgegennimmt: das resultierende XML-Objekt ist hierbei nicht mehr wohlgeformt, denn es enthält mehrere Wurzelelemente.

Um diese Probleme zu vermeiden, wird bei dem Datentransfer ein einfaches Protokoll verwendet. Ein Client schickt bei einer anstehenden Datenübertragung im ersten Schritt eine Struktur fester Länge zum Server, die diesem mitteilt, wie groß sowohl der unkomprimierte als auch der komprimierte Datensatz sind. Der Server kann dynamisch Speicherplatz in entsprechender Größe reservieren, worauf die eigentlichen Daten übertragen werden. So ist eine gute Effizienz im Umgang mit Speicherplatz und eine Verhinderung von Fehlfunktionen aufgrund zu kleiner Puffer gewährleistet.

9.7 Ermittlung der Serverhierarchie: die getserver()-Funktion

Die erste Aufgabe eines jeden Clients, sowohl innerhalb als auch außerhalb der Serverhierarchie, ist es, einen geeigneten Server zu finden (siehe auch Kap. 7.1). Bei der Implementierung des HCMS wurde hierzu die Funktion *getserver*() entwickelt, die ohne Parameter aufgerufen wird und im Erfolgsfall die IP-Adresse eines Servers zurückliefert, mit dem der Client daraufhin eine feste Verbindung eingehen kann. Hierzu wird zunächst aus der Konfigurationsdatei **XMLMonHierarchy** ausgelesen, welche Rechner zu der Hierarchie gehören und auf welchem Level sie sich jeweils befinden. Diese Konfigurationsdatei könnte so aussehen:

```
<?xml version=''1.0'' encoding=''iso-8859-1'' ?>
<configuration>
        <numberOfLevels>3</numberOfLevels>
        <Level level_number = ''0''>
                <numberOfIPs>3</numberOfIPs>
                <IP>10.0.2.10</IP>
                <IP>10.0.2.15</IP>
                <IP>10.0.2.14</IP>
        </Level>
        <Level level_number = ''1''>
                <numberOfIPs>2</numberOfIPs>
                <IP>10.0.2.17</IP>
                <IP>10.0.2.16</IP>
        </Level>
        <Level level_number = ''2''>
                <numberOfIPs>1</numberOfIPs>
                <IP>10.0.2.26</IP>
        </Level>
</configuration>
```

Mit aufsteigendem Level befinden sich die Server in der Hierarchie weiter oben.

Abb. 15 zeigt schematisch, wie diese Funktion einen Server findet.

Hat der Client seine IP herausgefunden und die XML-Konfigurationsdatei, in der die Serverhierarchie spezifiziert ist, gelesen, so weiß er, in welcher Ebene der Server liegt, der für ihn zuständig ist. Zum Auslesen der Konfiguration wird TinyXML verwendet.

Der Client baut vorübergehend eine Verbindung zu allen Servern in der gefundenen Ebene auf. Gelingt dies bei einem Server nicht, so scheidet er

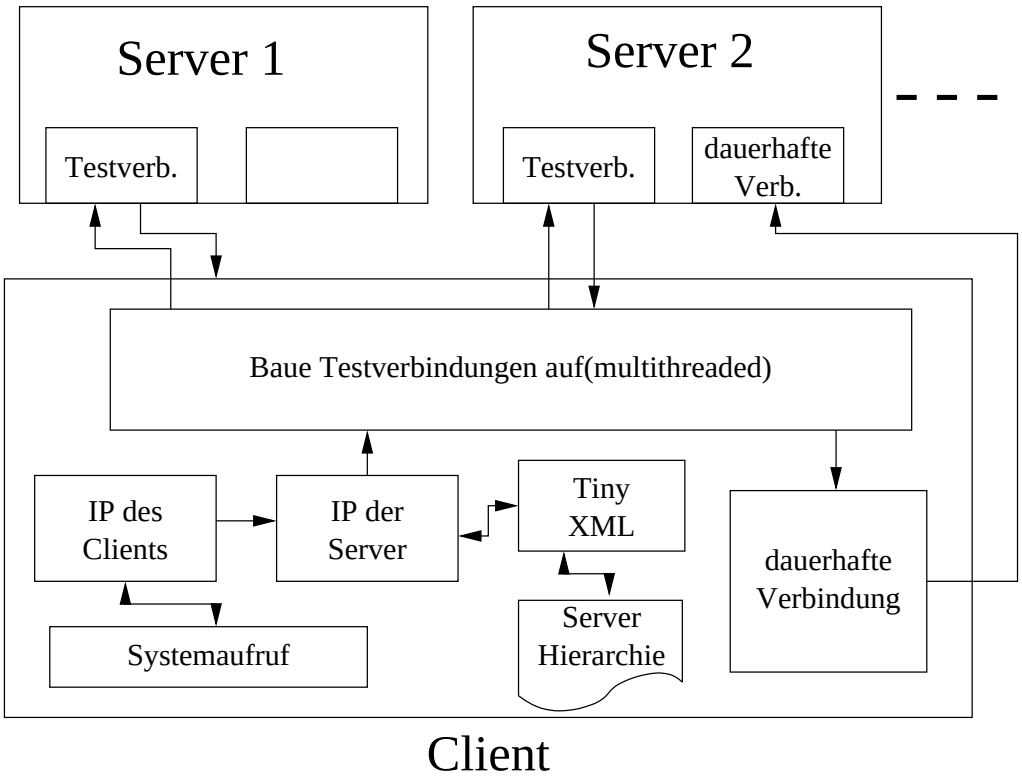

Abbildung 15: Auffinden eines geeigneten Servers in der Hierarchie

von vornherein aus. Die Verbindung erfolgt, indem der Client sich auf dem Port meldet, auf dem der Server lauscht, so daß dieser einen neuen Socketdeskriptor für ihn erstellen kann. Der Server weiß aber, daß die Verbindung nur vorübergehender Natur ist, da der Client sich über ein eigenes Protokoll mit einem Authentifizierungspaßwort meldet, das dem Server anzeigt, daß der Client keine dauerhafte Verbindung wünscht, sondern nur eine Rückmeldung über die Verfügbarkeit des Servers haben möchte. Diese Rückgabe ist eine einfache Zahl, deren Größe angibt, wie gut der Server einen neuen Client noch verwalten kann. Diese Zahl kann der Einfachkeit halber die Anzahl der bereits fest verbundenen Clients sein: je niedriger der Rückgabewert, desto besser ist der Server als dauerhafter Verbindungsendpunkt für den anfragenden Client geeignet. Letztlich baut der Client also eine dauerhafte TCP/IP-Verbindung mit dem Server auf, der den kleinsten Wert zurückgegeben hat. Kann kein Server gefunden werden, zeigt sich dies durch Rückgabe eines NULL-Pointers[20], der abgefangen werden muß.

