Entwicklung einer Umgebung zur Grid-basierten MonteCarlo-Produktion für das H1-Experiment

DIPLOMARBEIT
am Lehrstuhl für Experimentelle Physik V der
Universität Dortmund

vorgelegt von

Till Moritz Karbach

März 2006

Entwicklung einer Umgebung zur Grid-basierten MonteCarlo-Produktion für das H1-Experiment

Diplomarbeit
am Lehrstuhl für Experimentelle Physik V der
Universität Dortmund

vorgelegt von

Till Moritz Karbach

März 2006

Kurzfassung

Der Umbau des HERA-Speicherringes führte zu einer wesentlich gesteigerten Datenrate des H1-Experiments. Gleichermaßen stieg der Bedarf an MONTECARLO-Ereignissen und an Rechenkapazität, um diese zu produzieren. Eine vielversprechende Entwicklung in der Informationstechnologie ist das *Grid Computing*, das neue Rechnerinfrastruktur zur Verfügung stellt. In der vorliegenden Arbeit wurde eine Produktionsumgebung entwickelt, die das *LHC Computing Grid* (LCG) für die MONTECARLO-Produktion des H1-Experiments nutzbar macht. Nachdem Lösungen für viele Unzulänglichkeiten der aktuellen LCG-Middleware gefunden wurden, konnte die Produktionsumgebung in die offizielle MONTECARLO-Produktionskette integriert werden. Dabei wird bisher eine Leistung erreicht, die in derselben Größenordnung liegt, wie diejenige etablierter klassischer Rechnerfarmen.

Abstract

The upgrade of the HERA storage ring led to a significantly higher data rate of the H1 experiment. Therefore the need for MONTECARLO events increased, causing a raise of the computing ressources which are necessary to produce these events. A promising development concerning computing technology is known as *grid computing*, which provides access to new computing ressources. During this diploma thesis a framework was developed, which enables the H1 MONTECARLO production to make use of the *LHC Computing Grid* (LCG). Several solutions to overcome the insufficiencies of the LCG-Middleware have been implemented. Finally the framework has been integrated into the official MONTECARLO chain, where it reaches about the same performance as well established classical computing farms.

`afs/desy.de/user/m/mkarbach/public/thesis.pdf`

Inhaltsverzeichnis

Einleitung

Angetrieben vom Wunsch, die kleinsten Bausteine der Welt zu verstehen, bauten die Physiker im Laufe der Zeit immer leistungsfähigere Teilchenbeschleuniger. Diese Maschinen lassen im Zentrum von großen Detektoren Teilchen kollidieren, die auf hohe Energien beschleunigt wurden. Je genauer man dabei hinsah, desto mehr neue Teilchen wurden entdeckt. Heute gelten die Quarks und Leptonen als fundamentale Bausteine der Materie, die über den Austausch von so genannten Eichbosonen miteinander wechselwirken. Das Auflösungsvermögen der Teilchendetektoren ist über die Heisenbergsche Unschärferelation [1] mit dem Impulsübertrag verknüpft, der bei der Kollision der beschleunigten Teilchen auftritt.

$$\Delta x \cdot \Delta p \geqslant \hbar \tag{1}$$

Dabei bezeichnet Δx die Ortsunschärfe, und Δp lässt sich mit dem Impulsübertrag identifizieren. Der maximal mögliche Impulsübertrag steigt mit der Energie der Beschleuniger, und die Energie wiederum mit der Größe der Maschinen. Kurz gesagt gilt, dass je größer die Beschleuniger und Detektoren sind, desto größer ist ihre Auflösung. Heutzutage haben Teilchendetektoren die Dimension von mehrstöckigen Häusern erreicht, und sie können nur noch verstanden werden, wenn umfangreiche und komplexe Simulationsrechnungen durchgeführt werden – so genannte MONTECARLO-Produktionen.

Diese Großanlagen werden gebraucht, um das etablierte und sehr erfolgreiche *Standardmodell der Elementarteilchen* zu prüfen, und nach Hinweisen zu suchen, die eine Erweiterung des Modells erfordern würden. Das Standardmodell führt die Vielfalt der Elementarteilchen auf wenige grundlegende Eichsymmetrien zurück, aus denen die fundamentalen Wechselwirkungen folgen. Insbesondere sagt das Standardmodell voraus, wieviele Reaktionen einer bestimmten Art auftreten sollten, wenn hochenergetische Teilchen kollidieren. Um die Theorie zu testen, müssen also die auftretenden Ereignisse quantifiziert werden. Das ist unter anderem die Aufgabe des H1-Detektors am Hamburger Forschungszentrum DESY. In seinem Zentrum kreuzen sich die Teilchenstrahlen des Beschleunigers HERA. Abbildung 1 zeigt die vom H1-Experiment gesammelte integrierte Luminosität, die

proportional zur Zahl der gemessenen Ereignisse ist (der Proportionalitätsfaktor σ ist der Wirkungsquerschnitt[1]).

$$N = \sigma \cdot L \tag{2}$$

HERA wurde in den Jahren 2000 bis 2003 deutlich verbessert, weshalb in der Abbildung zwischen beiden Phasen unterschieden wird: HERA-I (1993 bis 2000) und HERA-II(2003 bis 2005). Allein im vergangenen Jahr 2005 wurden so viele Daten aufgezeichnet, wie in allen Jahren vor dem Ausbau zusammen. Es ist anzunehmen, dass sich diese günstige Entwicklung fortsetzt.

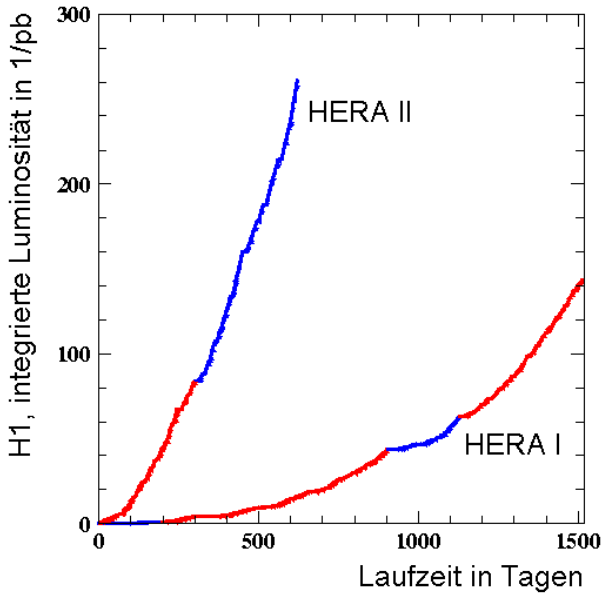

Abbildung 1: Die integrierte Luminosität des H1-Experiments. Die Ausbauphasen HERA-I (1993-2000) und HERA-II (2003-2005) sind unterschieden. [Quelle: Nelly Gogitidze]

Neben der Weiterentwicklung der Experimente gab es wesentliche Fortschritte in der Datenverarbeitung. Anfang der 90er Jahre revolutionierte das von dem Teilchenphysiker Tim Berners-Lee erfundene *World Wide Web* (WWW) die Informationstechnologie und machte den Computer gesellschaftsfähig. Berners-Lee arbeitete am Forschungszentrum CERN in Genf, und das WWW war eigentlich

[1]Der Wirkungsquerschnitt eines Prozesses kann als Wahrscheinlichkeit interpretiert werden, dass bei einer Teilchenkollision eben dieser Prozess stattfindet.

als Kommunikationsplattform für die LEP-Experimente gedacht. Auch heute sind Teilchenphysiker an der Entwicklung eines neuen Konzepts beteiligt. Auf ähnliche Weise, wie das WWW Informationen dezentral und allgegenwärtig zur Verfügung stellt, soll das *Grid Computing* Rechenleistung zugänglich machen [2].

Zielsetzung der Arbeit

Immer größere Experimente erzeugen immer größere Datenmengen. Diesen Daten muss eine ähnlich wachsende Menge an MONTECARLO-Simulationsrechnungen gegenübergestellt werden, weshalb auch der Bedarf an Rechenleistung steigt. Das Grid Computing hat das Potential, einen wertvollen Beitrag zu liefern, um den gestiegenen Bedarf zu decken. Zudem ist ein immer größer werdender Teil der bisherigen Rechnerkapazitäten nur noch über Grid Computing nutzbar. Deshalb erscheint der Beitrag des Grids umso wertvoller.

Es ist das Ziel dieser Arbeit, eine Umgebung zu entwickeln, mit der die MONTECARLO-Produktion des H1-Experiments am DESY erstmals durch Grid Computing unterstützt werden kann.

Kapitel 1

Einführung

1.1 Der HERA-Speicherring am DESY

Das Deutsche Elektronen-Synchrotron (DESY) in Hamburg ist eines der größten Forschungszentren für Teilchenphysik der Welt. Die beteiligten Wissenschaftler betreiben hier einen Teilchenbeschleuniger mit $6,4$ km Umfang, die *Hadron-Elektron-Ring-Anlage* (HERA). HERA beschleunigt Elektronen[1] und Protonen auf sehr hohe kinetische Energien, $27,5$ GeV und 920 GeV, und bringt sie zur Kollision. Die dabei aus der freiwerdenden Energie entstehende Vielzahl neuer Teilchen wird von zwei großen unterirdischen Detektoren, H1 und ZEUS, nachgewiesen. Beide Experimente erforschen auf diese Weise hauptsächlich die Protonstruktur, und damit die Zusammenhänge zwischen den Konstituenten des Protons, den Quarks und Gluonen.

Zwei weitere Detektoren, HERMES und HERA-B, nutzen nur einen der beiden Teilchenstrahlen. Sie lassen ihn jeweils mit einem feststehenden Ziel kollidieren. HERMES verwendet dafür den Elektronenstrahl und ein Gastarget, um die Spinstruktur[2] von Protonen und Neutronen zu untersuchen. Das HERA-B-Experiment, das bereits abgeschlossen ist, verwendete den Protonenstrahl und ein Drahttarget, um die Eigenschaften schwerer Mesonen[3] zu untersuchen, die bei der Kollision entstanden. Alle vier Experimentierhallen und die Beschleunigeranlagen sind in Abbildung 1.1 gezeigt.

Die Protonen und Elektronen werden nicht einzeln beschleunigt, sondern befinden sich als Pakete von mehreren Milliarden Teilchen im HERA-Ring. Diese Pa-

[1] HERA kann auch mit Positronen betrieben werden. Im Folgenden ist immer auch das Positron gemeint, wenn von Elektronen die Rede ist.

[2] Der Spin ist eine quantenmechanische Größe, die sich mit dem Eigendrehimpuls eines Teilchens in Verbindung bringen lässt.

[3] Mesonen sind Teilchen aus einem Quark und einem Antiquark.

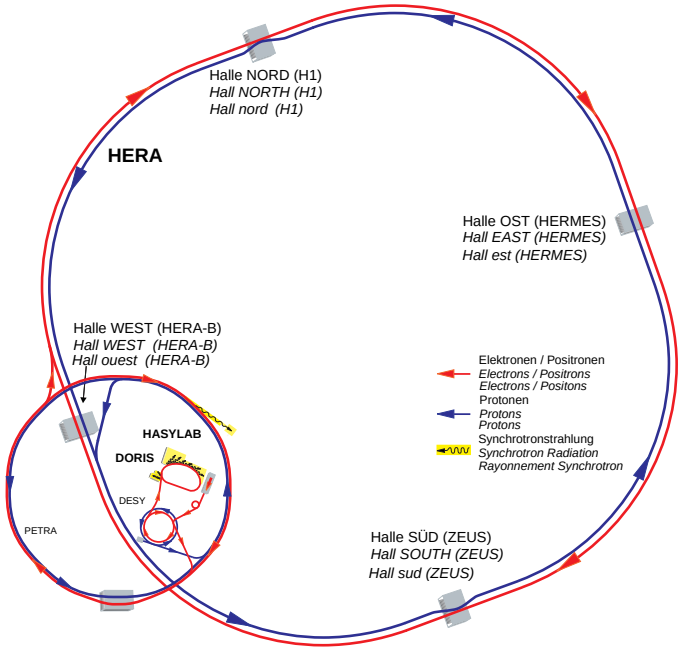

Abbildung 1.1: Der HERA-Ring und die Vorbeschleuniger am DESY. Weiterhin sind die Experimentierhallen und die dortigen Experimente eingezeichnet.

kete begegnen sich alle $96\,\text{ns}$ im Wechselwirkungspunkt des H1-Detektors, was einer Wechselwirkungsrate von $10,4\,\text{MHz}$ entspricht. Allerdings kommt es nur in etwa jedem zehntausendsten dieser Begegnungen vor, dass ein Elektron und ein Proton auch tatsächlich kollidieren. Wenn infolgedessen im Detektor Teilchen nachgewiesen werden, nennt man dies ein *Ereignis*. Die Art des Ereignisses bestimmt dabei die Impulse, Ladungen und Energien der entstandenen Teilchen.

Während der ersten Ausbaustufe des HERA-Rings (HERA-I, 1993-2000) wurde eine große Datenmenge gesammelt, weshalb Analysen von Prozessen mit großem Wirkungsquerschnitt mit hoher Präzision durchgeführt werden konnten. Die nächste Herausforderung war somit, auch Prozesse mit kleinem Wirkungsquerschnitt zu untersuchen. Von diesen interessanten Ereignissen waren noch zu wenige nachgewiesen worden, weshalb die Analysen in diesem Bereich statistisch limitiert waren. Daher wurde eine Umbaumaßnahme beschlossen, die zum Ziel hatte, die Ereignisrate deutlich zu erhöhen, um in gleicher Zeit wesentlich mehr Ereignisse sammeln zu können. Die Ereignisrate ist proportional zum Wirkungsquerschnitt, der Proportionalitätsfaktor heißt *instantane Luminosität*. Diese konnte in den Jahren 2000 und 2001 um etwa einen Faktor 4 gesteigert werden. Die so erreichte Ausbaustufe wird HERA-II genannt.

1.2 Das H1-Experiment

Im Folgenden wird der Vielzweck-Detektor H1 [3, 4] vorgestellt. Zunächst einmal beeindrucken die äußeren Maße des Großexperiments: Der zylindrische Detektor hat eine Gesamtmasse von $2800\,\mathrm{t}$, einen Durchmesser von $11\,\mathrm{m}$ und eine Länge von $15\,\mathrm{m}$. Ein Querschnitt durch den H1-Detektor ist in Abbildung 1.2 gezeigt. Mit ihm können Teilchenspuren mit einer Genauigkeit von bis zu $10\,\mu\mathrm{m}$ vermessen werden. Zur Bestimmung der Teilchenenergie benutzt H1 weit über 40.000 Kanäle. Ihre Daten und die Spurdaten erzeugen potenziell Datenraten von mehreren Gigabyte pro Sekunde. Eine Besonderheit der HERA-Experimente ist, dass sie aufgrund der asymmetrischen Strahlenergien in Vorwärtsrichtung wesentlich aufwendiger instrumentiert sind. Dabei ist die Vorwärtsrichtung als diejenige des einlaufenden Protons definiert.

Die Aufgabe des H1-Detektors ist es, die Impulse, Energien und die elektrische Ladung der Teilchen eines Ereignisses zu messen. Dazu ist er schalenförmig um den Wechselwirkungspunkt aufgebaut. „Dieses Prinzip ist dadurch vorgegeben, dass in der inneren Schale zunächst die Spuren geladener Teilchen sehr genau und mit wenig störender Materie in einem Magnetfeld vermessen werden. Daran schließen sich nach außen das elektromagnetische und das hadronische Kalorimeter an, beides Detektoren, die in möglichst dichter Materie die Energiedeposition der Teilchen messen. Noch weiter außen werden dann die durchdringenden Myonen vermessen.“[4]

Die innere Schale besteht bei H1 aus mehreren Komponenten. In unmittelbarer Nähe zum Wechselwirkungspunkt 1 befindet sich der *zentrale Siliziumdetektor* 2, der den Wechselwirkungspunkt von Elektron und Proton exakt bestimmt. In Vorwärts- und Rückwärtsrichtung werden die Spuren vom *Vorwärts-Siliziumdetektor* 4 beziehungsweise *Rückwärts-Silizumdetektor* 3 vermessen. Nach außen schließen sich die größten zentralen Spurdetektoren an: zwei Drahtkammern, die konzentrisch um das Strahlrohr angeordnet sind. Sie sind in Abbildung 1.2 als 7 und 8 gekennzeichnet. Die Drahtkammern sind mit einem Gasgemisch gefüllt, das durch passierende Teilchen ionisiert wird. Die so entstehenden Elektronen driften in einem starken elektrischen Feld zu Signaldrähten, in denen sie ein messbares Signal erzeugen. Da ein Teilchen in der Regel mehrere Signaldrähte aktiviert, kann aus deren Position die Spur rekonstruiert werden. H1 verfügt zusätzlich über einen Spurdetektor in Vorwärtsrichtung 9.