[20]Ein **Pointer** ist ein Zeiger auf einen Speicherbereich. Ein **NULL-Pointer** zeigt auf eine undefinierte Stelle.

Damit die Suche nach einem Server optimiert und schnell abläuft, wird die Serverliste nicht sequentiell durchgangen, sondern die vorübergehenden Verbindungen werden parallel mit Threads aufgebaut. Je ein erzeugter Thread der aufrufenden Funktion *getserver*() wird von einem Server bedient und liefert anschließend sein Ergebnis an den aufrufenden Prozeß zurück. Nach dieser Prozedur werden alle temporären Verbindungen geschlossen, und es kann mit dem der zurückgelieferten IP-Adresse zugehörigen Rechner eine dauerhafte Verbindung eingegangen werden.

10 Clientebene: der Sensormanager

Dieses Kapitel erläutert in Grundzügen den Aufbau des Sensormanagers. Der Sensormanager dient dazu, die Sensoren auf einem Knoten lokal zu verwalten, wobei dem Programmierer Funktionen angeboten werden, mit denen er einen selbstgeschriebenen Sensor an den Sensormanager ankoppeln kann. Weiterhin bereitet der Sensormanager die Daten der Sensoren mit Zusatzinformationen auf, konsolidiert sie und reicht sie schließlich an die die Serverhierarchie weiter. Der Sensormanager ist sowohl ein Server als auch ein Client. Für die lokal vorhandenen Sensoren ist er Server, gleichzeitig ist er Client eines Rechners in der Serverhierarchie.

In Abb. 16 ist der Aufbau des Sensormanagers im Überblick zu sehen.

10.1 Einlesen der Konfiguration

Nach dem Start des Sensormanagers liest dieser zunächst aus einer XML-Datei (**XMLMonConfig**) seine Konfiguration ein. Diese besteht primär aus zwei Parametern: der Port, auf dem er dem Server eine Verbindungsanforderung schicken kann (Escalator_Port), und eine Zeitscheibe, die das Intervall angibt, in dem er die konsolidierten Daten als komprimiertes XML-Objekt zu seinem Server schickt (Time_Slice). Ist einer der beiden der Parameter in der Datei nicht vorhanden, findet jeweils ein Standardwert Verwendung.

Die Konfigurationsdatei sieht folgendermaßen aus:

```
<?xml version=''1.0'' encoding=''iso-8859-1'' ?>
<Config>
        <Escalator_Port>2002</Escalator_Port>
        <Consolidator_Port>3003</Consolidator_Port>
        <Time_Slice>10</Time_Slice>
        <Lower_Limit>1500</Lower_Limit>
        <Upper_Limit>16000</Upper_Limit>
        <Queue_Size><Queue_Size>
        <DBMS>PostgreSQL<DBMS>
        <DB_Name>test<DB_Name>
        <DB_Table>mondat<DB_Table>
        <DB_User>hhillesh<DB_User>
        <DB_Passwd>wrdlbrmpft<DB_Passwd>
</Config>
```

Für den Sensormanager sind hierbei nur die zwei genannten Variablen sowie der Consolidator_Port von Bedeutung.

Der Sensormanager ist aber erst betriebsbereit, wenn eine dauerhafte Verbindung zu einem Server eingerichtet ist. Die möglichen Server bekommt er aus einer weiteren Konfigurationsdatei (**XMLMonHierarchy**). Mit Hilfe der Funktion *getserver*() (Kap. 9) wählt er einen der Server aus.

10.2 Kommunikation der Sensoren mit dem Sensormanager

Von zentraler Bedeutung für den Sensormanager ist, daß er dynamisch eine Vielzahl von Sensoren verwalten kann. Eine Programmierschnittstelle ist ebenfalls notwendig, damit ein jeder Sensor an den Sensormanager ankoppeln kann.

10.2.1 Dynamische Verwaltung der Sensoren

Der Sensormanager läuft in nur einem Prozeß. Dieser Weg zeigt sich als der beste, da andernfalls aufwendige Interprozeß-Kommunikationsmechanismen notwendig wären. Eine parallele Ausführung verschiedener Programmteile ist weiterhin unerwünscht, da nur bei sequentiellem Betrieb eine Konsolidierung der Daten und definierte Verpackung in ein XML-Dokument ohne komplexe Locking-Mechanismen[21] möglich ist.

[21] **Locking-Mechanismen** finden Verwendung, wenn Programmteile parallel abgearbeitet werden. An kritischen Stellen wird die Ausführung eines Teils so lange blockiert, bis ein oder mehrere andere Programmteile diese Blockierung wieder aufheben.

Die Verwaltung aller Sensoren findet also mittels E/A-Multiplexing über den *select*()-Mechanismus statt (Kap. 9). Die Daten der Sensoren können so strukturiert und eindeutig an der Schnittstelle abgegriffen werden, bevor sie nach Ablauf der Zeitscheibe weiterverarbeitet werden. Es ist dabei stets gewährleistet, daß die Datenpakete in einer sinnvoller Syntax und in adäquater Form vorliegen.

10.2.2 Schnittstelle zwischen Sensoren und Sensormanager

Jeder Sensor benutzt den Sensormanager als Server. In der Konfigurationsdatei XMLMonConfig ist in dem Element Consolidator_Port nachzulesen, auf welchem Port der Sensormanager auf eingehende Verbindungsanforderungen wartet. Erlaubt der Sensormanager die Verbindung, so liefert ein *connect*() des Sensorprogramms einen gültigen Socketdescriptor zurück, über den er nun Daten zum Sensormanager schicken kann. Das Protokoll zum Datentranfer ist dabei sehr einfach: Es wird einfach eine Zeichenkette versandt, die mindestens ein Gleichheitszeichen enthält. Die Zeichen links des ersten auftretenden Gleichheitszeichens bilden den Namen des Sensors, die rechts davon die zu übermittelnde Information selbst (siehe auch 7.3.1).

Auf diesem Wege läßt sich der Sensor in einer beliebigen Programmiersprache schreiben, die TCP/IP-Unterstützung bietet. Da aber insbesonders die Abfrage des $ConsolidatorPort$ in der Konfigurationsdatei $MonConfig$ ohne Parser ein schwieriges Unterfangen darstellt, scheint diese Methode nur leicht möglich zu sein, wenn die Standardwerte in der HCMS-Konfiguration zum Tragen kommen.

Damit solche Probleme zumindest in C/C++ nicht auftauchen, wurde eine einfache Sammlung von Funktionen als Schnittstelle geschrieben, die dem Programmierer des Sensors diese Arbeit abnehmen.

Es sind drei Funktionen vorhanden. Eine Funktion meldet einen Sensor bei dem Sensormanager an, eine weiter schließt diese Verbindung. Die letzte Funktion sorgt für den Transfer der eigentlichen Daten.

10.2.3 Aufbereitung der Datensätze

Die Datensätze der Sensoren, die bei dem Sensormanager eintreffen, sind noch nicht ausreichend mit Attributen versehen, um konsolidiert und in die Serverhierarchie eskaliert werden zu können. Jedes eintreffende Datumspaar wird noch zusätzlich mit einer Timestamp und der IP des Hosts versehen, auf dem die Messung stattfand. Dies ist notwendig, um eine gezielte Rückverfolgung eines Ereignisses nach Abfrage der Datenbanken zu ermöglichen (siehe Kap. 7.2.2).

Es werden nur Sensoren mit Namen akzeptiert, die über eine letzte, dritte Konfigurationsdatei (XMLMonTags) registriert sind. Ist der Name eines Sensorpakets nicht identisch mit einem der aufgeführten Tags, wird dieser ignoriert und nicht weiter verarbeitet. Dies erlaubt eine Validierung der Sensoren oder eine Filterung bestimmter Sensoren, die nicht registriert und gespeichert werden sollen.

10.3 Konsolidierung der Daten

Nach Ablauf der Zeitscheibe werden die gesammelten, in einem programminternen Format gespeicherten Daten gebündelt und unter der Vermeidung unnötiger Redundanzen in ein XML-Dokument gepackt. Die hierbei verwendete Funktion gewährleistet für den Fall, daß die Daten ohne einen Fehler eskaliert werden, durch einen strukturierten, sequentiellen Aufbau des Dokumentes von vornherein dessen Wohlgeformtheit und Gültigkeit. Die nächste Serverinstanz kann das XML-Objekt also parsen und auswerten.

Ein von dem Sensormanager erzeugtes XML-Dokument sieht aus wie folgt:

```
<?xml version=''1.0''?>
<MONHOSTS>
   <MONDAT HOST = ''IP des Hosts''>
      <Sensorname1 DATE =''aktuelles Datum'' TIME = ''Messzeitpunkt''>
         Inhalt des Sensors
      </Sensorname1>
      <Sensorname2 DATE =''aktuelles Datum'' TIME = ''Messzeitpunkt''>
         Inhalt des Sensors
      </Sensorname2>
      <Sensorname3 DATE =''aktuelles Datum'' TIME = ''Messzeitpunkt''>
         Inhalt des Sensors
      </Sensorname3>
      ...
      ...
   </MONDAT>
</MONHOSTS>
```

Die Datensätze müssen dabei nicht alle unterschiedlich sein, sondern können auch von einem Sensor zu unterschiedlichen Meßzeitpunkten stammen.

An diesem Beispiel zeigt sich auch, warum eine Komprimierung aufgrund ähnlicher Strukturen innerhalb des Dokumentes so effektiv, aber aufgrund der Größe auch notwendig ist.

10.4 Eskalierung der Daten

Ist das XML-Dokument erstellt, so wird es komprimiert und weitereskaliert. Der Sensormanager braucht dabei keinerlei Parameter, die sein Verhalten in einer bestimmten Form steuern: die Daten müssen in jedem Fall zumindest die unterste Hierarchieebene der Monitoring Server erreichen, sonst werden sie an keiner Stelle abgespeichert und sind somit verloren.

Es soll nicht unerwähnt bleiben, daß der Sensormanager zunächst weit komplexer aufgebaut und implementiert war. In dieser älteren Version gab es keine feste Zeitscheibe, nach deren Ablauf die Daten eskaliert wurden, statt dessen wurde bei jedem Sensor gemessen, mit welcher Frequenz er sendet. War dieser Wert bekannt, so wurden Sensoren ähnlicher Sendefrequenz zu einer Gruppe zusammengefaßt. Jede Gruppe bekam so ihre eigene Zeitscheibe, mit der sie unabhängig von anderen die Datenpakete weiterschickte. Mit diesem Konzept konnten auch Sensoren, die ein definiertes Vielfaches ihrer Gruppenzeitscheibe keine Daten mehr gesandt hatten, als obsolet betrachtet und somit von dem Sensormanager abgetrennt werden.

Dieses Konzept hatte aber den schwerwiegenden Nachteil, daß es nur bei festfrequenten Sensoren vernünftige Resultate lieferte. Ist die Frequenz eines Sensors nicht fest, so führt dies zwar zu keinen Programmfehlern, doch es kann passieren, daß gemessene Werte "verschluckt" werden und somit unberücksichtigt bleiben. Ein anderer möglicher Fall führt dazu, daß ein und derselbe Wert überflüssigerweise mehrfach versandt wird. Besonders schwer wiegt bei diesem Design zudem, daß viele kleine XML-Dokumente in das Netzwerk gesckickt werden, was zu einem großen Overhead durch das TCP/IP-Protokoll, den XML-Header und einen geringen Kompressionsgrad führt. Manchmal ist weniger eben doch mehr.

10.5 Rekonfiguration

Gelingt es dem Sensormanager nicht, einen Consolidator/Escalator in der untersten Serverhierarchieebene zu erreichen, so versucht er zunächst, die Konfiguration neu einzulesen. Nicht definierte Parameter werden dabei mit Standardwerten belegt. Tritt bei einem erneuten Verbindungsversuch kein Erfolg ein, so beendet sich der Sensormanager mit einer Fehlermeldung, da er die Daten eskalieren können muß: er stellt keine eigenständig lauffähige Funktionseinheit dar.

Bricht ein Sensor die Verbindung zu dem Sensormanager ab, wird dies ignoriert. Die Funktion des Sensormanagers selbst und aller anderer Sensoren ist davon in keinster Weise beeinträchtigt.