An die innere Schale schließt sich nach außen hin das wichtigste Kalorimeter (12, 13) an. Es besteht aus Absorberschichten aus Blei und Stahl, in denen die nachzuweisenden Teilchen elektromagnetische und hadronische Schauer erzeugen. Diese werden mittels eines so genannten *aktiven Mediums* nachgewiesen,

[4] Günter Flügge und Peter Jenni, Physik Journal, Februar 2006, S. 32

das im Falle von H1 flüssiges Argon ist. Dieses trägt mit $400\,\mathrm{t}$ einen beachtlichen Teil zur Gesamtmasse von H1 bei. Das Flüssig-Argon-Kalorimeter wird im rückwärtigen Bereich vom *Spaghetti-Kalorimeter* (14 , 15) ergänzt, das seinen Namen wegen seiner unzähligen szintillierenden Fasern erhielt, die es als aktives Medium benutzt [5].

Sowohl das Kalorimeter als auch die Spurdetektoren sind von einer supraleitenden Spule 18 umgeben, die das Magnetfeld erzeugt, welches für die Impuls- und Ladungsbestimmung notwendig ist. Es ist parallel zur Strahlachse ausgerichtet und hat eine Stärke von $1,15\,\mathrm{T}$. Um dieses Magnetfeld zu schließen, ist ein instrumentiertes Eisenjoch 19 installiert. Man bezeichnet es als „instrumentiert“, denn es ist mit speziellen Spurdetektoren ausgerüstet, mit denen die durchdringenden Myonen nachgewiesen werden können. So bildet das Eisenjoch gleichzeitig das zentrale Myonsystem, das noch vom Vorwärts-Myonsystem 20 unterstützt wird. Letzteres besitzt einen eigenen Magneten (einen Toroiden), um auch den Impuls der vorwärtigen Myonen präzise messen zu können.

Die hohe Wechselwirkungsrate von $10,4\,\mathrm{MHz}$ macht den Einsatz eines so genannten *Triggersystems* nötig. Denn weder kann man den Detektor so schnell auslesen, noch ist es möglich, mit dieser Rate Ereignisse dauerhaft zu speichern[5]. Die nötige Ratenreduktion wird durch mehrere separate Triggerstufen erreicht. Sie entscheiden anhand von sehr schnell verfügbaren Informationen, ob ein Ereignis interessant genug ist, um aufgezeichnet zu werden. Beispielsweise berechnet der *schnelle Spurtrigger* [6] erste Parameter der Spuren in den Drahtkammern. So kann er Ereignisse mit interessanter Topologie auswählen.

Der H1-Detektor ist ein Instrument, bei dem zwischen seiner eigenen Größe und der Präzision, mit der es misst, viele Größenordnungen liegen. Es ist praktisch unmöglich, eine hunderte von Tonnen schwere Detektorkomponente auf einen Mikrometer genau auszurichten. Der Ausweg ist, die exakten Positionen der Komponenten zu bestimmen, und die Ereignisse mit diesen Daten zu korrigieren. Dafür werden für jede Datennahmeperiode aus bekannten Ereignissen Korrekturwerte berechnet, und in der *zentralen* H1-*Datenbank* gespeichert. Ebenso werden die Spurkammern und Kalorimeter kalibriert, so dass Ereignisse auf ein verändertes Ansprechverhalten dieser Detektorkomponenten korrigiert werden können. Auch diese Kalibrierungsdaten werden in der zentralen H1-Datenbank abgelegt.

Moderne Teilchendetektoren sind aufgrund ihrer Größe und der Vielzahl ihrer Systeme ungemein komplex. Deshalb besteht der einzige gangbare Weg zur Interpretation der von ihnen gelieferten Daten darin, zusätzlich detaillierte Simulationsrechnungen durchzuführen. Diese Simulationsrechnungen werden im folgenden Kapitel 1.3 vorgestellt.

[5] Die bei H1 eingesetzten Datennahmesysteme erreichen eine Rate von bis zu 50 Ereignissen pro Sekunde.

	Detektorkomponente	**Abkürzung**
1	Wechselwirkungspunkt	**I**nteraction **P**oint (IP)
2	zentraler Siliziumdetektor	**C**entral **S**ilicon **T**racker (CST)
3	rückwärtiger Siliziumdetektor	**B**ackward **S**ilicon **T**racker (BST)
4	vorwärtiger Siliziumdetektor	**F**orward **S**ilicon **T**racker (FST)
5	innere Proportionalkammer	**C**entral **I**nner **P**roportional Chamber **2000** (CIP2000)
6	äußere Proportionalkammer	**C**entral **O**uter **P**roportional Chamber (COP)
7	zentrale Jetkammer 1	**C**entral **J**et **C**hamber **1** (CJC1)
8	zentrale Jetkammer 2	**C**entral **J**et **C**hamber **2** (CJC2)
9	vorwärtige Spurkammer	**F**orward **T**racking **D**etector (FTD)
10	rückwärtige Proportionalkammer	**B**ackward **P**roportional **C**hamber (BPC)
11	Flüssig-Argon Kryostat	**L**iquid **Ar**gon Kryostat (LAr-Kryostat)
12	Flüssig-Argon Kalorimeter (elektromagnetisch)	**L**iquid **Ar**gon Calorimeter, elm. (LAr elm.)
13	Flüssig-Argon Kalorimeter (hadronisch)	**L**iquid **Ar**gon Calorimeter, hadr. (LAr hadr.)
14	Spaghetti-Kalorimeter (elektromagnetisch)	**Spa**ghetti **Cal**orimeter, elm. (SpaCal elm.)
15	Spaghetti-Kalorimeter (hadronisch)	**Spa**ghetti **Cal**orimeter, hadr. (SpaCal hadr.)
16	supraleitender Magnet (Rückwärtsrtg.)	(GG)
17	supraleitender Magnet (Vorwärtsrtg.)	(GO)
18	supraleitende Spule	–
19	zentrales Myonsystem	**C**entral **M**yon **S**ystem (CMS)
20	Myon-Toroid Magnet	–
21	vorwärtiges Myonsystem	**F**orward **M**yon **S**ystem (FMS)

Tabelle 1.1: Aufschlüsselung der Detektorkomponenten nach den in Abbildung 1.2 gezeigten Ziffern.

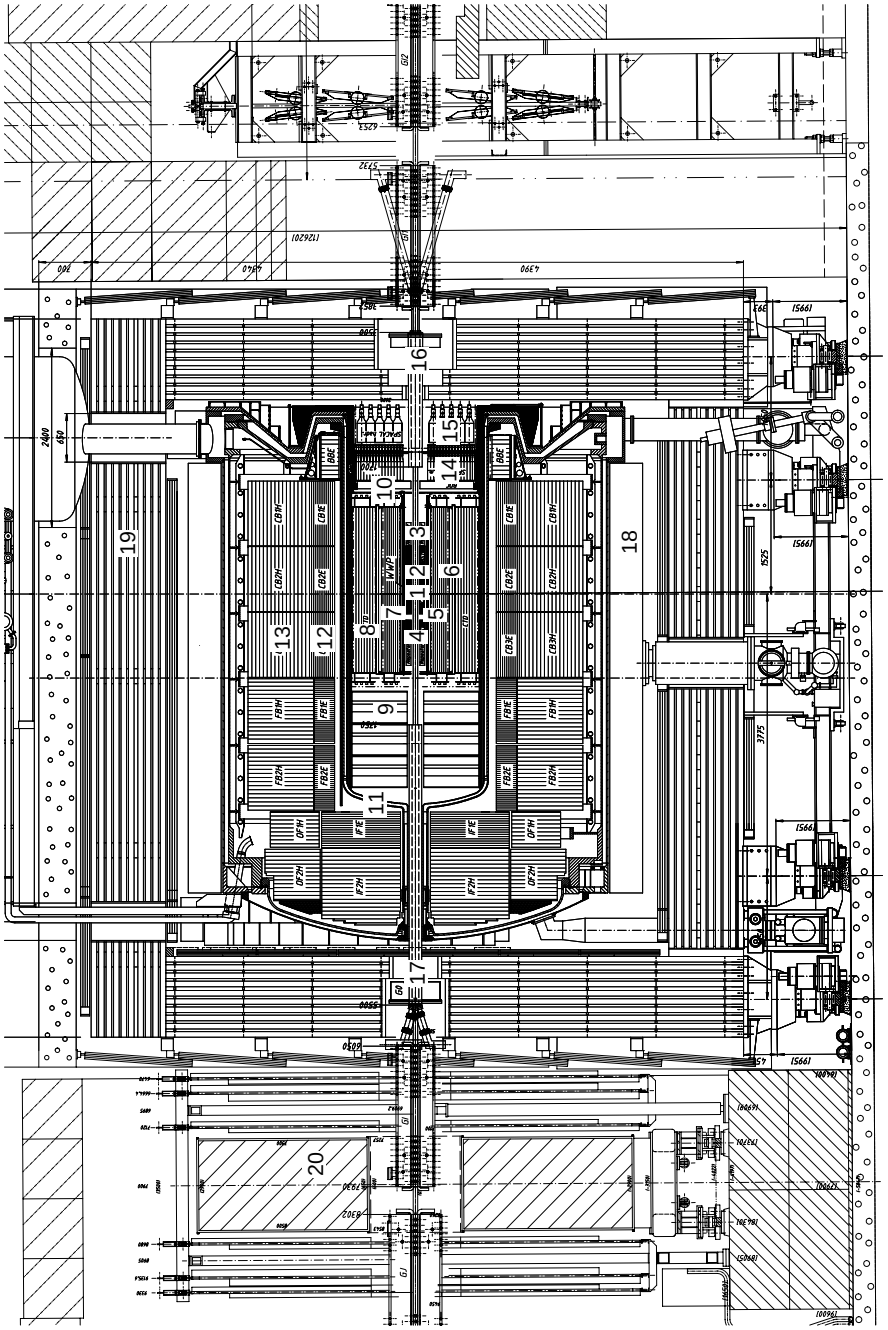

Abbildung 1.2: Querschnitt durch den H1-Detektor. Die Ziffern sind im Text und in Tabelle 1.2 erläutert.

1.3 MONTECARLO-Simulationen

Dieses Kapitel führt in die MONTECARLO-Simulationen ein, die in der experimentellen Teilchenphysik unverzichtbar sind. Die Bezeichnung „MONTECARLO“ leitet sich vom gleichnamigen Stadtteil von Monaco ab, der für seine Spielbanken berühmt ist, und in denen spielt bekanntermaßen der Zufall die Hauptrolle. Auch MONTECARLO-Verfahren bedienen sich des Zufalls, um analytisch schwer oder gar nicht lösbare mathematische Probleme in den Griff zu bekommen. Die Verbindung des Zufalls mit Mathematik überrascht vielleicht zunächst, aber in der Tat lassen sich mit MONTECARLO-Methoden die Lösungen solcher Probleme im Prinzip beliebig genau bestimmen – wenn die zur Verfügung stehende Rechenleistung für die gewünschte Präzision ausreicht. Die experimentelle Teilchenphysik untersucht durchweg statistische Prozesse, schließlich lässt das Standardmodell der Elementarteilchen als quantenmechanische Theorie nur Aussagen über die Wahrscheinlichkeit eines Prozesses zu. Auch deshalb bieten sich MONTECARLO-Simulationen zur Interpretation der realen Daten an.

Teilchenphysiker simulieren in der Praxis zweierlei. Erstens wird „ausgewürfelt“, welche Teilchen der Endzustand einer Kollision zwischen Elektron und Proton enthalten soll, wofür ein *Ereignisgenerator* verwendet wird. Zweitens wird simuliert, wie diese Teilchen mit dem Detektor wechselwirken. Als Ergebnis dieser *Detektorsimulation* ist zum Beispiel für ein definiertes Ereignis bekannt, an welchem Draht der Drahtkammer welcher Spannungsverlauf gemessen worden wäre. Die simulierten Daten durchlaufen anschließend dasselbe Rekonstruktionsprogramm, das auch für echte Daten verwendet wird. Danach lässt sich das Ereignis visualisieren, wie Abbildung 1.3 zeigt.

1.3.1 MONTECARLO-Methoden

Zwei MONTECARLO-Methoden werden im Folgenden genauer diskutiert: Das oben schon erwähnte „Auswürfeln“, bei dem zufällig entschieden wird, ob ein bestimmtes Ereignis eintritt, und das Auswerten von Integralen mittels zufällig verteilter Stützstellen. Für das Auswürfeln sei hier ein kleines Beispiel gegeben, in dem der radioaktive Zerfall von Atomkernen simuliert wird. Er soll durch die Annahme modelliert werden, dass jeder Kern in einem Zeitintervall Δt eine gleichbleibende Wahrscheinlichkeit P hat, zu zerfallen, wobei λ die Zerfallsrate bezeichnet

$$P = \lambda \cdot \Delta\mathrm{t}. \tag{1.1}$$

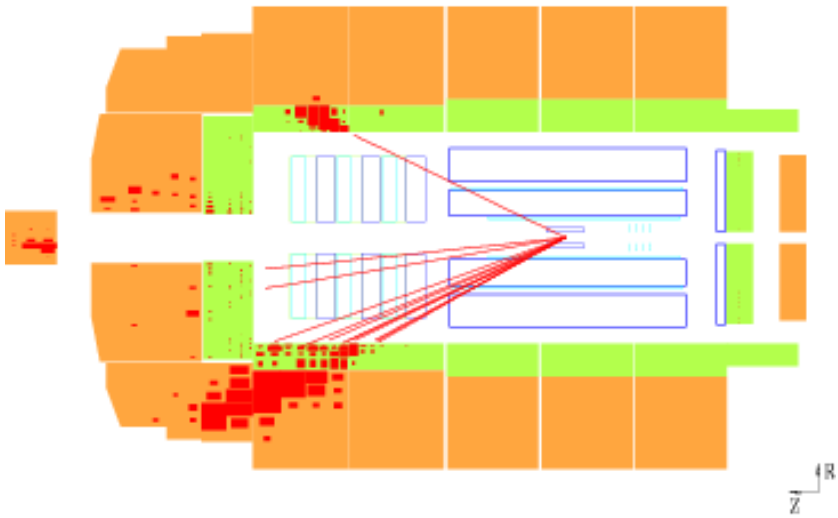

Abbildung 1.3: Ein Ereignis im H1-Detektor. Linien bezeichnen Teilchenspuren in den Spurdetektoren, Rechtecke Energiedepositionen in den Kalorimetern. Oben ist das rückgestreute Elektron zu sehen, unten die Überreste des Protons.

Dies führt über die differentielle Gleichung für die Zahl der nichtzerfallenden Kerne N, $dN = -N\lambda\, dt$, zum bekannten Zerfallsgesetz (N_0 bezeichnet die Anfangszahl unzerfallener Kerne)

$$N = N_0\, e^{-\lambda t}. \tag{1.2}$$

Der radioaktive Zerfall lässt sich simulieren, indem wiederholt eine zufällig ausgewählte Zahl aus dem Intervall $[0, 1]$ mit der Zerfallswahrscheinlichkeit P verglichen wird, wie folgender Pseudocode zeigt:

```
LOOP von t = 0  bis t in Schritten von Δt
  LOOP über die unzerfallenen Kerne
    # entscheide, ob ein Kern zerfällt
    IF ( random < λ · Δt  ) THEN
      verringere die Zahl unzerfallener Kerne um 1
    ENDIF
  ENDLOOP über die Kerne
  notiere Zahl unzerfallenen der Kerne
ENDLOOP über die Zeit
```

Die Abbildungen 1.4 und 1.5 zeigen die Ergebnisse einer realen Implementierung dieses Pseudocodes für $N_0 = 100$ beziehungsweise $N_0 = 5000$ zusammen

mit dem analytisch gewonnenen Zerfallsgesetz (Gleichung 1.2). Es wird deutlich, dass das Ergebnis der Simulation umso präziser wird, je mehr Simulationsschritte durchgeführt werden – also je mehr Kerne berücksichtigt werden.