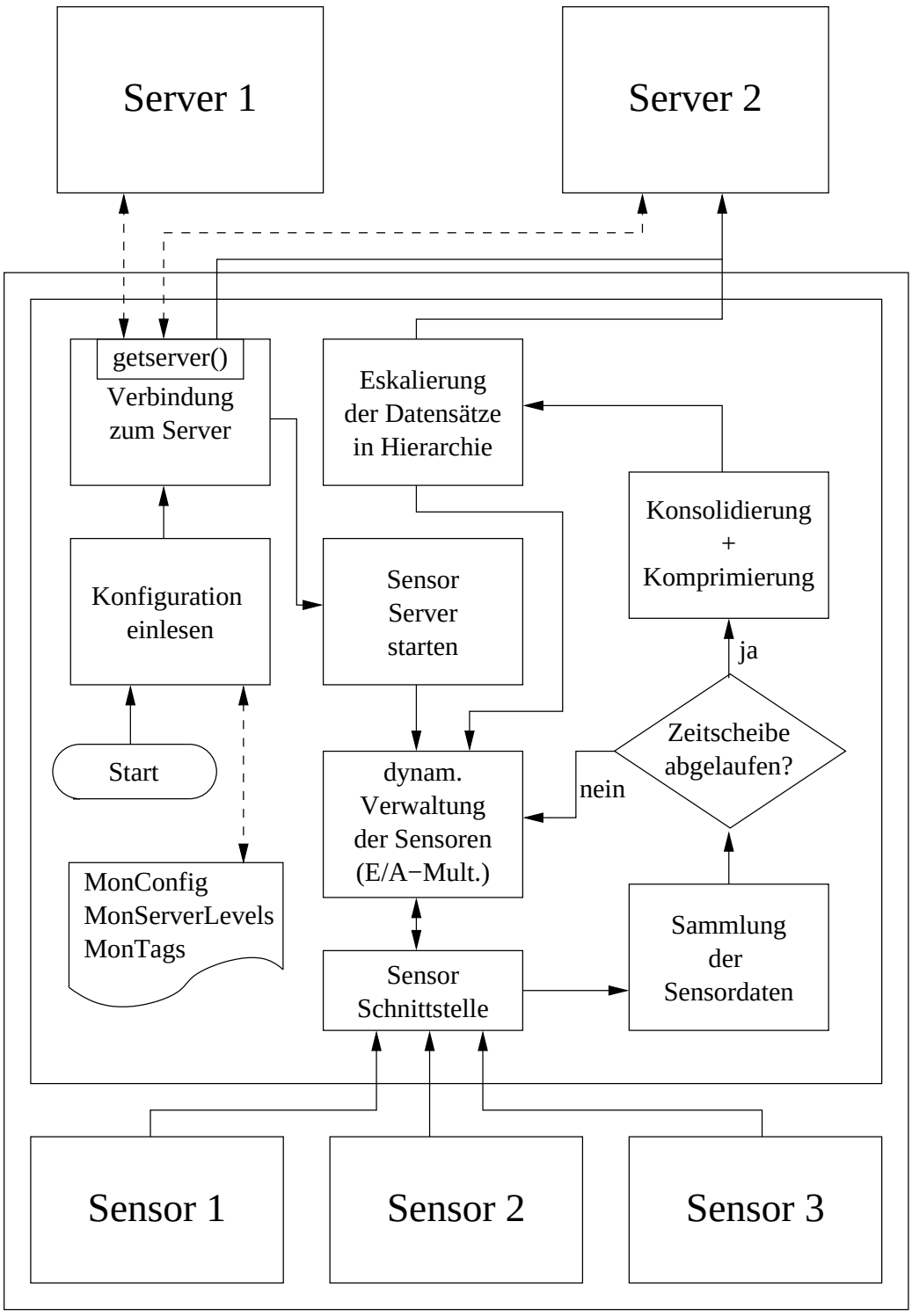

Abbildung 16: Aufbau des Sensormanagers

11 Serverebene: der Consolidator/Escalator

Der Consolidator/Escalator (im Weiteren kurz Escalator genannt) ist das Programm des HCMS, das auf jedem Server in der Hierarchieebene abläuft. Es nimmt konsolidierte, komprimierte Sensordaten mehrerer Clients in einem XML-Format entgegen. Die Daten werden geparst, in einem programmeigenen Format zwischengespeichert und nach Ablauf einer Zeitscheibe konsolidiert. Abhängig von den Möglichkeiten und der Größe der Daten werden diese in komprimierter Form weitereskaliert oder aber lokal in einer SQL-Datenbank abgelegt.

Abb. 17 zeigt den schematischen Aufbau des Consolidators/Escalators.

11.1 Kommunikationsschnittstellen

Der Escalator ist wie der Sensormanager zugleich Server und Client. Er fungiert als Server für ein Escalator-Programm der nächsttieferen Hierarchieebene. In der untersten Ebene nimmt er dagegen die Daten eines oder mehrerer Sensormanager entgegen. Dies ist möglich, da sowohl ein Escalator als auch ein Sensormanager Datenpakete in Form von XML-Dokumenten versenden, die gleiches Vokabular, die gleiche Grammatik und den selben hierarchischen Aufbau verwenden: inhärent genügen also alle XML-Dokumente einer DTD, obwohl niemals aufgrund ungenügender Fähigkeiten des XML-Parsers deren Gültigkeit geprüft wird.

Client ist der Escalator für ein identisches Programm in der nächsthöheren Hierarchieebene, falls eine Eskalierung möglich ist.

Die Schnittstellen werden automatisch aufgebaut, es ist also nicht notwendig, eine Programmierschnittstelle bekannt zu machen. Das XML-Format ist dem Escalator bekannt, die Hierarchie selbst wird mit der Funktion *getserver*() und der Konfigurationsdatei XMLMonConfig aufgebaut, die die Ports der Verbindungsendpunkte spezifiziert. Verbindungsersuchen externer Programme werden wieder durch ein Authentifizierungsmodul abgelehnt.

11.2 Sammeln der Daten

Auch die Verwaltung der Clients ist analog zu dem Sensormanager: mit Hilfe der Systemfunktion *select*() wird eine Liste von Socketdeskriptoren dynamisch verwaltet und auf ankommende Daten überprüft. Kommt über den Socketdeskriptor, welcher den Port repräsentiert, auf dem der Server auf Verbindungsanfragen lauscht, eine authentifiziertes Ansuchen, so wird ein neuer, einem zusätzlichen Client entsprechenden Socketdeskriptor der vorhandenen Liste angefügt.

Die empfangenen Daten werden entpackt und gespeichert, wobei die Gesamtgröße dieser Datensätze seit dem letzen Ablauf der Zeitscheibe registriert wird.

11.3 Konsolidierung der Daten

Nach Ablauf der Zeitscheibe werden alle empfangenen XML-Dokumente geparst und die gewonnenen Informationen konsolidiert und in ein neues XML-Objekt gepackt. Im Unterschied zu dem Sensormanager ist es hierbei nicht notwendig, die resultierenden Datensätze mit Zusatzinformationen zu versehen, da die Sensordaten bereits vollständig charakterisiert sind.