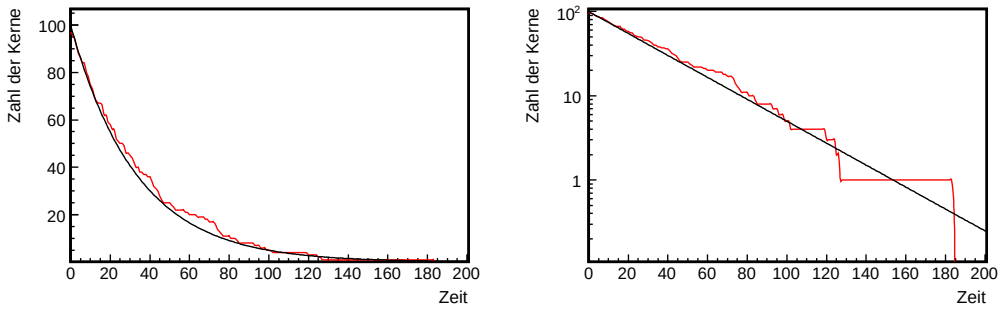

Abbildung 1.4: Simulierter Zerfall von 100 Nukleonen, Zerfallsrate $\lambda = 0,03$ (linear und logarithmisch)

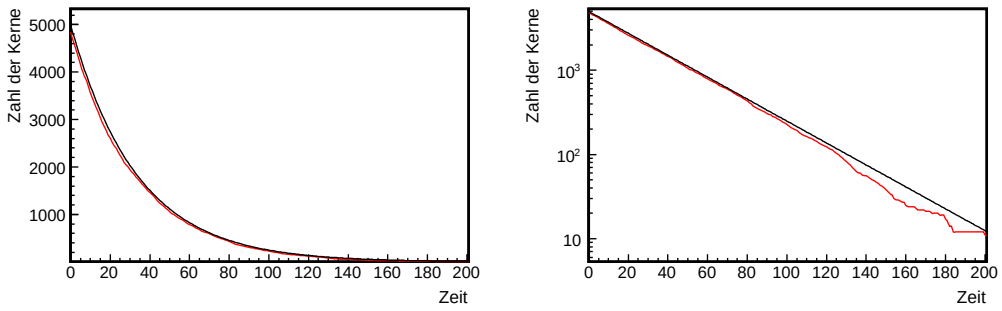

Abbildung 1.5: Simulierter Zerfall von 5000 Nukleonen, Zerfallsrate $\lambda = 0,03$ (linear und logarithmisch)

Die Grundlage für die zweite wichtige MONTECARLO-Methode, die Integralauswertung, sind zufällig verteilte Stützstellen, mit deren Hilfe das Integral angenähert werden kann. Wählt man zum Beispiel Werte x_i, die gleichförmig in einem Intervall $[a, b]$ verteilt sein sollen, so gilt folgende Gleichung [7]:

$$I = \int_a^b g(x)\,\mathrm{d}x \approx I_{MC} = \frac{b-a}{n}\sum_{i=1}^{n} g(x_i) \tag{1.3}$$

Der Fehler dieser Schätzung ist proportional zu $n^{-\frac{1}{2}}$, was insbesondere auch dann noch gilt, wenn man mehrdimensionale Integrale berechnen will. In diesem Fall ist die MONTECARLO-Methode anderen numerischen Verfahren überlegen. Zum Beispiel ist der Fehler bei der Trapezregel proportional zu $n^{-\frac{2}{d}}$ [8], wobei d die Zahl der Dimensionen bezeichnet.

Es sei noch angemerkt, dass es ein gewisses Problem darstellt, geeignete Zufallszahlengeneratoren zu implementieren. Es gibt Ansätze, die zum Beispiel das Rauschen eines Transistors auswerten. Allerdings können solche Apparate nicht den immensen Bedarf an Zufallszahlen decken. Man behilft sich mit so genannten Pseudozufallszahlen, die von einem Computeralgorithmus erzeugt werden. Da aber ein Computer eine deterministische Maschine ist, kann er per Definition keinen echten Zufall produzieren. Das führt dazu, dass ein solcher Zufallszahlengenerator eine Periodenlänge besitzt, nach der sich die Zahlen wiederholen. Bei guten Generatoren liegt diese in der Größenordnung von 10^{43} [9]. Außerdem hängt die erzeugte Zahlenreihe von einem Startwert ab, für den oft die Systemzeit des Computers gewählt wird.

1.3.2 Ereignisgeneratoren

Ereignisgeneratoren haben die Aufgabe, zu simulieren, welche Reaktionsprodukte bei einer Kollision von Primärteilchen (wie zum Beispiel Elektron und Proton) entstehen. Dabei berücksichtigen sie die bekannte Physik nach bestem Wissensstand[6]. Zunächst werden die Wahrscheinlichkeiten für bestimmte Prozesse berechnet. Das Standardmodell der Teilchenphysik sagt diese Wahrscheinlichkeiten voraus – allerdings oft nur in Form von differentiellen Verteilungen, die vom Ereignisgenerator mit MONTECARLO-Methoden integriert werden. Danach werden die Teilchen des Endzustandes „ausgewürfelt“, gewissermaßen analog zum Beispiel in vorigem Kapitel 1.3.1. Insbesondere werden dabei ihre Vierimpulse[7] festgelegt. Heutzutage reicht die Rechenleistung eines Arbeitsplatzrechners aus, um Ereignisse zu generieren.

1.3.3 Detektorsimulation und Rekonstruktion

Weitaus rechenaufwändiger als die Ereignisgeneratoren ist die Detektorsimulation, die die vom Generator erzeugten Teilchen durch den Detektor verfolgt. Dabei sind vielfältige Wechselwirkungen von Teilchen mit Materie zu berücksichtigen, zum Beispiel die Bremsstrahlung, die Paarbildung und der Comptoneffekt bei der

[6] Es gibt ebenso Generatoren, die so genannte *neue Physik* jenseits des Standardmodells implementieren.

[7] relativistische Notationsform für Energie und Impuls

Entstehung elektromagnetischer Schauer [10]. Anschließend kann die simulierte Detektorreaktion an dasselbe Rekonstruktionsprogramm weitergegeben werden, das auch echte Ereignisse rekonstruiert. Bei H1 sind beide Teile in einem einzigen Programmpaket zusammengefasst, das H1SIMREC genannt wird [3, 29]. Der Simulationsteil dieses Pakets basiert auf dem GEANT-Projekt [11], das auch bei der Detektorsimulation vieler anderer Experimente verwendet wird. Um H1SIMREC in großem Maßstab auszuführen, braucht man die Leistung von großen Rechnerfarmen. Denn pro simuliertem Ereignis wird eine Laufzeit von 1 bis 15 Sekunden benötigt, abhängig davon, welcher physikalische Prozess vorliegt. So brauchen Ereignisse, die viele Teilchen enthalten, wesentlich länger, als solche mit wenigen Teilchen. Im Folgenden werden die Daten vorgestellt, die von H1SIMREC benötigt werden.

- Die *zentrale* H1-*Datenbank* enthält dreierlei. Zunächst die Detektorkalibrierungskonstanten, die das Ansprechverhalten der Detektorkomponenten beschreiben. Dann die Ausrichtungskonstanten, die die Abweichungen der Detektorkomponenten von den bekannten räumlichen Positionen beschreiben. Und schließlich die Detektorgeometrie selbst. Das Gesamtvolumen der zentralen H1-Datenbank beträgt etwa zwei Gigabyte.

- Die *Noise Files* enthalten Informationen über das elektrische Rauschen der Kalorimeter und bestimmter Spurdetektoren (Proportionalkammern, siehe Tabelle 1.2). Von Zeit zu Zeit werden neue Noise Files aufgezeichnet, um veränderte Bedingungen zu berücksichtigen. Das Volumen der Noise Files für eine Zeit, in der die Bedingungen relativ konstant sind, beträgt ungefähr zwei Gigabyte.

- Die MONTECARLO-*Eingangsdatei* enthält die generierten Ereignisse, wie sie vom Ereignisgenerator erzeugt wurden. Eine Eingangsdatei, die 10.000 Ereignisse enthält, ist circa 15 Megabyte groß.

- In der Konfigurationsdatei für H1SIMREC werden Parameter konfiguriert, die für die jeweilige Simulation wichtig sind – zum Beispiel, für welches Jahr der Detektor simuliert werden soll. Die Größe der Konfigurationsdatei beträgt einige Kilobyte.

- H1SIMREC schreibt Informationen über seine Arbeit einerseits nach StdOut[8], andererseits legt es eine Logdatei mit einer Zusammenfassung an.

[8] kurz für *standard output*. Daten, die nach StdOut geschrieben werden, erscheinen auf der interaktiven Konsole eines Unix-Systems. Es ist auch möglich, sie in eine Datei umzuleiten.

- Die fertig simulierten und rekonstruierten Ereignisse werden in besonderen Outputdateien gespeichert, so genannten DST[9]*-Dateien*. Ihre Größe beträgt bei 10.000 Ereignissen etwa 400 Megabyte.

1.3.4 Die H1-MONTECARLO-Kette

Die MONTECARLO-Großproduktion des H1-Experiments lässt sich als Kette beschreiben. Zunächst teilen die Analysegruppen dem „MONTECARLO-Koordinator“ mit, welche Eingangsdateien prozessiert werden sollen. Daraufhin fasst der Koordinator den Auftrag als *Request* zusammen und speichert ihn in der H1-MONTECARLO-Datenbank[10]. Weitere Mitarbeiter lesen die neuen Requests aus der Datenbank und bereiten sie soweit vor, dass sie auf einer Rechnerfarm bearbeitet werden können. Da ein Ereignis nicht mit einem anderen korreliert ist, ist es möglich, H1SIMREC parallel auszuführen. So können zum Beispiel die ersten 10.000 Ereignisse vom ersten Farmrechner bearbeitet werden, die nächsten 10.000 vom zweiten und so weiter[11]. Also müssen die Mitarbeiter entscheiden, wie viele Ereignisse pro Farmrechner bearbeitet werden sollen. Dann wird der Request zu einer Farm geschickt, zum Beispiel zur DESY-eigenen, oder zur Farm des Rutherford Appleton Laboratory (RAL) in Großbritannien. Später werden die DST-Dateien im DESY-Tapepool gespeichert, von wo aus sie den Analysegruppen zur Verfügung stehen. Auf diese Weise wurden in den Jahren 1999 bis 2005 insgesamt $1,4$ Milliarden Ereignisse produziert, aufgeschlüsselt wie Abbildung 1.6 zeigt. Man erkennt deutlich, dass die Ausbaustufe HERA-II (ab dem Jahr 2003) in den Folgejahren wesentlich mehr MONTECARLO-Ereignisse erforderlich machte.

Die soeben vorgestellte Produktionskette nutzt durchweg etablierte Techniken und Strukturen. Um sie von der in dieser Arbeit entwickelten gridbasierten MONTECARLO-Produktion abzugrenzen, heiße sie im Folgenden die *klassische* MONTECARLO-*Produktion*. Das nächste Kapitel 1.4 stellt die neue Gridtechnologie vor, die nicht nur in der Teilchenphysik große Aufmerksamkeit genießt.

[9] Data Summary Tape

[10] https://www-h1.desy.de/icgi-mc/viewRequest (nur DESY-intern)

[11] Es kann vorkommen, dass dabei zwei Prozesse denselben Startwert für den Zufallszahlengenerator verwenden. Solche Effekte sind hierbei aber vernachlässigbar.

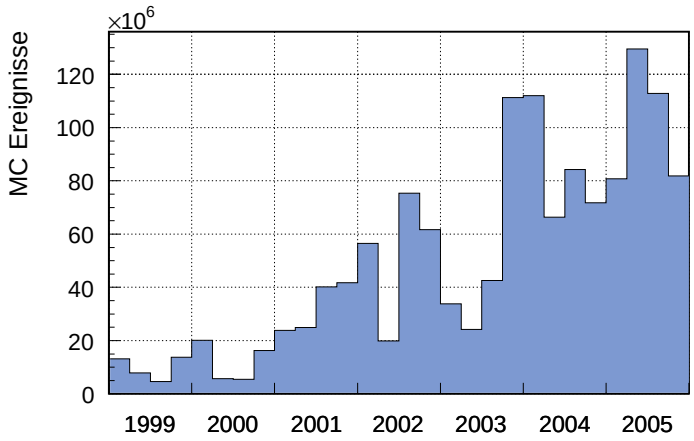

Abbildung 1.6: Simulierte MONTECARLO-Ereignisse. [Quelle: H1-MONTECARLO-Datenbank]

1.4 Grid Computing

Heutzutage ist es selbstverständlich geworden, an fast jedem Punkt der Erde Zugriff auf nahezu unbegrenzte Datenmengen zu haben. Diese Daten werden vom Internet bereit gestellt, und eigentlich ist es gar nicht so wichtig, wo genau sie gespeichert sind. Man findet sie eben „im Internet“. Besser wäre es noch, wenn an fast jedem Punkt der Erde auch nahezu unbegrenzte Rechenleistung zur Verfügung stünde – genau das ist das Konzept des *Grid Computings* [12]. Der Name entstand in Analogie zum Stromnetz, das auf Englisch *power grid* genannt wird. Die Analogie besteht darin, dass komplexe Vorgänge vor dem Anwender verborgen werden. Denn wer eine Steckdose benutzt, ist in der Regel nicht daran interessiert, wie der Strom erzeugt wird und in diese gelangt. Auf ähnliche Weise soll es dem Benutzer des Computing Grids möglich sein, Rechenleistung zu nutzen ohne zu wissen, wie diese produziert wird. In den meisten heute existierenden Grids funktioniert das, indem der Benutzer seine Aufgabe als so genannten *Job* definiert. Diesen schickt er dann ins Grid. Nach einiger Zeit ist der Job bearbeitet und die Ergebnisse stehen bereit, so dass der Benutzer sie abholen kann. Was verborgen bleibt, ist, wie genau das Grid den Job bearbeitet. Mit Fragen wie *Auf welchem Rechner lief der Job?* oder *Wo hat er seine Daten herbekommen?* braucht sich der Benutzer im Idealfall nicht zu beschäftigen. Das Grid ist also ein Modell für verteiltes Rechnen, das jedem einzelnen Benutzer eine einheitliche Schnittstelle zu Ressourcen wie Rechenleistung, Netzwerk- und Datenverwaltungsdiensten bietet.

Ein Grid besteht aus vielen verschiedenen Komponenten, deren Zusammenspiel

von der so genannten *Middleware* geregelt wird. Sie ist die Mittelschicht zwischen der Hardware des Grids und der Software der Benutzers. Grundlage für die Middleware vieler heutiger Grids ist das *Globus Toolkit* [13]. Es steht unter Open-Source-Lizenz und hat sich als Quasistandard für Grids im Forschungsbereich etabliert.

1.4.1 Das LHC-Computing-Grid

Das Grid, das in dieser Arbeit verwendet wird, ist das LHC-*Computing-Grid* [14, 15], kurz LCG[12]. Das LCG-Projekt entstand, um die nötige Rechnerinfrastruktur für die kommenden Experimente am *Large Hadron Collider* (LHC) zu schaffen. Denn der LHC wird der weltgrößte Teilchenbeschleuniger sein, und als solcher wird er ganz besondere Anforderungen stellen. Momentan läuft die Aufbauphase auf dem Gelände des CERN[13] bei Genf, und es ist geplant, ihn ab 2007 zu betreiben. Ab dann werden die LHC-Experimente ATLAS, ALICE, CMS und LHCb enorme Datenmengen produzieren. Erwartet werden mehrere Petabyte pro Jahr, was entschieden mehr ist, als mit Mitteln einer einzigen Institution verarbeitet werden kann. Diese Daten zu speichern und zu analysieren wird die Aufgabe von LCG sein, das die Arbeit auf viele Institute in vielen verschiedenen Ländern verteilen wird.

Die Middleware des LCG entstand aus einer Reihe von anderen Projekten, wie DataGrid, das oben genannte Globus und das EGEE[14]-Projekt. Letzteres wird stark von der Europäischen Union gefördert, und ging seinerseits aus einem Projekt namens *European Data Grid* (EDG) hervor. Zum Zeitpunkt dieser Arbeit beinhaltet es 25.871 Prozessoren und $3,5$ Petabyte nutzbares Datenvolumen[15]. Abbildung 1.7 zeigt eine Karte der vielen an LCG teilnehmenden Institute, und Tabelle 1.4.1 die für H1 zugänglichen Ressourcen.

Im LCG wird den teilnehmenden Instituten ein besonderer Rang zugeordnet: *Tier 0* bis *Tier 3* (*tier* ist Englisch für *Rang*). Diese Titel spiegeln die Leistungsfähigkeit im Sinne der LHC-Datennahme und -analyse wieder. Folgerichtig ist das Tier-0-Zentrum das CERN selbst. Dann kommen die Tier-1-Zentren, eines pro Land oder Region, zum Beispiel GridKa in Karlsruhe für Deutschland. Sie halten Sicherheitskopien der LHC-Rohdaten auf Bandspeicher vor. Außerdem sollen hier zentralisierte Berechnungen, die einen Großteil der gespeicherten Ereignisse betreffen, vorgenommen werden. Die kleineren Tier-2-Zentren, wie zum Beispiel

[12] Im Folgenden werden daher die Begriffe „LCG“ und „Grid“ synonym verwendet.
[13] Conseil Européen pour la Recherche Nucléaire
[14] Enabling Grids for E-sience in Europe
[15] im März 2006

Abbildung 1.7: Karte der an LCG teilnehmenden Institute, Stand März 2006 [16].

DESY, sind für die MONTECARLO-Produktion zuständig. Außerdem laufen hier die Analysen der Arbeitsgruppen. Tier-3-Zentren sind die übrigen Institute.

Die Benutzer des LCG sind in so genannte virtuelle Organisationen (*Virtual Organizations, VOs*) eingeteilt. In der Regel sind das reale Kollaborationen wie zum Beispiel die H1-Kollaboration am DESY – sie benutzt die virtuelle Organisation *hone*[16]. Ebenso gibt es virtuelle Organisationen aber auch für die Teilnehmer von Schulungen oder für die Grid-Administratoren. Die an LCG teilnehmenden Institute entscheiden, welche virtuelle Organisationen sie unterstützen wollen. So haben ausschließlich Benutzer der unterstützten virtuellen Organisationen Zugriff auf die Ressourcen dieser Institute. Ein zentraler Punkt in Grids ist die Sicherheit gegenüber Fremdzugriff. Zu diesem Zweck werden in LCG von einer vertrauenswürdigen Institution Zertifikate[17] an die Benutzer vergeben. Bevor ein Nutzer Zugriff auf die Gridressourcen bekommt, muss er sich authentifizieren, indem er mit seinem Zertifikat eine *Grid-Proxy*-Datei erstellt. Diese wird dann automatisch

[16] *to hone* – Englisch für *verbessern*
[17] In LCG werden X.509-Zertifikate verwendet [15].

Grid-Site	CPUs	Speicherplatz
Universität Dortmund	54	340 GB
Deutsches Elektronen Synchrotron (DESY)	204	123 TB
Rutherford Appleton Laboratory (RAL)	966	6 TB
Birmingham	126	1,6 TB
Queen Mary	558	15,1 TB
Krakau	86	2,4 TB
Kosice	14	550 GB
Swiss National Supercomputing Centre (CSCS)	36	448 GB
Marseille	28	361 GB
Lancaster	386	908 GB
Prag	198	2,4 TB
University College London (UCL)	156	820 GB
Oxford	74	1,5 TB
Summe	**2886**	**155,3 TB**

Tabelle 1.2: Zum Zeitpunkt dieser Arbeit (Stand Januar 2006) hatte das H1-Experiment mit der VO *hone* Zugriff auf die aufgelisteten Ressourcen. Der Speicherplatz am DESY versteht sich inklusive des Tapepools.

mit den Jobs mitgeschickt, und hilft ihnen, alle Sicherheitsprüfungen zu bestehen. Grid Proxies haben eine begrenzte Lebensdauer.

1.4.2 Die Struktur des LCG

Die zentralen Komponenten des LCG, die zum Verständnis dieser Arbeit nötig sind, werden nun vorgestellt. Zunächst einmal gibt es die *Grid-Sites*. Das sind kleine oder große Institute, die dem Grid Rechnerkapazität und Speicherplatz zur Verfügung stellen. Eine ordentliche Grid Site besteht aus einer Rechnerfarm, einem Speicherpool und einem oder mehreren Zugangsrechnern. Letztere sind die Maschinen, mit denen ein Benutzer das Grid bedient. Die Ressourcen einer Grid-Site sind über so genannte *lokale* Dienste erreichbar. Im Gegensatz dazu gibt es *gridweite* Dienste, die die Zusammenarbeit der vielen Grid-Sites koordinieren. Auch wenn die gridweiten Dienste nur ein einziges Mal vorhanden sein müssen, so werden sie in der Praxis dennoch von mehreren Grid-Sites angeboten, um die Ausfallsicherheit zu steigern.

Abbildung 1.8 zeigt eine abstrakte Struktur des LCGs. Der vielleicht wichtigste gridweite Dienst ist die Ressourcenzuteilung. Sie teilt den Jobs Ressourcen zu, indem sie entscheidet, an welche Grid-Site ein Job geschickt wird. Dabei verwendet

sie Informationen über den aktuellen Status der Sites, die der gridweite Informationsdienst gesammelt hat. Der gridweite Dateikatalog schließlich indiziert sämtliche Daten, die in den Speicherpools der Grid-Sites abgelegt sind. Wegen dieser Dienststruktur und des zertifikatbasierten Sicherheitskonzepts ist es nicht notwendig, dass LCG-Benutzer an den Grid-Sites ein lokales Benutzerkonto haben. Das vereinfacht es enorm, die Grid-Sites zu administrieren. Abschließend sei erwähnt, dass zum jetzigen Zeitpunkt das gesamte LCG unter dem Betriebssystem Linux[18] läuft. Im Folgenden werden den bisher abstrakt gebliebenen Diensten Namen gegeben.

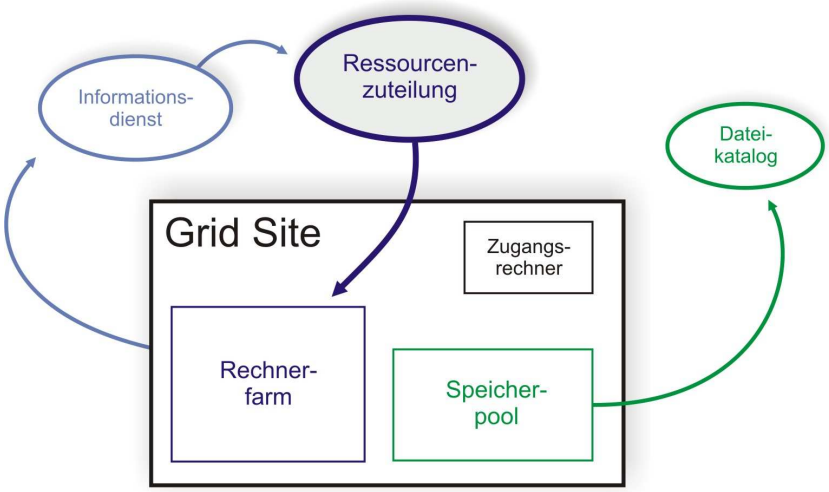

Abbildung 1.8: Abstrakte Struktur der Komponenten in LCG

User Interface (UI)

Das User Interface ist der Zugangsrechner und gewissermaßen das Tor zum Grid (siehe auch Abbildung 1.9). Es ist ein Rechner, auf den die LCG-Benutzer Zugriff haben und auf dem alles Nötige installiert ist, um mit dem Grid zu arbeiten. Zunächst muss sich der Benutzer auf dem User Interface gegenüber dem Grid authentifizieren, danach stehen ihm alle Dienste zur Verfügung. Die Dienste sind über Kommandozeilenbefehle zugänglich. Mit diesen kann man Jobs abschicken und betreuen, Daten transferieren und Informationen über das Grid abrufen.

[18] Sientific Linux 3 [17]

Resource Broker (RB)

Der Resource Broker stellt die Ressourcenzuteilung des Grids dar. Er nimmt die Jobs entgegen, die vom User Interface abgeschickt werden, und sucht passende Grid-Sites, die den Job annehmen können. Dabei berücksichtigt er die Bedürfnisse des Jobs und die der Rechnerfarmen an den Grid-Sites gleichermaßen. Welches die Bedürfnisse des Jobs im Einzelnen sind, kann der Benutzer zu einem gewissen Grad beeinflussen (Grid-Jobs werden detailliert im Kapitel 1.4.3 eingeführt). Idealerweise würde der Job etwa an diejenige Site gehen, auf deren Speicherpool alle nötigen Daten lagern, deren Farm ausreichend lange Rechenzeit zur Verfügung stellt und auf deren Farmrechnern vom Job gewünschte Spezialsoftware installiert ist. Außerdem speichert der Resource Broker die Konfigurationsdateien[19] des Jobs solange, bis sie zu einer Site kopiert wurden, und die Logdateien[20] des Jobs solange, bis sie vom Benutzer zurück auf das User Interface geholt wurden. Die dritte Aufgabe des Resource Brokers ist es, den Status der Jobs bereitzuhalten. Wenn der Benutzer nach diesem fragt, antwortet der Resource Broker.

Zusätzlich ist oft auch der gridweite Informationsdienst (*Berkeley Database Information Index*, BDII) auf der gleichen Maschine wie der Resource Broker implementiert. Er kommuniziert mit untergeordneten lokalen Informationsdiensten der Grid-Sites (Site BDII, *Grid Resource Information Service*, *Grid Index Information Server*) und stellt die vom Resource Broker benötigten Informationen über die Sites zusammen.

Computing Element (CE)

Ein Computing Element ist ein Rechner, der den Zugang zu den Farmrechnern einer Grid-Site bereitstellt. Es bekommt Jobs vom Resource Broker zugewiesen, prüft sie, und leitet sie an einen freien Farmrechner weiter. Jede Grid-Site besitzt mindestens ein Computing Element. Letztlich arbeitet auf dem Computing Element ein klassisches Batchsystem[21], das mit verschiedenen zusätzlichen Diensten ausgerüstet wurde, um mit dem Resource Broker zu kommunizieren. Ein Computing Element wird über die gridweit eindeutigen Namen seiner *queues* angesprochen. Die *queues* sind Warteschlangen für die Jobs. An der einen Seite der Schlange werden sie vom Resource Broker angereiht, an der anderen von den Farmrechnern abgearbeitet. In der Regel haben solche Warteschlangen eine Zeitperiode konfiguriert, nach deren Ablauf die laufenden Jobs abgebrochen werden. Viele Computing Elements bieten mehrere *queues* mit verschiedenen Zeitlimits

[19] Input Sandbox, siehe Kapitel 1.4.3
[20] Output Sandbox, siehe ebenfalls Kapitel 1.4.3
[21] Möglich sind hier zum Beispiel das Portable Batch System (PBS) oder LSF.

an. Die *queue* des Dortmunder Computing Elements, die für das H1-Experiment vorgesehen ist, hat beispielsweise ein Zeitlimit von 48 Stunden und heißt

```
e5grid06.physik.uni-dortmund.de:2119/jobmanager-lcgpbs-hone.
```

Worker Node (WN)

Der Worker Node ist das „Arbeitstier“ im Grid. Er ist derjenige Farmrechner, der vom Computing Element den Job zugewiesen bekommt, und der ihn letztlich ausführt. Ein Worker Node ist gewissermaßen ein herkömmlicher Farmrechner, der über zusätzliche Software zur Kommunikation mit dem Computing Element verfügt. Der Worker Node stellt dem Job eine Standardumgebung zur Verfügung, zu der zum Beispiel ein freies Verzeichnis mit etwa fünf Gigabyte Festplattenplatz gehört, gewisse Standardsoftware (wie etwa der Skriptinterpreter PERL) und die wichtigsten der Kommandozeilenbefehle, die auch schon auf dem User Interface vorhanden sind. Dazu ist der Worker Node mit einem schnellen Netzwerk an den lokalen Speicherpool angebunden. Ein wichtiger Punkt ist, dass ein Worker Node nicht von außen zu erreichen ist. Die einzige Instanz, die mit ihm von außen kommunizieren kann, ist das Computing Element. Aus diesem Grund kann sich kein LCG-Benutzer auf einem Worker Node einloggen und mit einem Job interagieren.

Storage Element (SE)

Was das Computing Element für Rechenkapazität ist, ist das Storage Element für Speicherplatz. Es regelt den Zugriff auf Speichermedien, die in der Regel von den Rechenzentren der Grid Sites betrieben werden. Dabei können zum Beispiel große Festplattenserver oder Tape Pools Verwendung finden. Das Storage Element entscheidet darüber, auf welchem dieser verfügbaren Speichermedien eine Datei gespeichert wird, die beispielsweise von einem User Interface auf das Storage Element kopiert wird. Jede Grid-Site besitzt mindestens ein Storage Element. Dieses heißt auch das *lokale SE*.

File Catalog

Ein weiterer zentraler Dienst im Grid ist der File Catalog. Das ist eine Datenbank, in der sämtliche im Grid vorhandenen Dateien indiziert sind. So müssen nicht alle vorhandenen Storage Elements abgefragt werden, wenn eine bestimmte Datei gesucht wird. Zusätzlich lassen sich Attribute und Kommentare zu einer Datei im Katalog ablegen. Jede virtuelle Organisation unterhält einen eigenen Katalog. Derzeit verwendet die VO *hone* den so genannten EDG-Katalog. Es ist jedoch geplant, in näherer Zukunft auf den neuen *LHC File Catalog* (LFC) zu wechseln.

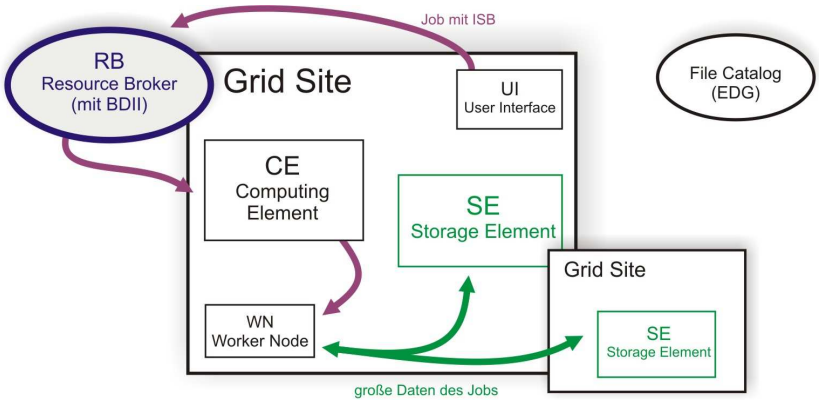

Abbildung 1.9: Die wichtigsten Komponenten des LCG. Dargestellt ist zusätzlich der Weg, den ein Job und seine Input Sandbox nimmt. Große Datenmengen bezieht der Job allerdings nicht über die ISB, sondern von einem Storage Element.

1.4.3 Die Grid-Jobs

Ein Grid-Job wird durch die Angabe bestimmter Informationen definiert. Die wichtigste Information ist der Name des Programms, das auf dem Worker Node gestartet werden soll. Das kann ein Programm sein, das schon auf dem Worker Node installiert ist, oder es ist ein Programm, das mit der so genannten *Input Sandbox* mitgeschickt wurde. Die Input Sandbox ist ein Bündel von Dateien, das zusammen mit dem Job zuerst auf den Resource Broker und von dort auf den Worker Node gelangt (wie in Abbildung 1.9 gezeigt). Typischerweise enthält die Input Sandbox zusätzlich zu dem Programm auch von diesem benötigte Konfigurationsdateien. Die Input Sandbox ist auf eine Maximalgröße limitiert, die vom Resource Broker abhängt. Beim Resource Broker am DESY sind es zum Beispiel 10 Megabyte. Falls der Job größere Datenmengen braucht, muss er sie von einem Storage Element beziehen. Sobald der Job angelaufen ist, wird die Ausgabe des Programms in speziellen Dateien (zum Beispiel *std.out* und *std.err*) gespeichert. Später, nachdem das Programm geendet hat, wird ein Bündel von Dateien zurück zum Resource Broker transferiert. Analog zur Input Sandbox heißt dieses Bündel *Output Sandbox*. Zum Schluss kann der Benutzer die Output Sandbox abholen und den Inhalt auswerten. In der Regel enthält sie zumindest obige Ausgabedateien des ausgeführten Programms.

Alle zur Definition eines Jobs nötigen Informationen werden in einer besonderen Datei gespeichert, wofür die so genannte *Job Definition Language* (JDL) [15] verwendet wird. Abbildung 1.10 zeigt ein Beispiel so einer Datei, wie sie

im Rahmen dieser Arbeit verwendet wird. Der Name des auszuführenden Programms ist *h1mcLauncher.sh*, seine Ausgabe geht nach *std.out* und *std.err*. Das Programm wird innerhalb der Input Sandbox zusammen mit nötigen Bibliotheken (*h1mcJobwrapper.tar.gz*) und Konfigurationsdateien zum Worker Node transferiert. Die Output Sandbox enthält die Ausgabedateien des Programms und verschiedene Logdateien. Die Angabe zu *MyProxyServer* betrifft das Sicherheitskonzept des Grids. *Requirements* und *Rank* beeinflussen die Entscheidung des Resource Brokers, an welche Grid-Site dieser Job gehen soll. So bewirkt etwa die hier angegebene `Requirements`-Sektion, dass nur solche Sites ausgewählt werden, deren *queues* ein Zeitlimit von über 1.200 Sekunden haben, und an denen zusätzlich weniger als drei Jobs darauf warten, ausgeführt zu werden.