Die Struktur des erzeugten XML-Dokuments ist analog zu dem eines Sensormanagers, bis auf einen Unterschied: Unter dem Wurzelelement kann es nun mehrere Elemente des Typs MONDAT geben, wobei diese paarweise verschiedene HOST-Attribute besitzen:

```
<?xml version=''1.0''?>
<MONHOSTS>
   <MONDAT HOST = ''IP des Hosts eines Clients mit Sensoren''>
      <Sensorname1 DATE =''aktuelles Datum'' TIME = ''Messzeitpunkt''>
         Inhalt des Sensors
      </Sensorname1>
      <Sensorname2 DATE =''aktuelles Datum'' TIME = ''Messzeitpunkt''>
         Inhalt des Sensors
      </Sensorname2>
      <Sensorname3 DATE =''aktuelles Datum'' TIME = ''Messzeitpunkt''>
         Inhalt des Sensors
      </Sensorname3>
      ...
      ...
   </MONDAT>
   <MONDAT HOST = ''IP des Hosts eines anderen Clients mit Sensoren''>
      <Sensorname4 DATE =''aktuelles Datum'' TIME = ''Messzeitpunkt''>
         Inhalt des Sensors
      </Sensorname4>
      <Sensorname5 DATE =''aktuelles Datum'' TIME = ''Messzeitpunkt''>
         Inhalt des Sensors
      </Sensorname5>
      <Sensorname6 DATE =''aktuelles Datum'' TIME = ''Messzeitpunkt''>
         Inhalt des Sensors
      </Sensorname6>
      ...
      ...
   </MONDAT>
</MONHOSTS>
```

Die Namen der Sensoren sowohl verschiedener MONDAT-Elemente als auch unterhalb eines solchen Elements können dabei identisch sein. Dieser Zustand tritt im ersten Fall auf, falls die Daten von Sensormanagern stammen, die auf verschiedenen Rechnern Sensoren mit gleichem Namen verwalten, im zweiten Fall dagegen ist die Meßfrequenz eines Sensors größer als die Frequenz, die der Aktionszeitscheibe eines Sensormanagers entspricht.

11.4 Rekonfiguration und intelligente Datenweiterverarbeitung

Nach der Konsolidierung muß der Escalator entscheiden, wie er weiter mit den Daten verfährt. Dies ist nicht so einfach wie bei dem Sensormanager, der in jedem Falle seine Daten weitereskalieren muß, um eine Speicherung der gesammelten Sensordaten zu ermöglichen.

Grundsätzlich stehen dem Escalator zwei Optionen zur Verfügung: er kann die Daten weitereskalieren oder sie stattdessen lokal in einer Datenbank speichern.

Zuerst überprüft der Escalator, ob bei fehlender oder abgebrochener Verbindung zur nächsten Hierarchieebene eine neue Verbindung zu einem Server in derselben erzeugt werden kann. Ist das unmöglich, wird der MAIN_SERVER-Modus aktiv: die Daten werden in der lokalen Datenbank dauerhaft abgelegt. Dieser neue Zustand ist irreversibel und bleibt bis zu einem Neustart des Programms erhalten.

Selbst wenn ein Server vorhanden ist, kann es nötig sein, eine Weitereskalierung der Daten zumindest temporär zu unterbinden. Das ist der Fall, wenn die Gesamtgröße der empfangenen Daten innerhalb eines Zeitscheibenintervalls gegebene Limits überschreitet. Diese Limits kommen v.a. durch die beschränkt mögliche Füllrate in ein DBMS zustande, aber auch durch beschränkte Netzwerkbandbreiten und Rechnerressourcen. Da es ein schwieriges Unterfangen darstellt, diese Limits zur Laufzeit herauszufinden, sind sie in der Konfigurationsdatei XMLMonConfig angegeben und müssen aufgrund von Erfahrungswerten gesetzt werden.

Zwei Limits sind spezifiziert: Wird der Wert des Elements Lower_Limits überschritten, so werden die Verbindungen zu dem zuständigen Server gekappt und die Daten in der Datenbank abgelegt. Liegt die Größe der Datensätze zudem über Upper_Limits, so wird zusätzlich der Client mit der höchsten Last abgekoppelt und keine neuen Verbindungen akzeptiert, bis dieser kritische Bereich verlassen ist. Diese Maßnahmen sind aber reversibel: in einer Queue, der Queue_Size Elemente enthält, wird die Größe der folgenden Datensätze abgelegt. Sind alle Elemente kleiner als Lower_Limit,

wird eine Verbindung zu einem neuen Server aufgebaut und die nachfolgend empfangenen Sensordaten können wieder weitereskaliert werden.

Die reguläre oder auch irreguläre Beendigung eines Clients wird analog zu dem Sensormanager ignoriert und der Kommunikationssocket entfernt. Der entfernte Client kann eine neue Verbindung zu demselben Server oder einem anderen in dessen Ebene mit Hilfe der Funktion *getserver*() aufbauen.

11.4.1 Eskalierung in nächste Hierarchieebene

Die Eskalierung der Daten funktioniert analog zu dem Sensormanager. Das konsolidierte XML-Dokument wird komprimiert und schließlich über den eingerichteten Socket zu dem Server gesandt. Wiederum wird ein Protokoll zur dynamischen Reservierung von Speicherplatz verwendet.

11.4.2 Speicherung in lokaler Datenbank

Müssen die Daten lokal gespeichert werden, werden die konsolidierten Datensätze aus dem programminternen Format durch eine Funktion in ein datenbankverständliches Format übersetzt. Diese Daten werden über eine generalisierte Schnittstelle an das verwendete Datenbankmodul weitergerecht, welches die DBMS-spezifische Programmierschnittstelle benutzt, geeignete Datenbank-Kommandos erzeugt und an den DBMS-Server absetzt, um die Daten zu sichern. Die Variable DBMS gibt das benutzte DBMS an. DB_Name spezifiziert den Namen der Datenbank, DB_Table den Namen der Tabelle. Als Tabellenname muß mondat genommen werden, da das Programm sich sonst unverständlicherweise beendet. Dies ist besonders seltsam, da alle Variable mit der identischen Funktion aus XMLMonConfig eingelesen werden. Der Name eines Benutzers mit Lese- und Schreibrechten auf die Datenbank muß in DB_User angegeben sein. Das eventuell notwendige Passwort ist in dem Element DB_Passwd enthalten.

12 Zusammenfassung

Ein Monitoring System ist zwangsläufig auf vielen Rechnern verteilt. Dennoch läßt sich die erwünschte Funktionalität mit nur zwei unterschiedlichen Programmen erreichen.

Das erste Programm, der Sensormanager, dient der lokalen Verwaltung von Sensoren. Die Sensoren können dynamisch über eine einfach geartete Schnittstelle an den Sensormanager ankoppeln, wo sie mit Timestamps versehen werden.