Ein Job im Grid durchläuft eine Reihe von so genannten LCG *Zuständen*, die sich vom User Interface aus abfragen lassen. Der erste Zustand ist LCG *submitted*, er bezeichnet Jobs, die noch keine Grid-Site zugewiesen bekommen haben. Es geht weiter über LCG *running* und LCG *done*, wonach die Output Sandbox abgerufen werden kann. Danach ist der letzte Zustand erreicht, LCG *cleared*. Eine detaillierte Beschreibung der Zustände und der möglichen Übergänge findet sich im Anhang B.1.

In der Regel gibt man in der JDL-Datei nicht direkt den Namen des Programms an, das die eigentliche Arbeit leisten soll. Meistens braucht man ein Skript, das erst eine gewisse Umgebung bereitstellt, indem es zum Beispiel benötigte Eingangsdaten von einem Storage Element besorgt, bevor es dann die Hauptanwendung startet. Ebenso würde es später die vom Programm erzeugten Daten auf einem Storage Element abspeichern. So ein Skript wird auch *Job Wrapper* genannt.

Sobald sich der Job im Grid befindet, bekommt er einen eindeutigen Bezeichner zugewiesen, die *LCG-Job-Id*. Über diese wird der Job von den Befehlen zur Jobbetreuung auf dem User Interface angesprochen. Falls der Bezeichner verloren gehen sollte, solange der Job noch im Grid ist, ist der Job gewissermaßen verschollen. Denn dann gibt es keinen Weg mehr, seinen Status abzufragen oder die Output Sandbox abzuholen. Ein Beispiel einer LCG-Job-Id ist

```
https://grid-rb0.desy.de:9000/Cj-10LAfWhE1C8Ob5YhM_A.
```

1.4.4 Das Datenmanagement des Grids

Die Idee hinter dem Datenmanagement in LCG ist, dass es möglich sein soll, eine Datei ganz abstrakt „im Grid“ zu speichern. So eine Datei steht dann allen Grid-Sites zur Verfügung. Dieses Konzept ist folgendermaßen umgesetzt: Eine

```
VirtualOrganisation = "hone";
Executable = "h1mcLauncher.sh";
StdOutput = "std.out";
StdError = "std.err";

InputSandbox = {
"/pfad/h1mcLauncher.sh",
"/pfad/h1mcJobwrapper.tar.gz",
"/pfad/checksums.txt",
"/pfad/wrapper.conf",
"/pfad/04_05_ele.simrec.run.stripped"
};

OutputSandbox = {
"std.out",
"std.err",
"statistics.log",
"cmds.log",
"files.lfn",
"files.guid",
"logs.tar.gz"
};

MyProxyServer = "grid-pxy.desy.de";

Requirements = (other.GlueCEPolicyMaxCPUTime >= 1200) \
&& (other.GlueCEStateWaitingJobs < 3);

Rank = "( other.GlueCEUniqueID=="e5grid06.physik.uni-d\
ortmund.de:2119/jobmanager-lcgpbs-hone" ? other.GlueCE\
StateFreeCPUs*10 : other.GlueCEStateFreeCPUs )"
```

Abbildung 1.10: Der Inhalt einer JDL-Datei, die einen H1-MONTECARLO-Grid-Job beschreibt.

Datei im Grid hat einen eindeutigen Dateinamen, die so genannte GUID (*Grid Unique IDentifier*)[22]. Natürlich muss die Datei letzten Endes auf einem Rechner (auf einem Storage Element) gespeichert sein, wofür sie einen lokalen Dateinamen benötigt. Dieser wird *Storage URL* (SURL) genannt. Damit eine Datei möglichst schnell an allen Sites verfügbar ist, lassen sich Kopien auf weiteren Storage Elements speichern. Diese Kopien heißen *Replikate* und haben alle dieselbe GUID, aber unterschiedliche SURLs. Zusätzlich kann jede Datei im Grid noch eine (oder mehrere) LFNs (*Logical File Name*) besitzen. Im Gegensatz zu den anderen Namen sind die LFNs frei wählbar. Über sie wird der Benutzer die Griddatei ansprechen wollen. Abbildung 1.11 verdeutlicht die Relationen zwischen diesen Bezeichnungen. Alle Dateinamen sind im File Catalog gespeichert. Auf einem User Interface und einem Worker Node sind Befehle installiert, mit denen tief in dieses Gefüge eingegriffen werden kann. So kann man mit ihnen zum Beispiel Dateien löschen, ohne sie aus dem Katalog auszutragen, oder auch umgekehrt.

[22] Es hat sich eingebürgert, *die* GUID als weiblich zu sehen. Gleiches gilt für *die* LFN.

Diese Befehle werden *low level grid tools* genannt. Sie sollten wegen der Gefahr von Inkonsistenzen nur im Ausnahmefall benutzt werden.

Für den eigentlichen Datentransfer werden im Grid besondere Protokolle wie *gridFTP* benutzt. Diese leisten ähnliches wie zum Beispiel das bekannte FTP[23]-Protokoll. Die Protokolle stammen aus dem Globus-Toolkit.

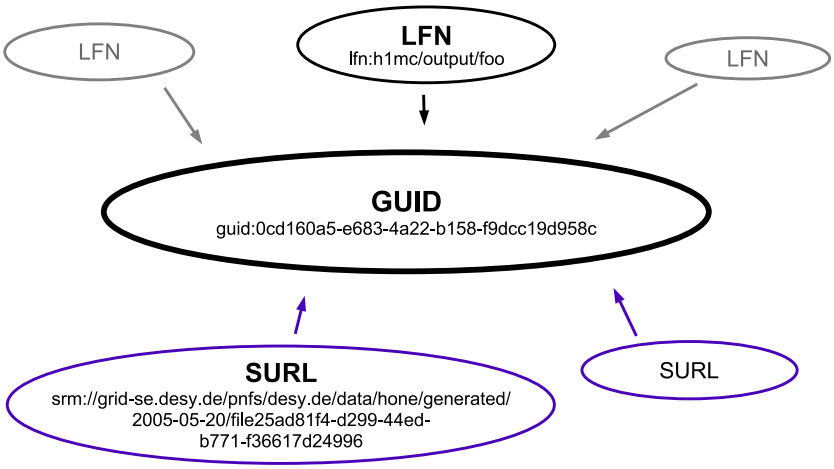

Abbildung 1.11: Relationen zwischen den verschiedenen Dateinamen im Grid

1.4.5 Über die Gridphilosophie

Wie zuvor schon angeklungen, gelten in einem idealen Grid Richtlinien, die als „Gridphilosophie“ zusammengefasst werden sollen. Leider ist LCG (noch) kein ideales Grid, weshalb an einigen Stellen dieser Arbeit gegen die Gridphilosophie verstoßen werden muss.

Das Grid entscheidet, wo ein Job gerechnet wird.

Die Entscheidung, wo ein Job gerechnet wird, trifft der Resource Broker. Allerdings ist seine „Intelligenz“ noch sehr unausgereift, so dass man oft detailliert vorschreiben muss, an welchen Sites ein Job laufen darf und an welchen nicht. Insbesondere wird die Verteilungssituation der vom Job benötigten Daten noch recht statisch berücksichtigt. Das bedeutet, dass bisher nur berücksichtigt wird, an welchen Grid-Sites die Daten vorliegen. Es gibt allerdings laufende Projekte (u.A. D-Grid [18]), die dynamische Lösungen zum Ziel haben. Diese könnten

[23] File Transfer Protokoll

zum Beispiel auch den Aufwand berücksichtigen, der nötig ist, eine entfernte Datei zur ausgewählten Grid-Site zu kopieren – oder den Kopiervorgang sogar selbst einleiten.

Das Grid entscheidet, welche Datei wo vorgehalten wird.

Bis jetzt bleibt diese Entscheidung vollständig dem Benutzer überlassen. Er muss sich darum kümmern, dass von seinen Dateien an den richtigen Storage Elements Replikate vorliegen.

Das Grid entscheidet, welches Replikat bei einem Datentransfer benutzt wird.

In der Theorie sollte es vollkommen ausreichen, die abstrakten LFNs zu verwenden, wenn eine Datei kopiert werden soll. In der LCG-Version 2.4, die bis August 2005 aktuell war, wurde dann allerdings dasjenige Replikat angefordert, das zufällig (!) als erstes gelistet wird. Dadurch wurde der Datentransfer unerträglich langsam. In der jetzigen Version 2.6 ist das Verhalten aber deutlich besser geworden, wie in Kapitel 3.2.1 gezeigt wird.

Man kann sich nicht auf einem Worker Node einloggen.

Anders als viele Benutzer es von ihren klassischen Rechnerfarmen gewohnt sind, kann man sich im Grid nicht auf einem Worker Node einloggen. Somit kann man nicht direkt nachsehen, ob der Job noch sinnvolles tut. Daher müssen Grid-Jobs enorm fehlertolerant sein, womit sich Kapitel 3 befasst. Der Vollständigkeit halber sei erwähnt, dass es ein datenbankbasiertes System namens RGMA [19] gibt, mit dessen Hilfe eine Kommunikation mit Gridjobs aufgebaut werden kann. Das war im Rahmen dieser Arbeit allerdings nicht notwendig, da auch die gewählten Lösungen zufriedenstellend arbeiten.

Für den Datentransfer sind die Storage Elements zu verwenden.

Der Datentransfer über die Storage Elements ist auch mit aktuellen LCG-Versionen noch schwierig. Daher ist die Versuchung groß, die Input Sandbox und die Output Sandbox zum Transport großer Dateien zu missbrauchen. Die Sandboxes sind jedoch ausschließlich für Logdateien gedacht. Da unter Umständen sehr viele Sandboxes auf dem Resource Broker zwischengespeichert werden müssen, liegt es auf der Hand, dass jede einzelne nicht allzuviel Platz beanspruchen darf.

Kapitel 2

Die H1-MonteCarlo-Produktion im Grid

Im vorigen Kapitel wurde zunächst die klassische MonteCarlo-Produktion vorgestellt, anschließend wurde in die Gridtechnologie eingeführt. Im Rahmen dieser Arbeit wurde erforscht, wie sich beide Bereiche zusammenbringen lassen, wie sich also eine MonteCarlo-Produktion im Grid realisieren lässt. Die Ergebnisse werden nun diskutiert.

Eine Gridfarm unterscheidet sich in zentralen Punkten von einer klassischen Rechnerfarm, die für H1-MonteCarlo verwendet wird. Diese Unterschiede stehen einer direkten Adaption der klassischen Methoden im Weg, und müssen folglich überwunden werden. Zunächst einmal befindet sich auf einem Worker Node nur eine standardisierte Softwareumgebung, auf einem Rechner einer H1-Farm hingegen ist alles verfügbar, was H1simrec benötigt (vergleiche Kapitel 1.3.4). Man braucht also ein Job-Wrapper-Skript, das auf einem Worker Node die H1-Umgebung aufbaut, bevor es H1simrec startet. Weiterhin ist es nicht möglich, als Gridbenutzer auf einen Worker Node zuzugreifen, weshalb man auch nicht direkt interagieren kann, falls ein Fehler auftritt.

Nun läuft H1simrec unter Umständen sehr instabil, gerade wenn es sich um „aktuellere“ Versionen handelt, in denen oft noch unentdeckte Programmierfehler stecken. Allerdings enthalten im Falle eines Absturzes die bis dahin produzierten Dst-Outputdateien meist schon nutzbare Ereignisse. Diese lassen sich auf einer klassischen Farm manuell retten, auf einem Worker Node muss dies autonom durch den Wrapper geschehen. Zur Umgebung von H1simrec gehört auch eine Verbindung zur zentralen H1-Datenbank mit Detektorgeometrie und -kalibrierungsdaten. Auch wenn die Datenbank über einen speziellen Datenbankserver prinzipiell von einem Worker Node aus erreichbar wäre, so ist das doch nicht machbar. Denn der Datenbankserver ist nicht dafür ausgelegt, Anfragen von

vielen hundert Jobs gleichzeitig zu bearbeiten. Glücklicherweise bietet H1SIMREC die Möglichkeit, stattdessen mit einer Datenbankdatei zu arbeiten. Diese enthält normalerweise den gesamten Inhalt der zentralen H1-Datenbank. Die Datei kann, genau wie die Noise Files, von einem Storage Element bezogen werden.

Der nächste Unterschied betrifft die Art und Weise, wie die DST-Outputdateien in den DESY-Tapepool gelangen. Klassische H1-Farmen haben direkten Zugriff auf diesen Pool, somit können die Outputdateien direkt dorthin kopiert werden. Die Gridproduktion muss dafür das wichtigste Storage Element am DESY verwenden, das

`srm-dcache.desy.de`.

Es ist direkt mit dem Tapepool verbunden, wodurch es einerseits alle Daten, die im Pool gespeichert sind, im Grid zur Verfügung stellt. Andererseits legt es Daten, die es aus dem Grid empfängt, automatisch im Tapepool ab. Dort können sie wie gewohnt von den Analysegruppen gelesen werden.

Es war der erste Teil dieses Projekts, einen geeigneten Job-Wrapper zu entwickeln. Er wird im folgenden Kapitel 3 vorgestellt. Der zweite Teil widmet sich der Aufgabe, eine Vielzahl von Grid-Jobs zu betreuen. Anders als klassische Farmsysteme (wie etwa dem *Portable Batch System*) stellt die LCG-Middleware keine Hilfsmittel zur Verfügung, die den Status vieler Jobs erfassen können. Außerdem ist der Teil der erfolglosen Jobs im Grid deutlich höher als bei klassischen Farmen, da es ungleich mehr Fehlerquellen gibt. Also gehört es auch zur Betreuung der Jobs, die erfolglosen erneut abzuschicken. Das hier entwickelte *Grid Batch System* erfüllt diese Aufgabe und wird im Kapitel 4 beschrieben.

2.1 Direktiven

Aus den bisher dargelegten Überlegungen und der Gridphilosophie (vergleiche Kapitel 1.4.5) wurden Direktiven entwickelt, die eine effiziente MONTECARLO-Produktion im Grid möglich machen. Diese sind in folgender Liste zusammengefasst.

- *Die Laufzeit eines* LCG*-Jobs beträgt* 15 *Stunden.* Längere Jobs sind nicht wünschenswert, da relativ schnell auf Veränderungen reagiert werden muss. Außerdem kann man erst nach Ablauf dieser Zeit wirklich sicher sein, ob der Job alles richtig gemacht hat, denn vorher hat man keine Möglichkeit, mit ihm zu kommunizieren. Hätte ein fehlerhafter Job eine Laufzeit von beispielsweise 48 Stunden, wie bei klassischer Produktion üblich, würde er entsprechend länger Ressourcen blockieren.

- *Die* MONTECARLO*-Eingangsdateien werden gestückelt.* Die angegebene Laufzeit reicht, um etwa etwa 10^4 Ereignisse zu prozessieren. Daher werden die Eingangsdateien in kleinere Dateien zerteilt, die jeweils 10^4 Ereignisse enthalten. Eingangsdateien werden komprimiert.

- *Die* MONTECARLO*-Eingangsdateien werden auf ein (einziges) Storage Element kopiert, vorzugsweise das* `srm-dcache.desy.de`. Von dort aus aus holt sie der Job-Wrapper auf einen Worker Node. Falls die komprimierten Eingangsdateien klein genug sind, dürfen sie auch mit der Input Sandbox zum Worker Node gelangen.

- *Die Noise Files sind auf allen zugänglichen Storage Elements gespeichert.* Die Verteilung der Noise Files wird manuell gesteuert, da sie nur selten aktualisiert werden müssen.

- *Die Datenbankdatei ist auf allen zugänglichen Storage Elements gespeichert.* Die Datenbankdatei enthält dabei nicht den gesamten Inhalt der zentralen H1-Datenbank, sondern nur den Teil, der von dem aktuellen Request gebraucht wird. Welche das sind, richtet sich nach den Datennahmeperioden, für die Ereignisse simuliert werden sollen. Für jede Periode liegt eine separate Datenbankdatei vor, die beim Erzeugen des Requests manuell ausgewählt werden muss. Auch wenn die Datenbankdateien öfter aktualisiert werden müssen als die Noise Files, so lässt sich auch ihre Verteilung gut manuell steuern.