Das zweite Programm, der Escalator, läuft in verschiedenen Instanzen auf ausgezeichneten Rechnern. Gemeinsam bilden diese Programme eine Serverhierarchie, die sich selbst rekonfigurieren kann und somit ein load-balancing ermöglicht. Auf diese Weise wird eine gute Skalierbarkeit des HCMS erreicht. Können Daten nicht mehr eskaliert werden, so werden sie in dezentralen Datenbanken gespeichert. Diese ermöglichen durch eine geeignete Schnittstelle einen gezielten Transfer gewünschter Datensätze an eine beliebigen, zentralen Ort.

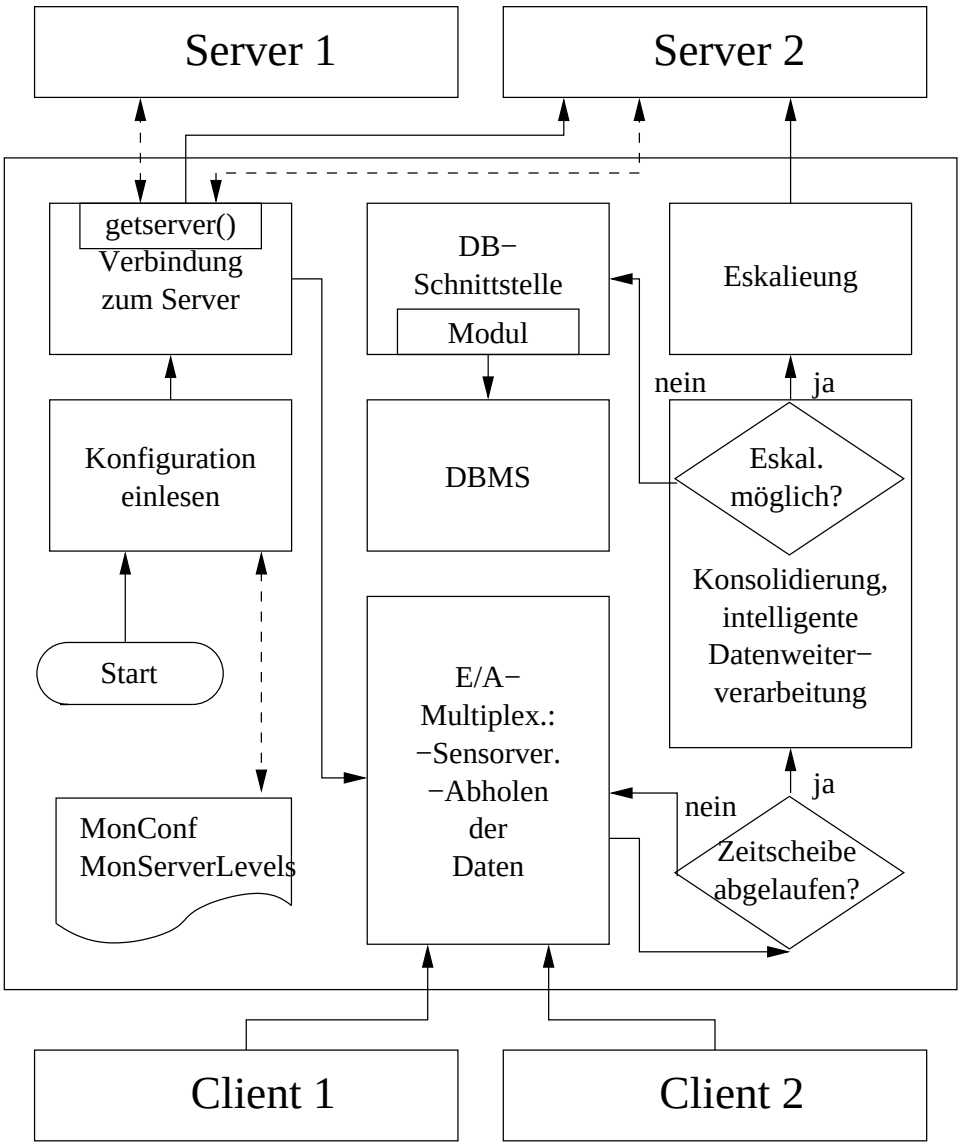

Abbildung 17: Aufbau des Consolidators/Escalators

Teil VI
Ausblick

Bei der ersten Implementierung eines Programmes schleichen sich zwangsläufig Programmfehler ein, die nicht sofort offen zu Tage treten. Auch Schwächen der Architektur selbst werden oft erst auf einen zweiten Blick offenbar. Kap. 13 behandelt fehlerhafte Stellen und architekturelle Probleme des HCMS.

Da nur beschränkt Zeit zur Implementierung des HCMS zur Verfügung stand, konnten viele Aspekte nicht oder nur in gewissem Umfang eingebaut werden. Manche dieser Punkte sind aber wichtig für einen konsistenten, automatisierten und fehlerfreien Betrieb des Cluster Monitoring Systems. Eine Auswahl dieser nicht umgesetzten Ideen wird in Kap. 14 besprochen.

Die Liste an Unzulänglichkeiten und fehlender Funktionalität wird keineswegs vollständig sein. Viele Probleme und notwendige Erweiterungen dürften sich aber erst nach längerem Testbetrieb zeigen. Ausführliche Tests waren aufgrund der zu knappen Zeit leider nicht mehr möglich.

13 Bekannte Schwächen des HCMS

- **Sicherstellung einer globalen Konfiguration:** Das HCMS benötigt für seinen Betrieb drei Konfigurationsdateien: XMLMonConfig, XMLMonHierarchy und XMLMonTags. Sind diese nicht vorhanden, finden Standard-Werte Verwendung. Dies ist aber für die Angabe der Serverhierarchie nicht möglich. Besonders wichtig ist, daß alle Rechner die gleichen Konfigurationsdateien besitzen, um eine Kommunikation zwischen ihnen zu ermöglichen und allen ein identisches Bild der Serverhierarchie zu vermitteln.

  Die drei Konfigurationsdateien müssen momentan von Hand an eine geeignete Stelle kopiert werden. Sinnvoll wäre an dieser Stelle ein Konfigurationsserver. Dieser besitzt stets die aktuelle Konfiguration des HCMS und ist für alle Clusterknoten erreichbar. Zu diesem Zweck könnte DHCP [DHC] verwendet werden. Auf einem Rechner läuft ein DHCP-Server der zentral Informationen für seine Clients bereithält. Ein DHCP-Client[22]ann über eigens definierte Optionen[23] den DHCP-Server nach Konfigurationsdateien fragen. Diese Anfrage kann sowohl

[22] k

[23] Eine **Option** ist eine selbst definierte Anfrage. Bei deren Auftreten wird die gewünschte Information zurückgeliefert, zudem kann ein Programm gestartet werden.

zeitlich getriggert als auch unmittelbar ausgelöst werden. Die Dateien werden auf den Client in ein angegebenes Verzeichnis kopiert.

- **Langsames Suchen eines Servers:** Die Funktion *getserver*(), welche für die Suche nach der IP eines geeigneten Servers zuständig ist (Kap. 9.7), braucht trotz der Verwendung von Threads Zeiten im Bereich von Sekunden, um ein Ergebnis zurückzuliefern. Ursache hierfür ist eine zwar funktionierende, aber für zeitkritische Anwendungen ungünstige Implementierung der Funktion. Diese Funktion wurde im Rahmen des HCMS als erstes geschrieben und sollte neu programmiert werden.

- **Race-Condition[24] bei dem Aufbau der Hierarchie:** Starten alle Escalator-Programme zur gleichen Zeit (beispielsweise bei einem Neustart des Clusters), kann es passieren, daß ein Programm keinen Server findet, obwohl sich in der nächsten Hierarchieebene ein Rechner mit einem weiteren Escalator befindet. Die Ursache ist wiederum die Funktion *getserver*(), mit der ein Server gesucht wird: solange sie in Aktion ist, kann kein Port eingerichtet werden, auf dem der Escalator selbst auf Verbindungsanfragen lauscht.

  Dieses Verhalten kann nur verhindert werden, indem ein Escalator, welcher sich auf einem Rechner weiter oben in der Hierarchie befindet, früher gestartet wird als ein Escalator unterhalb dieser Ebene.

- **MAIN_SERVER-Status irreversibel:** Ein Escalator versucht stets, eine Verbindung zu einem Server aufzubauen. Gelingt dies nicht, wird der MAIN_SERVER-Modus aktiv: alle Daten werden lokal in einer Datenbank abgespeichert. Auch wenn physikalisch eine Weitereskalierung der Daten wieder möglich wäre, bleibt der Modus erhalten; dieser Zustand ist also irreversibel.

- **Keine Sensoren auf oberster Hierarchieebene:** Ein Rechner in der obersten Ebene der Escalatorhierarchie muß die Daten stets in einer Datenbank abspeichern. Bei der Suche nach einem Server wird also keiner gefunden. Da der Sensormanager dieselbe Funktion wie der Escalator zum Auffinden des Servers verwendet, beendet sich das Programm wegen fehlender Eskalierungsmöglichkeiten. Auf einem Rechner dieser Ebene können also keine Sensoren des HCMS laufen.

[24] Eine **race-condition** tritt auf, wenn das Resultat eines Programms, welches verschiedene Teile parallel in Prozessen oder Threads ausführt, von der Reihenfolge der Beendigung dieser Teile abhängt.

- **Lokale Sensordaten nicht sofort lokal verfügbar:** Alle Sensoren eines Computers schicken ihre Daten an den lokalen Sensormanager, der sie weitereskaliert. Diese Daten werden an einer Stelle in der Hierarchie gespeichert und sind somit global verfügbar. Wünschenwert wäre aber eine lokale, instantane Verfügbarkeit der Daten.

- **Verzögerte globale Verfügbarkeit:** Ausgewertet werden können die Datensätze erst, wenn sie in einer Datenbank gespeichert sind. Bezeichnet man mit N die Tiefe der Serverhierarchie und stellt T Das Zeitintervall dar, innerhalb dessen ein Eskalierungsvorgang stattfindet, so kann es $N \cdot T$ dauern, bis die Daten verfügbar sind.

- **Datenbank-Performance:** Das Datenbank-Modul eines Escalators fügt jeden Datensatz einzeln in eine Tabelle ein. Die Datensätze könnten aber schneller abgelegt werden, wenn die Informationen zunächst in einer Datei gespeichert werden würden. Diese Datei könnte schließlich in die Datenbank eingelesen werden. Effektiv wären durch diese Maßnahme die Limits des Escalators höher gesetzt.

14 Fehlende Funktionalität

Eine zusätzliche Erweiterung des HCMS wäre wünschenswert, zum einen, um es einfacher bedienen zu können, zum anderen, um zusätzliche Funktionalität und Betriebssicherheit zu gewinnen.

Folgende Stichpunkte stellen einen Auszug an Verbesserungsmöglichkeiten für das HCMS dar:

- **Filterung von Sensoren:** Nur wenn der Name eines Sensors identisch mit einem in der Datei XMLMonTags spezifizierten Namen ist, wird der Sensor bei der Konsolidierung mit aufgenommen und seine Daten weitereskaliert. Diese Funktionalität ist bei der Umstellung von dem alten Sensormanager (verschiedene Zeitscheiben, festfrequente Sensoren) zu dem neuen verloren gegangen. Momentan wird jeder Name eines Sensors akzeptiert.

- **Abholen der verteilten Daten:** Das HCMS legt die Daten letzlich in verteilten Datenbanken ab. Eine Schnittstelle zu den DBMS ist vorhanden, um das Abholen der Daten muß sich eine Anwendung bisher aber selbst kümmern. Eine Möglichkeit, den Zugriff auf die Daten zu erleichtern wäre es, einen zentralen Server einzurichten, an den Anfragen abgesetzt werden können. Dieser befragt wiederum die dezentral lokalisierten Datenbanken.

- **Aufsetzen der Datenbank:** Damit das HCMS funktioniert, muß auf jedem Rechner der Serverhierarchie ein DBMS installiert werden. Auch die Tabelle, in welcher die Daten der Sensoren abgelegt werden, ist vor dem Anlauf des HCMS zu erledigen. An dieser Stelle bietet sich ein Skript an, das diese auf jedem Rechner identische Prozedur erledigt.

- **Datenbank-Module:** Aufgrund von Zeitmangel ist nur ein Datenbank-Modul für PostgreSQL vorhanden.

Teil VII
Zusammenfassung

Die vorliegende Diplomarbeit befaßt sich mit der Entwicklung eines Cluster Monitoring Systems. Dieses sammelt mit Hilfe von Sensoren gewünschte Informationen auf jedem Knoten. Die Daten werden aufbereitet und schließlich so gespeichert, daß sie auf dem ganzen Cluster verfügbar sind. Wichtigster Aspekt ist hierbei eine gute Skalierbarkeit des Systems. Da es möglich sein soll, beliebige Datensätze der gespeicherten Daten effizient auszulesen, wird ein DBMS zur Verwaltung der Datenbanken benutzt.

Die Skalierbarkeit eines Cluster Monitoring Systems hängt wesentlich von dem verwendeten DBMS ab. Zwei relationale Open Source DBMS wurden einer Betrachtung unterzogen: MySQL und PostgreSQL. Beide verwenden SQL als Datenbanksprache. Die durch die DBMS gegebenen Limitierungen sind sich stärker als erwartet. Systematische Tests der Leistungsfähigkeit bei dem Einfügen von Datensätzen in eine Datenbank erwiesen sich als notwendig.