- *Das* H1SIMREC*-Programm ist als „*H1SIMREC*-Paket“ auf allen zugänglichen Storage Elements gespeichert.* Es enthält zusätzlich zum Programm auch alle benötigten Systembibliotheken.

- *Jeder Grid-Job erzeugt* DST*-Outputdateien von maximal* 400 *Megabyte Volumen.* Die genaue Größe hängt einerseits von der Anzahl der Ereignisse ab. Andererseits spielt der simulierte physikalische Prozess eine Rolle, insbesondere die Zahl der in den Ereignissen enthaltenen Spuren. Die Schwankungen bewegen sich im Bereich eines Faktors 2. Sollte eine Outputdatei von H1SIMREC größer als 400 Megabyte werden, wird sie noch auf dem Worker Node in Stücke zerteilt, die jeweils kleiner als 400 Megabyte sind. In diesem Fall gibt es mehrere Outputdateien pro Job, weshalb alle Teile des Systems auch mit mehreren Dateien umgehen können müssen.

- *Grid-Jobs schicken ihre* DST*-Outputdateien auf irgendein verfügbares Storage Element.* Es wäre unglücklich, wenn der DST-Output nirgendwo gespeichert werden könnte, weil gerade ein bestimmtes Storage Element aus-

gefallen ist. Daher sollte der Job die Möglichkeit haben, mehrere Storage Elements aus einer Liste durchzuprobieren.

- *Die* DST-*Outputdateien werden zurück an das SE* `srm-dcache.desy.de` *repliziert.* Nachdem die Jobs ihre DST-Dateien auf irgendein Storage Element kopieren, müssen die Dateien anschließend zurück ans DESY gelangen. Dafür wird das `srm-dcache.desy.de` benutzt, denn von dort werden die Dateien automatisch im DESY-Tapepool abgelegt.

- *Die* DST-*Outputdateien müssen nach einem bestimmten Schema benannt sein.* Ansonsten sind sie für die meisten Analysegruppen nur von eingeschränktem Nutzen. Das Schema enthält einen fortlaufenden alphanumerischen Index.

2.2 Implementierung

Der grundlegende Ansatz bei der Implementierung sowohl des Job-Wrappers als auch des Grid-Batch-Systems ist, Kommandozeilenbefehle der Middleware skriptgesteuert aufzurufen. Gerade für den Datentransfer sind umfangreiche Programmskripte nötig, die viele verschiedene Befehle ausführen. Als Skriptsprache wurde PERL gewählt, denn einerseits ist der PERL-Interpreter auf den Worker Nodes standardmäßig verfügbar, und andererseits erlaubt PERL eine objektorientierte Programmierung. Ein Modul des Grid-Batch-Systems allerdings, der Job-Maker, ist in der Skriptsprache PYTHON implementiert.

Der Job-Wrapper wird in folgendem Kapitel 3 detailliert beschrieben; Kapitel 4 widmet sich dem Grid-Batch-System.

Kapitel 3

Der H1-MONTECARLO-Grid-Job

Die Aufgabe eines H1-MONTECARLO-Grid-Jobs sei noch einmal kurz umrissen. Er soll die Detektorsimulation H1SIMREC auf einem Worker Node erfolgreich und effizient laufen zu lassen. Ein kompletter MONTECARLO-Request besteht dabei aus sehr vielen dieser Grid-Jobs, und jeder von ihnen bearbeitet einen Teil der Ereignisse, die insgesamt prozessiert werden sollen. Ein Grid-Job ist vollkommen auf sich selbst gestellt, da man vom User Interface aus keine Möglichkeit hat, mit ihm zu kommunizieren. Er muss also selbst die nötige Intelligenz mitbringen, um auf einen Großteil der auftretenden Fehler richtig zu reagieren – seien es nun Fehler, die „das Grid“ verursacht hat oder solche, die sich auf H1SIMREC zurückführen lassen. Zusätzlich soll es möglich sein, im Nachhinein zu rekonstruieren, was tatsächlich auf dem Worker Node vor sich ging. Also sind umfangreiche, aber dennoch übersichtliche Logdateien unumgänglich.

Die grundlegende Struktur eines Grid-Jobs erscheint einfach. Zunächst muss er die nötigen Daten auf den Worker Node kopieren, dann die Anwendung starten und zum Schluss die erzeugten Daten an geeigneter Stelle speichern. Allerdings hat jeder dieser Punkte seine Schattenseiten, die in den folgenden Kapiteln beleuchtet werden (Kapitel 3.2 behandelt den Datentransfer und Kapitel 3.3 das Ausführen von H1SIMREC). Um einen groben Überblick über die Kenndaten eines H1-MONTECARLO-Grid-Jobs zu geben, sind sie in Tabelle 3.1 zusammengefasst. Alle Daten sind Beispiele, die eine Vorstellung von den Größenordnungen geben sollen. Die genauen Werte hängen im Wesentlichen von den Charakteristika des physikalischen Prozesses ab, der simuliert werden soll.

3.1 Ablauf eines H1-MONTECARLO-Grid-Jobs

In diesem Kapitel wird der Ablauf eines H1-MONTECARLO-Grid-Jobs genauer beschrieben. Als Überblick bietet Abbildung 3.1 eine detailliertere Ansicht über

Zahl der Ereignisse	10.000
Laufzeit	15 Stunden
CPU-Zeit pro Ereignis	$1 - 15$ Sekunden
Zeit für den Datentransfer	1 Stunde
Datenvolumen der Eingangsdateien	2 Gigabyte
Datenvolumen der DST-Outputdatei	400 Megabyte

Tabelle 3.1: Größenordnungen eines typischen H1-MONTECARLO-Grid-Jobs. Die große Streuung bei der CPU-Zeit pro Ereignis ist verschiedenen physikalischen Prozessen geschuldet.

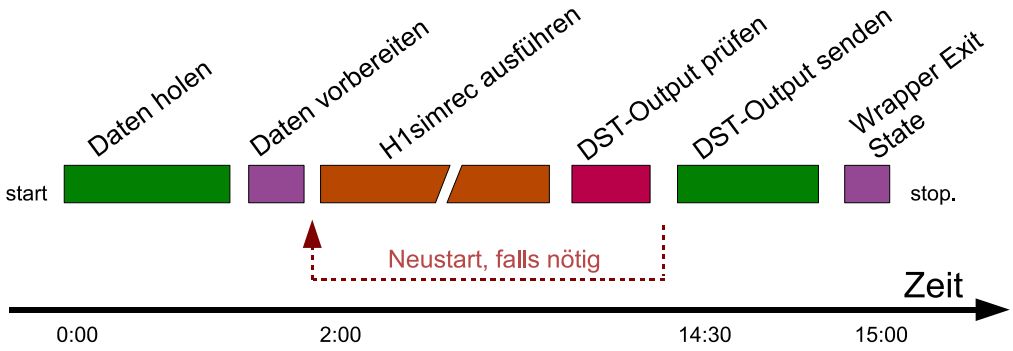

Abbildung 3.1: Ablauf eines H1-MONTECARLO-LCG-Jobs. Nachdem sich der Job alle Daten geholt hat, startet er H1SIMREC. Lief es erfolgreich, kann der DST-Output an ein Storage Element gesendet werden.

den zeitlichen Ablauf des Jobs. Zuerst muss sich der Job die Daten beschaffen, die zum Ausführen der Anwendung benötigt werden. Im Falle von H1SIMREC handelt es sich insgesamt um ein Volumen von etwa zwei Gigabyte. Im Einzelnen sind das das H1SIMREC-Paket selbst, die MONTECARLO-Eingangsdatei, die Noise Files und die Datenbankdatei. All diese Daten werden von einem Storage Element zum Worker Node kopiert. Danach müssen die heruntergeladenen Dateien aufbereitet werden. Zum Beispiel liegen einige von ihnen als komprimierte Tar-Archive[1] vor (in der Regel das H1SIMREC-Paket und die MONTECARLO-Eingangsdatei), die extrahiert werden müssen. Die benötigten Anwendungen und Hilfsprogramme (H1SIMREC selbst, FPLIST[2] und FPACK[3]) benötigen besondere

[1] *Tar* ist ein Standardformat für Archivdateien.
[2] FPLIST ist ein Programm, das die in DST-Dateien enthaltenen Ereignisse zählt.
[3] FPACK ist ein Programm, mit dem sich auf Daten aus DST-Dateien zugreifen lässt.

Systembibliotheken. Daher wird überprüft, ob die Bibliotheken alle zur Verfügung stehen. Wenn das nicht der Fall ist, müssen stattdessen die mitgeschickten Bibliotheken aus dem H1SIMREC-Paket verwendet werden. Jetzt ist alles bereit, um mit der eigentlichen Arbeit zu beginnen, und H1SIMREC zu starten.

Nach einer mehr oder minder langen Zeit ist H1SIMREC beendet, und es wird der erzeugte DST-Output geprüft. Ist er in Ordnung, wird versucht, ihn auf ein Storage Element zu transferieren. Ist der DST-Output hingegen fehlerhaft, wird H1SIMREC neugestartet – näheres dazu in Kapitel 3.3.2. Im nächsten Schritt werden sämtliche Daten, die für eine statistische Auswertung wichtig sind, in einer Datei gespeichert. Besonders interessant sind beispielsweise Laufzeiten, Datentransferraten und die aufgetretenen Fehler. Die Statistikdatei wird mit der Output Sandbox zurückgeschickt und später ausgewertet. Zuletzt wird der *Wrapper-Exit-State* ermittelt. Das ist die wichtigste Information in der Output Sandbox, denn basierend auf dem Wrapper-Exit-State trifft das Grid-Batch-System die Entscheidung, wie mit dem Job weiter zu verfahren ist. Zum Beispiel zeigt der Status `ERRORS_OCCURRED` an, dass zwar Fehler auftraten, der Job aber dennoch zu einem guten Ende gebracht werden konnte. Eine Auflistung der möglichen Wrapper-Exit-States findet sich im Anhang B.3.

3.2 Der Datentransfer

Innerhalb der jetzigen Versionen von LCG[4] ist der Datentransfer noch sehr fehleranfällig, weshalb man nicht davon ausgehen kann, dass ein angeforderter Datentransfer stets erfolgreich und schnell ist. Daher ist es ein wichtiger Teil dieser Arbeit, den Datentransfer einer Gridanwendung zu optimieren. Im einzelnen muss mit einer Vielzahl von Problemen umgegangen werden, wie zum Beispiel den folgenden:

- Der Kopierbefehl bricht mit Fehlermeldung ab.
- Es kommt nur ein Dateifragment an.
- Der Befehl hängt „für immer“ oder beendet sich nach einer gewissen Zeit selbst.
- Es wird ein ungünstiges (weil weit entferntes) Replikat als Quelle genommen.

[4] Im Gebrauch sind die Versionen 2.4.0 und 2.6.0.

Abbildung 3.2 illustriert die Lage anhand der beobachteten Ineffizienz des Middleware-Kommandos `lcg-cp`, das den Datenempfang vom Storage Element implementiert. Die Ineffizienz ist dabei in obige Kategorien aufgeschlüsselt. Die vielen „`lcg-cp` hängt“-Fehler im September wurden von einem ausgefallenen Storage Element verursacht, die vielen Dateifragmente allerdings von einem Fehler im Programmcode des Job-Wrappers, ebenso wie für die vielen „`lcg-cp` hängt“-Fehler im Dezember. Leider wird deutlich, dass sich die Leistung der Middleware in den vergangenen Monaten nicht nennenswert verbessert hat – was die hier entwickelten Methoden um so nötiger macht.

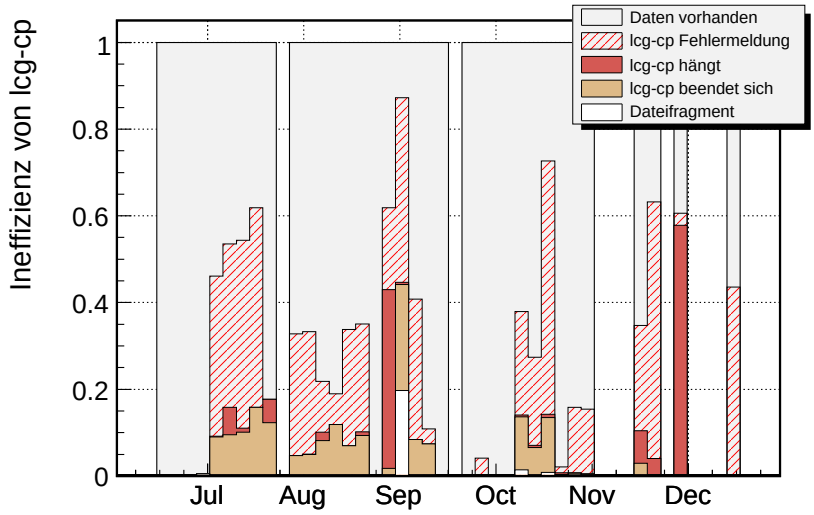

Abbildung 3.2: Ineffizienz des `lcg-cp`-Kommandos, das den Datenempfang vom Storage Element ausführt, aufgeschlüsselt nach verschiedenen Fehlern. Für gewisse Zeiträume stehen keine Daten zur Verfügung.

Daten müssen an vielen Stellen des Projekts transferiert werden – nicht nur innerhalb eines Grid-Jobs. Um nicht die gleichen Probleme mehrfach lösen zu müssen, wird für alle Datentransfers derselbe Programmcode verwendet. In diesem Kapitel werden daher drei Szenarien beschrieben: Datenempfang vom Storage Element (*get*-Funktion, Kapitel 3.2.1), Daten zu einem Storage Element senden (*put*-Funktion, Kapitel 3.2.2) und das Replizieren von Dateien (*replicate*-Funktion, Kapitel 3.2.3). Alle drei Funktionen nutzen jedoch die gleiche einfache Grundstrategie, um mit Fehlschlägen fertig zu werden: erneut probieren. Damit ist zunächst einmal die exakte Wiederholung gemeint; zusätzlich werden aber auch alternative Quellen oder alternative Ziele ausprobiert (falls möglich). Die weiteren Maßnahmen unterscheiden sich je nach Funktion.

Alle drei Funktionen müssen zudem mit einem gemeinsamen Problem umgehen. Wie schon erwähnt kommt es häufig vor, dass ein Gridbefehl „für immer“ hängt. Das betrifft nicht nur `lcg-cp`, sondern auch alle anderen Kommandos der Middleware. Dieses lästige Verhalten macht ein Timeout nötig, nach dessen Ablauf der Befehl zu einem Ende gezwungen wird[5]. Standardmäßig beträgt die Zeitspanne 30 Minuten. Diese harsche Methode ist problematisch, da der jeweilige gridFTP-Server eine gewisse Zeit auf den nicht mehr vorhandenen Klienten wartet, was unnötig Ressourcen belegt. Verzichtet man allerdings auf den Timeout innerhalb eines Grid-Jobs, würde der gesamte Job erst nach Ablauf seines Proxies beendet (üblicherweise 72 Stunden) oder wenn das Zeitlimit der *queue*[6] erreicht wird. Da der gesamte Worker Node solange blockiert wäre, wäre das noch schlimmer als ein wartender gridFTP-Server. Seit LCG 2.6 ist in den meisten Gridbefehlen eine Timeout-Option eingebaut, die sich vom Benutzer konfigurieren lässt. Wenn der Befehl nicht vorher zu einem Ende kommt, beendet er sich selbst. Das ist für den `lcg-cp`-Befehl in Abbildung 3.2 als „`lcg-cp` beendet sich“ notiert. Leider hat sich gezeigt, dass die Kopierbefehle trotz dieser Option gelegentlich hängen bleiben. Also gibt man dem Gridbefehl zunächst die Chance, sich nach Ablauf seines Timeouts selbst zu beenden. Tut er das nicht innerhalb von fünf Minuten, wird er wie gehabt zum Ende gezwungen.

3.2.1 Datenempfang vom Storage Element

Die *get*-Funktion, die den Datentransfer vom Storage Element zum Worker Node ausführt, sei nun beschrieben. Zuerst wird geprüft, ob genug freier Speicherplatz zur Verfügung steht, um die empfangene Datei unterzubringen. Da sich derzeit nicht mit einfachen Mitteln bestimmen lässt, wie groß die zu empfangende Datei sein wird, wird prophylaktisch auf ein freies Gigabyte getestet. Es erstaunt, dass die Dateigröße, die im EDG-File-Catalog enthalten ist, nicht einfach abrufbar ist. Sobald die virtuelle Organisation *hone* allerdings auf den leistungsfähigeren LFC umsteigt, wird diese Information direkt verfügbar sein (siehe auch Anhang A).

Der nächste Schritt dient der Optimierung des Datendurchsatzes. Während des ersten halben Jahres dieser Arbeit lief LCG unter der Middleware-Version 2.4. In dieser Version wird standardmäßig ein zufällig bestimmtes Replikat als Übertragungskandidat ausgewählt, wenn der Benutzer allein mit LFNs arbeitet. Durch die zufällige Wahl des Übertragungskandidaten genießen die lokalen Storage Elements keinerlei Bevorzugung, was die durchschnittliche Übertragungsrate stark begrenzt. Das verdeutlicht Abbildung 3.3. Sie zeigt die effektiven Übertragungsraten eines H1MC-Grid-Jobs, links zu entfernten Storage Elements und rechts zu

[5] mittels `kill -9`

[6] vergleiche Kapitel 1.4.2, „Computing Element“

lokalen[7]. Es wurde jeweils die komplette Transferzeit berücksichtigt, inklusive der durch erfolglose Versuche anfallende Wartezeit. Seit LCG 2.6 fordert `lcg-cp` nun zuerst ein lokales Replikat an, sofern eines vorhanden ist. Wird das gleiche Kommando danach noch einmal aufgerufen, wählt es zufällig ein Replikat auf einem entfernten Storage Element.

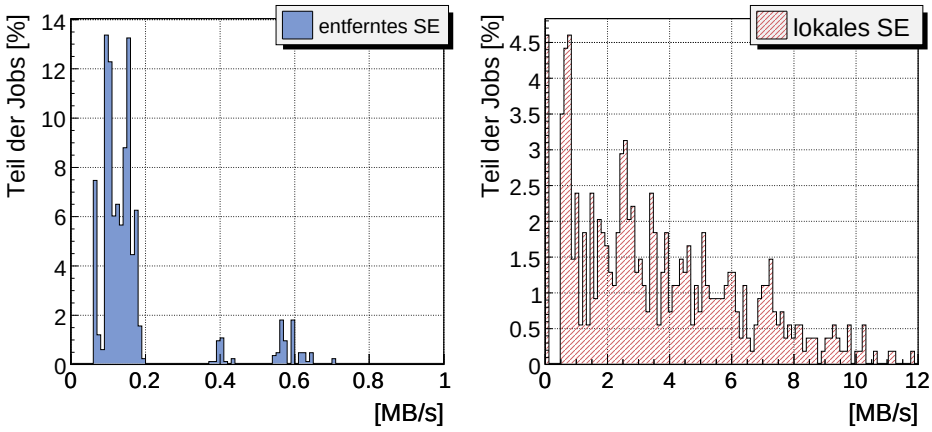

Abbildung 3.3: Datendurchsatz des `lcg-cp`-Kommandos für den Datentransfer von einem Storage Element zum Worker Node.

Im Rahmen dieser Arbeit wurde ein Verfahren entwickelt, das es auch unter LCG 2.4 erlaubt, zunächst ein lokales Replikat zu benutzen. Um die geographische Entfernung abzuschätzen, macht es sich zunutze, dass die SURLs den aus dem Internet bekannten URLs ähneln, also etwa

`protocol://host.site.country`.

Da die „Ortsauflösung“ von rechts nach links besser wird, lassen sich die SURLs der Replikate mit dem Hostname des Worker Nodes vergleichen und so ungefähr nach Entfernung ordnen. Höchste Priorität erhalten Replikate, die an der gleichen Grid-Site gespeichert sind, dann folgen die im gleichen Land, dann die restlichen. Nun wird versucht, das erste Replikat aus der sortierten Liste zu empfangen. So konnten auch unter LCG 2.4 verbesserte Durchsatzraten erzielt werden.

Wenn der Kopierbefehl beendet ist, wird die Integrität der empfangenen Datei geprüft. So wird sichergestellt, dass die empfangenen Dateien auch die sind, für die man sie hält, was grundlegend ist für ein Vertrauen in die Ergebnisse von grid-prozessiertem MONTECARLO. In Abbildung 3.2 ist der Anteil der Fälle zu sehen,

[7] Diese Daten wurden mit der LCG-Version 2.6 gewonnen, decken sich aber mit den Erfahrungen mit Version 2.4.

in denen tatsächlich nur ein Dateifragment ankommt. Es sei bemerkt, dass dieser Fall sporadisch immer wieder auftritt, auch wenn die dazugehörigen kleinen Zahlen nicht deutlich in der Abbildung zu erkennen sind[8]. Zum Prüfen der Integrität wird die Prüfsumme[9] der Datei mit einem Referenzwert verglichen. Stimmen beide überein, war der Transfer erfolgreich. Die Referenzprüfsumme wird schon beim Hochladen der Datei auf ein Storage Element als Attribut der Datei im EDG-Katalog abgelegt. Als Alternative gibt es aber auch die Möglichkeit, die Referenzprüfsumme in einer Datei zu speichern und diese mit der Input Sandbox auf den Worker Node zu schicken. Das ist insbesondere für Testzwecke praktisch. Auf der anderen Seite birgt es jedoch die Gefahr, dass eine veraltete Prüfsummen-Datei benutzt wird.

Falls der Prüfsummentest negativ ausfällt, wird das nächste Replikat aus der Liste der SURLs angefordert. Wenn alle Replikate einmal erfolglos versucht worden sind, wird wieder mit dem ersten Replikat der Liste begonnen. Diese Schleife durchläuft maximal N_{retry}^{max} Wiederholungen, wobei man N_{retry}^{max} in der Konfiguration des Job-Wrappers festlegt.

3.2.2 Transfer vom Worker Node zum Storage Element

Die *put*-Funktion, die den Datentransfer vom Worker Node oder User Interface auf das Storage Element ausführt, ist ebenfalls Teil des in dieser Arbeit entwickelten optimierten Datentransfers. Sie wird in vielen verschiedenen Zusammenhängen benutzt und lässt sich mit entsprechenden Optionen feinkonfigurieren. Erneut wird mit Wiederholungen gearbeitet. Das Kriterium, das dabei Auskunft über den Erfolg gibt, ist die Ausgabe des Kopierbefehls `lcg-cr`[10] – denn dieser gibt im Erfolgsfall die GUID der gespeicherten Datei zurück. In der Regel hat man eine genaue Vorstellung davon, auf welches Storage Element die Datei transferiert werden soll. So gehen beispielsweise die DST-Outputdateien der Jobs möglichst direkt auf das Storage Element `srm-dcache.desy.de` da sie früher oder später sowieso dorthin repliziert werden (vergleiche Kapitel 2). Ebenso gibt es Storage Elements, an die eine Datei *nicht* gehen soll. Zum Beispiel kommt es vor, dass das lokale Storage Element einer Site zu klein ist, um den Output der Jobs aufzunehmen. Aus diesen Gründen wird bei der Produktion einfach eine Liste von Storage Elements spezifiziert, die in gegebener Reihenfolge und eventuell mehrmals durchprobiert wird.

[8] Es handelt sich um insgesamt 97 Fälle aus 11359 Versuchen im betrachteten Zeitraum (Fragmente, die durch hartes Beenden des Transfers entstanden, sind nicht berücksichtigt).

[9] Als Prüfsummenalgorithmus dient das `md5sum`-Programm, das auf allen gängigen Linux-Installationen vorhanden ist.

[10] *cr* steht für *copy and register*

Wenn die *put*-Funktion auf dem Worker Node aufgerufen wird, sollen teuer produzierte Dateien transferiert werden. Das muss unter allen Umständen gelingen, ansonsten würden die Dateien nach dem Ende des Jobs unwiederbringlich gelöscht. Natürlich empfiehlt es sich dafür, die Liste der möglichen Storage Elements möglichst lang zu machen. Zusätzlich aber bietet die *put*-Funktion speziell für diese Situation zwei Notlösungen. Die erste hilft, wenn die gewünschte LFN schon vergeben ist. Das kommt gelegentlich vor, meistens in Folge von Konfigurationsfehlern, die dem Benutzer unterlaufen sind. Es tritt aber auch auf, wenn ein Datentransfer zu einem Ende gezwungen wird, nachdem die Datei schon im File Catalog registriert wurde. Ist die LFN schon vergeben, wird der String „.N“ angehängt, wobei N eine Zahl zwischen 1 und einem konfigurierbaren N_{max} ist. Die zweite Notlösung benutzt die Output Sandbox des Jobs, um die Dateien vom Worker Node ans User Interface zu bekommen. Falls also die *put*-Funktion trotz aller Wiederholungen die Dateien nicht auf ein Storage Element kopieren kann, erzeugt sie ein komprimiertes Tar-Archiv, den so genannten *Rescue Tarball*. Dieser gelangt mit der Output Sandbox zurück auf den Resource Broker, und später von dort ans User Interface, wo der Benutzer die Daten weiterverarbeiten kann. Es sei angemerkt, dass das ein „Verstoß“ gegen die Grid-Philosophie ist, da der Resource Broker nur für verhältnismäßig kleine Logdateien zu verwenden ist. Falls mehrere Jobs ihren Output gleichzeitig vom Rescue Tarball Gebrauch machen, besteht durchaus die Gefahr, dass der Resource Broker schnell aufgefüllt wird. Außerdem ist der spätere Umgang der Rescue Tarballs relativ aufwendig (siehe auch Kapitel 4.7). Dennoch hat die Erfahrung gezeigt, dass $4,6\%$ der Jobs durch diese Maßnahme gerettet werden konnten, was etwa einem CPU-Jahr[11] entspricht. Das ist besonders wertvoll, wenn man bedenkt, dass der Rescue Tarball nur dann benutzt wird, wenn ein wichtiger lokaler oder gridweiter Dienst ausgefallen ist. Dann allerdings haben meistens alle Jobs eines Requests das gleiche Problem. Ohne den Rettungsmechanismus würde man also oft den gesamten Request verlieren. Abbildung 3.4 zeigt die Verwendung des Rescue Tarballs.

Durch viel praktische Erfahrung wurde eine weitere Problematik der Middlewarekommandos für den Datentransfer aufgezeigt: Wenn ein Transfer zu einem Storage Element fehlschlägt, kann es sein, dass auf dem Storage Element schon eine Null-Byte-Datei angelegt wurde. In diesem Fall bricht ein erneuter Transfer mit der sinngemäßen Fehlermeldung „die Datei existiert schon“ ab. Deshalb versucht die *put*-Funktion nach einem fehlgeschlagenen Kopierversuch ein eventuell angelegtes Dateifragment auf dem Storage Element zu löschen. Dazu wird der Befehl `lcg-del` auf die LFN angewandt, was nur funktionieren kann, wenn die Datei schon im Katalog registriert wurde. War das nicht der Fall, hat man keine Chance;

[11] Ein CPU-Jahr entspricht der Arbeit, die eine CPU in einem Jahr zu leisten vermag. Eine im Rechnerfarmbereich übliche Einheit.

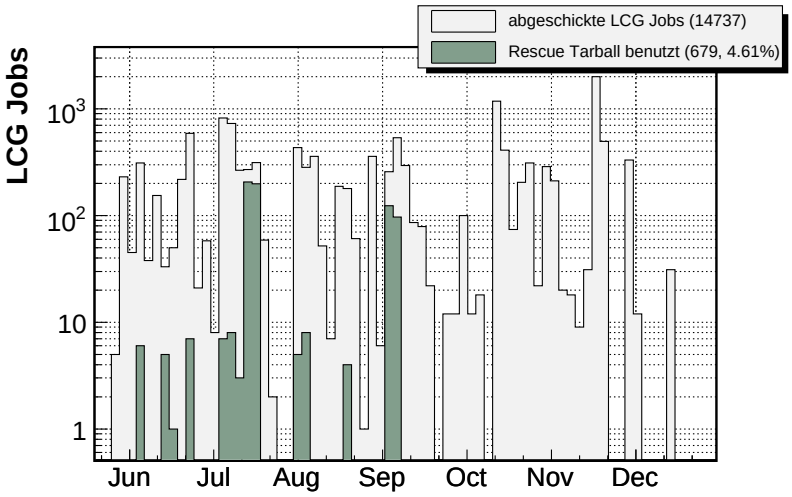

Abbildung 3.4: Verwendung von Rescue Tarballs

im nächsten Versuch würde dann ein anderes Storage Element versucht werden. Es kommt außerdem vor, dass ein Datentransfer zu einem Storage Element hart beendet wird, weil er schlicht zu langsam ist. In so einem Fall reichte die Zeit des konfigurierten Timeouts nicht aus, um mit der verfügbaren Übertragungsrate den Transfer abzuschließen. Dann bleiben in der Regel die bis zu diesem Zeitpunkt transferierten Daten auf dem Storage Element. Das ist besonders ärgerlich, da sich diese Dateifragmente schnell auf mehrere hundert Gigabyte summieren und so wertvollen Speicherplatz sinnlos belegen. Da die Fragmente dann noch keinen Katalogeintrag haben, ist das anschließende Entfernen der Bruchstücke mit *low level grid tools*[12] sehr aufwendig. So muss zunächst der gesamte (!) Inhalt des Storage Elements aufgelistet werden. Für jede dieser SURLs muss dann die GUID ermittelt werden. Möchte man einen sinnvollen Namen sehen, müssen über die GUIDs die LFNs abgefragt werden. Schließlich werden diejenigen SURLs gelöscht, die keine GUID im Katalog haben. Um dem Datentransfer wenigstens eine realistische Chance zu geben, wird die Zeitspanne des Timeouts anhand der Dateigröße und einer zu erwartenden minimalen Durchsatzrate (typischerweise 200 kB/s) berechnet. Zum Abschluss berechnet die *put*-Funktion die MD5-Prüfsumme der Datei und speichert sie im EDG-Katalog.

Die Effizienz der entwickelten Maßnahmen ist erfreulich, wie Abbildung 3.5 belegt. Sie zeigt den Anteil aller abgeschickten LCG Jobs, die ohne diese Maß-

[12] siehe Kapitel 1.4.4

nahmen gescheitert wären, und denjenigen Teil, der trotz aller Bemühungen gescheitert ist. Dabei wird ein Job als „wäre gescheitert“ gezählt, falls er mehr Transferversuche brauchte, als die Zahl der zu transferierenden Dateien angibt. Es sind Daten der *get*- und der *put*-Funktion berücksichtigt. Es fällt auf, dass die Datentransfereffizienz mit der Zeit besser geworden ist. Das lässt sich einerseits mit verbesserter Gridinfrastruktur und Middleware erklären. Andererseits wurde bei jüngeren Produktionen darauf geachtet, dass viele der benötigten großen Dateien (wie Noise Files und die Datenbankdatei, vergleiche Kapitel 2.1) auf jedem lokalen Storage Element im Grid vorliegen.

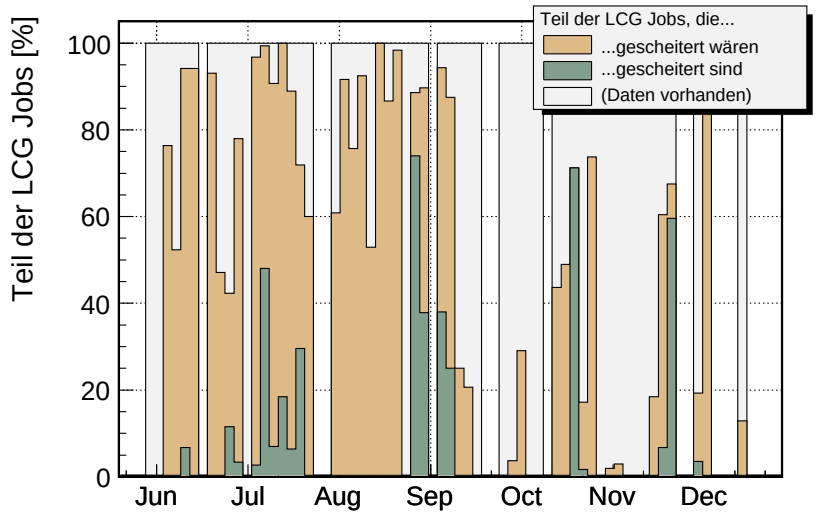

Abbildung 3.5: Die Effizienz der Maßnahmen, die den Datentransfer zwischen Storage Element und Worker Node verbessern, ergibt sich als Verhältnis der Zahl der Jobs, die wiederhergestellt werden konnten, zur Zahl der Jobs, die gescheitert wären. Ersteres ist die Zahl der Jobs, die gescheitert wären, abzüglich der Zahl der Jobs, die gescheitert sind.