Das Cluster Monitoring System besteht aus drei Stufen: den Sensoren, dem Sensormanager und einer Serverhierarchie auf deren Knoten der Escalator installiert ist.

Sensoren messen auf jedem Knoten des Clusters die interessierende Größe. Diese Daten werden einem lokalen Server, dem Sensormanager, als Zeichenkette übergeben. Auf diese Weise können beliebige Informationen übermittelt werden. Es wird eine einfache Schnittstelle verwendet, welche die Implementierung eines Sensors in vielen Programmiersprachen erlaubt. Eine in C++ geschriebene Bibliothek bietet hierzu geeignete Funktionen zur Kommunikation mit dem Sensormanager an.

Der Sensormanager sammelt die Daten der Sensoren und versieht sie mit einer Timestamp zur späteren Identifikation. Die Sensoren werden dynamisch verwaltet und können in beliebiger Anzahl unabhängig voneinander an den

Sensormanager ankoppeln. Die Datensätze werden konsolidiert und eine hierarchische XML-Struktur geschrieben. Diese wird komprimiert und an einen Rechner in einer Serverhierarchie weitergereicht.

Die Serverhierarchie besteht aus ausgezeichneten Rechnern des Clusters. Auf jeder Ebene der Hierarchie befinden sich durch ihre IP eindeutig bezeichnete Computer. Diese Strukturierung wird in einer Konfigurationsdatei angegeben. Die anfallenden Daten werden durch die Hierarchie von unten nach oben durch den Escalator durchgereicht und dabei auf jedem Knoten konsolidiert, um eine Datenbündelung ohne redundante Informationen zu erreichen. Hierzu ist eine Dekompression der Datensätze vor und eine erneute Kompression nach der Konsolidierung notwendig. Werden angegebene Limits überschritten oder können die Daten nicht weiter eskaliert werden, so werden die gesammelten Informationen auf dem entsprechenden Knoten lokal in einer Datenbank abgelegt. Dieses Konzept der dezentralen Datenspeicherung erlaubt eine gute Skalierbarkeit des Cluster Monitoring Systems. Durch Verwendung von DBMS zur Verwaltung der Datenbanken ist eine zentrale Verfügbarkeit dieser Daten möglich, zudem kann gezielt auf bestimmte Datensätze zugegriffen werden.

Das entwickelte Cluster Monitoring System kann sich selbst rekonfigurieren. Ist ein Rechner nicht mehr erreichbar, so wird testweise eine neue Verbindung zu einem Computer in derselben Hierarchieebene aufgebaut. Um eine gute Lastverteilung zu erreichen, liefert der kontaktierte Rechner bei diesem Test die Zahl der mit ihm bereits verbundenen Maschinen zurück. Es wird mit dem Computer mit der geringsten Last eine dauerhafte Verbindung erzeugt.

Es bleibt abzuwarten, ob das HCMS den gestellten Anforderungen gerecht wird. Dies wird sich erst nach einem längeren Testbetrieb zeigen.

Teil VIII
Die CD

Die beigefügte CD enthält die vorliegende Diplomarbeit als Latex-, DVI- und Postscript-Dateien. Sie ist im Verzeichnis /Diplom zu finden.

Im Verzeichnis /DBMS liegt der Datenbankbenchmark in verschiedenen Versionen, die jeweils zu einem Datensatztyp gehören. Der Benchmark ist nicht ohne weiteres lauffähig und wurde nur der Vollständigkeit halber mit hinzugefügt.

Im Verzeichnis /HCMS ist das entwickelte Cluster Monitoring System zu finden. Die Datei INSTALL.txt gibt einige Hinweise zur Installation. Die Dokumentation muß leider noch als unzureichend gelten und wird überarbeitet werden.

Literatur

[Bad01] Badach, A.: *Technik der IP-Netze*. Hanser, 2001.

[CLU] http://www.kip.uni-heidelberg.de/ti/cluster.

[DHC] http://www.dhcp.org.

[FT] http://hep-proj-grid-fabric.web.cern.ch/hep-proj-grid-fabric/Tasks/fault-tolerance. Data Grid Project, Work Package 4. Fault Tolerance.

[GAN] http://ganglia.sourceforge.net.

[Her99] Herold, H.: *Linux-Unix Systemprogrammierung*. Addison-Wesley, 1999.

[Ker90] Kernighan, B. W. und Ritchie, D. M.: *Programmieren in C*. Carl Hanser Verlag, 1990.

[Mei02] Meissner, N.: *Gut bemessen*. iX, Seiten 58–64, Januar 2002.

[MYS] http://www.mysql.com.

[NGO] http://fermitools.fnal.gov/abstract/ngop.

[ORA] http://oracle.com.

[PAM] http://hep-proj-grid-fabric.web.cern.ch/hep-proj-grid-fabric/Tasks/monitoring. Data Grid Project, Work Package 4. Monitoring.

[PER] http://www.perl.com.

[Pet99] Petkovic, D.: *SQL-Die Datenbanksprache*. McGraw-Hill, 1999.

[PGS] http://www.postgresql.org.

[Ray01] Ray, E. T.: *Einführung in XML*. O'Reilly, 2001.

[Str98] Stroustrup, B.: *Die C++-Programmiersprache*. Addison-Wesley, 1998.

[SUP] http://www.acl.lanl.gov/supermon.

[TIN] http://www.grinninglizard.com/tinyxml.

[VAC] http://www.siasa.com.ar.

[ZLI] http://www.gzip.org.

Teil IX
Danksagung

An erster Stelle möchte ich meine Dankesworte an Herrn Prof. Lindenstruth sowohl für das spannende Thema als auch für die Unterstützung in jeder Form richten.

Dank besonders auch an Lord Hess sowie Frank Pister. Nur durch ihre Hilfestellungen und Anregungen konnte die Diplomarbeit in dieser Form zustande kommen.

Danken will ich auch Arne Wiebalck sowie Timm Morten Steinbeck. Stets waren sie bereit, mir jede mögliche Form an Hilfe zu geben.

Danke an die Mitglieder des Lehrstuhls für die angenehme Atmosphäre und die.schöne Zeit. Dank auch auch an alle Mitarbeiter des Kirchhoff-Instituts für die stets freundliche Zusammenarbeit.

Nicht zuletzt Dank an den bisher noch unbekannten Zweitkorrektor dieser Arbeit.

Ich bedanke mich besonders bei meiner Freundin Patricia Montag für die moralische Unterstützung und das Verständnis in Streßphasen.

Besonderen Dank auch an meine Eltern, welche mir mein Studium ermöglicht haben und immer für mich da waren.

Erklärung:

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den _____ _____

Unterschrift