3.2.3 Replizieren von Dateien im Grid

Die dritte Datentransferfunktion ist die *replicate*-Funktion. Soll eine Datei von einem Storage Element auf ein anderes repliziert werden, hat man mit den gleichen Problemen wie bei der *put*-Funktion und der *get*-Funktion zu kämpfen – dementsprechend sind die gleichen Lösungsansätze implementiert. Ob ein Transfer erfolgreich war oder nicht wird danach bestimmt, ob die neue SURL im File Catalog auftaucht. In diesem Projekt wird die *replicate*-Funktion hauptsächlich im DST-Receiver (Kapitel 4.7) verwendet, um die Outputdateien der Jobs ans

DESY zu replizieren. Dabei müssen die Replikate unter ganz bestimmten SURLs gespeichert werden (vergleiche Kapitel 2.1), deshalb sind diese schon vor Beginn des Transfers bekannt (im Normalfall reicht zum Replizieren die Angabe des Ziel-SEs). Also ist es auch möglich, einem gescheiterten Transfer mit effizienteren Methoden „hinterherzulöschen“. Denn bei bekannter SURL kann ein eventuell schon angelegtes Dateifragment auf dem Storage Element entfernt werden, auch wenn es nicht im File Catalog vorhanden ist. Dafür werden *low level grid tools* benutzt. In der *put*-Funktion ist dagegen die SURL nicht im Vorhinein bekannt.

3.3 Das Ausführen von H1SIMREC

Der Umgang mit dem Programm H1SIMREC gestaltet sich unerwartet schwierig. Zu kleineren Hürden, wie zum Beispiel fehlende oder sinnlose Fehlermeldungen zu kritischen Fehlern, kommen unangekündigte Abstürze und Endlosschleifen. Da es prinzipiell nicht möglich ist, sich auf einem Worker Node einzuloggen, ist es auch nicht möglich, auf eventuelle Fehler von H1SIMREC manuell zu reagieren. So muss der Job-Wrapper intelligent genug sein, um mit den verschiedenen Fällen umzugehen. Im Wesentlichen treten vier Probleme auf.

1. H1SIMREC *schreibt große Mengen sinnloser Zeichenketten (z.B. „0“) nach StdErr*[13]. Dies tritt immer dann auf, wenn etwas mit den Eingangsdateien (MONTECARLO-Eingangsdatei, Datenbankdatei, Noise Files) nicht in Ordnung ist, zum Beispiel falls die MONTECARLO-Eingangsdatei nicht gefunden wird, oder falls die Logdatei von H1SIMREC schon existiert. Solche Probleme treten nur aufgrund fehlerhafter Konfiguration auf, also im Falle eines Benutzerfehlers. Die Folge ist, dass die StdErr-Datei (vergleiche Kapitel 1.4.3) mit einigen 100 MB/s geschrieben wird. Die jetzige LCG-Version kann mit solchen Fällen nicht umgehen, so dass die Folgen von mehreren Gigabyte großen Output Sandboxes bis zum Absturz des Worker Node reichen.

2. H1SIMREC *trifft auf Endlosschleifen.* In diesem Fall werden über Stunden keine Ereignisse mehr geschrieben, obwohl der Prozess weiter CPU-Leistung verbraucht. Tut man nichts dagegen, wird der Job schließlich beendet, wenn sein Proxy ausläuft, oder das Zeitlimit der *queue* des Computing Elements erreicht wird.

[13] kurz für *standard error output*. Analog zu StdOut, allerdings für Fehlerausgaben der Programme. Daten, die nach StdErr geschrieben werden, erscheinen auf der interaktiven Konsole eines Unix-Systems.

3. H1SIMREC *prozessiert nicht alle Ereignisse.* Sehr selten tritt der Fall ein, dass sich H1SIMREC scheinbar korrekt beendet (zum Beispiel sehen die letzten Zeilen der Logdateien wie im Erfolgsfall aus), aber dennoch zu wenig Ereignisse prozessiert wurden. Die Ursache hierfür ist noch nicht verstanden.

4. H1SIMREC *stürzt ab.* Gerade „frische“ HERA-II-Versionen von H1SIMREC sind sehr instabil.

Die Erfahrungen aus der klassischen Produktion zeigen, dass die letzte Fehlerkategorie am häufigsten auftritt. Oft kommt es vor, dass H1SIMREC nach einer ungefähren Zahl prozessierter Ereignisse abstürzt. Bei Ereignissen, in denen viele Sekundärteilchen auftreten, ist 7.000 ein guter Wert für diese Grenze [20] – also noch deutlich unter den angestrebten 10.000 Ereignissen pro Grid-Job. Das zweithäufigste Problem, obgleich deutlich seltener als ersteres, ist das Auftreten einer Endlosschleife (Fehlerfall 2). Zum Umgang mit diesen Fehlern sind zwei Konzepte implementiert: Ein Überwachungsprozess für H1SIMREC (der so genannte *Wachhund*) und die H1SIMREC-*Recovery*. Auf beide wird im Folgenden eingegangen.

3.3.1 Überwachungsprozess für H1SIMREC: Der Wachhund

Die Aufgabe des Wachhunds ist es, bei grobem Fehlverhalten von H1SIMREC den Prozess zu beenden. Um das Fehlverhalten zu erkennen, kontrolliert er im Sekundenrhythmus die Dateigröße von verschiedenen kritischen Dateien[14]. Falls die Summe der Dateigrößen eine gewisse Zeit unverändert bleibt, wird H1SIMREC beendet (Fehlerfall 2). Ein guter Wert für diese Gnadenfrist ist eine Stunde, Werte darunter haben sich als zu kurz erwiesen. Falls andererseits der Umfang anderer Dateien[15] zu groß wird (Fehlerfall 1), wird H1SIMREC ebenfalls beendet. Technisch gesehen ist das Beenden mittels `kill -9` anspruchsvoller als erwartet, da man unter anderem berücksichtigen muss, dass durchaus zwei Jobs (also auch zwei Instanzen von H1SIMREC) auf dem gleichen Worker Node laufen können. Zusätzlich enthält PERL einige Fallen, was das Starten von externen Programmen angeht, so wird zum Beispiel in bestimmten Fällen ein Shell-Interpreter (`sh -c`) dazwischengeschaltet. Man muss dann sicherstellen, dass nicht lediglich der Shell-Interpreter vom Wachhund beendet wird, H1SIMREC aber weiter läuft.

[14] h1mc.log, h1simrec.out, std.err und die DST-Outputdateien
[15] h1simrec.out und std.err

3.3.2 H1SIMREC-Recovery

Die H1SIMREC-Recovery behandelt die Fehlerfälle 3 und 4, in denen die Zahl der prozessierten Ereignisse kleiner ist als der erwartete Wert. Die Erfahrungen [21] haben gezeigt, dass H1SIMREC in so einem Fall meistens erfolgreich durchläuft, wenn man es einfach eine gewisse Zahl von Ereignissen vor dem fehlerhaften neustartet. Der Grund für dieses Verhalten wird in Fehlern in der internen Speicherverwaltung der Detektorsimulation vermutet. Genau dieses Neustarten von H1SIMREC mit der dadurch notwendigen Behandlung der Outputdateien leistet die H1SIMREC-Recovery völlig autonom auf dem Worker Node. Wie sie im Detail funktioniert, zeigt Abbildung 3.6: Nachdem H1SIMREC beendet ist (sei es von selbst oder durch den Wachhund), wird zunächst die Zahl der prozessierten Ereignisse ermittelt. Ist sie zu klein, wird H1SIMREC d Ereignisse vor dem zuletzt berechnetem neugestartet (bewährt hat sich $d = 2$, [21]). Falls H1SIMREC nun in einem späteren Lauf zu einem erfolgreichen Ende gelangt, liegt zu jedem Lauf eine DST-Outputdatei vor, die aber unter Umständen am Ende korrupte Ereignisse enthält. Diese werden von den Outputdateien abgeschnitten. Schließlich werden die entstandenen Dateifragmente zu einer einzigen DST-Datei zusammengefügt, die dann den gewohnten Gang zum Storage Element geht. Bleiben dennoch alle vorgesehenen Wiederholungen erfolglos, erhält der Job den Wrapper-Exit-State `DST_ERROR`, worauf das Grid-Batch-System um manuelle Interaktion bittet. Für jeden Lauf von H1SIMREC werden leicht veränderte Konfigurationsdateien benutzt und zusätzliche Logdateien erzeugt. Zum Ende des Jobs werden all diese Dateien in einem Tarball zusammengefasst, der mit der Output Sandbox zurückgeschickt wird.

Beispielhaft sei hier ein Request erwähnt, der von der H1SIMREC-Recovery regen Gebrauch machen musste. Er bestand aus 100 Jobs, die jeweils 10.000 Ereignisse (Charm-Meson-Produktion in tiefinelastischer Streuung, HERA-II) prozessieren sollten. Bei 95 Jobs musste H1SIMREC durch die Recovery wiederhergestellt werden.

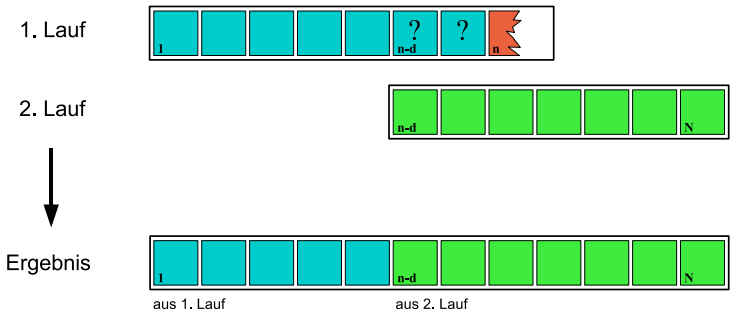

Abbildung 3.6: Ablauf der H1SIMREC-Recovery. Im Falle eines Absturzes wird zwei Ereignisse vor dem fehlerhaften neu gestartet. Anschließend werden die DST-Outputdateien der verschiedenen Läufe kombiniert.

Kapitel 4

Das H1-MONTECARLO-Grid-Batch-System

Im Kapitel 3 wurde der erste Teil des Produktionssystems vorgestellt, der Job Wrapper. Das ist der Teil, der es ermöglicht, dass H1SIMREC autonom und reibungslos auf einem Worker Node ausgeführt werden kann. Die zweite Herausforderung ist es nun, eine große Zahl von Jobs zu betreuen, und zwar mit einem Minimum an manueller Interaktion. Es wird also ein System benötigt, das selbstständig Jobs abschickt, regelmäßig ihren Status abfragt, für beendete Jobs die Output Sandbox abholt, diese auswertet und abhängig davon weitere Schritte einleitet. Bei alledem muss es möglichst intelligent auf Fehler reagieren. Dieses System ist das Grid-Batch-System, es wird im Folgenden vorgestellt.

4.1 Funktionsprinzip

Das Grundprinzip des Grid-Batch-Systems ist es, die verschiedenen Aufgaben von einer Reihe von gekapselten Modulen erledigen zu lassen. Deren einzige Schnittstelle ist eine zentrale Datenbank, die GBS-Datenbank. Jeder Job befindet sich in einem bestimmten Status, der in der Datenbank gespeichert ist, und jedes Modul wirkt nur auf Jobs, die einen zum Modul passenden Status haben. Nachdem ein Modul seine Arbeit an einem Job verrichtet hat, ändert es den Status dieses Jobs in der Datenbank. Nun kann der Job von einem anderen Modul weiter verarbeitet werden. Der große Vorteil einer solchen Architektur ist einerseits, dass jedes Modul zu einem beliebigen Zeitpunkt aufgerufen werden kann – sogar unabhängig davon, welche anderen Module im gleichen Moment arbeiten. Andererseits ermöglicht es die Modulstruktur, Teile des Projekts in völlig anderen Programmiersprachen zu implementieren. Zum Beispiel wurde das *Job-Maker*-Modul [22] in PYTHON geschrieben, wodurch der Autor seine Arbeitskraft direkt

beisteuern konnte ohne sich zunächst eine neue Programmiersprache aneignen zu müssen. Das Grid-Batch-System besteht aus sechs Modulen, die in Tabelle 4.1 und in Abbildung 4.1 kurz vorgestellt werden. In den folgenden Kapiteln werden sie detaillierter diskutiert.

Modul	Aufgabe
Job-Maker	bereitet die Jobs vor
Job-Submitter	schickt Jobs in das Grid
Job-Update-Modul	fragt den LCG *Status* der Jobs ab
OSB-Receiver	ruft die Output Sandbox eines Jobs ab
OSB-Checker	überprüft die Output Sandbox und entscheidet über das weitere Vorgehen
DST-Receiver	transferiert die DST-Outputdateien eines Jobs zum Storage Element `srm-dcache.desy.de`

Tabelle 4.1: Die Module des Grid-Batch-Systems

Wie bereits beschrieben, besteht ein MONTECARLO-Request aus vielen Jobs. Jeder von ihnen wird ins Grid geschickt, dort berechnet, und kehrt schließlich zurück. Allerdings kommt es oft vor, dass der Job erfolglos war, so dass er nochmals abgeschickt werden muss. Dennoch sollten die Daten, die beim ersten Versuch entstanden sind, erhalten bleiben, denn sie sind zum Beispiel nützlich zur Fehleranalyse. Das Grid-Batch-System genügt dieser Anforderung, indem es einen Request durch eine Hierarchie von Objekten abbildet, wie Abbildung 4.2 zeigt. Ganz oben steht das Request-Objekt, es enthält alle relevanten Informationen über den Request. In erster Linie ist das die eindeutige *Request-Nummer*, die in der Regel mit derjenigen übereinstimmt, die der Request in der H1-MONTECARLO-Datenbank erhalten hat.

Das Request-Objekt besitzt viele H1MC-*Job*-Objekte, denen mehrere LCG-*Job*-Objekte zugeordnet sind. Ein H1MC-Job enthält alle Daten, die erhalten bleiben, falls der Job erneut abgeschickt wird. Welche das konkret sind, zeigt Tabelle 4.2. Ein LCG-Job hingegen enthält alles, was sich beim nochmaligen Abschicken neu ergibt (zum Beispiel die LCG-Job-Id). Zusätzlich speichert ein LCG-Job Daten eher informativen Charakters, die zum Erstellen von Statistiken gebraucht werden. Beispiele hierfür sind die Laufzeit von H1SIMREC oder die durchschnittliche Datentransferrate. Wenn ein H1MC-Job erneut abgeschickt wird, bekommt er einfach einen weiteren LCG-Job zugeteilt.

Sowohl H1MC-Job als auch LCG-Job haben einen eigenen Status, die folgerichtig H1MC *Status* und LCG *Status* genannt werden. Der H1MC-Status ist in der zen-

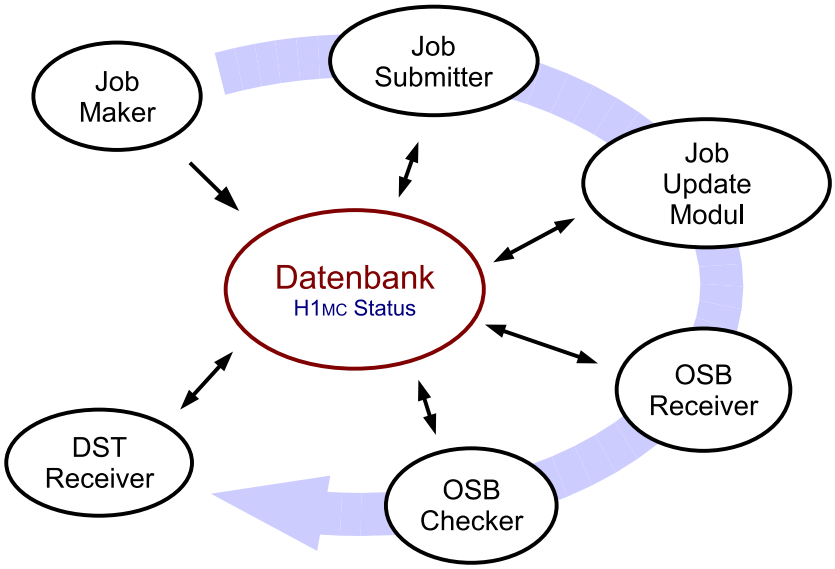

Abbildung 4.1: Die verschiedenen Module des Grid Batch Systems kommunizieren miteinander ausschließlich über die zentrale Datenbank. Ein erfolgreicher Job durchläuft die Module im Uhrzeigersinn, angefangen beim Job-Maker.

tralen Datenbank gespeichert und ermöglicht die Kommunikation zwischen den Modulen. Der LCG-Status ist direkt der Grid-Status, wie er im Kapitel 1.4.3 über die Grid-Jobs eingeführt wurde. Alle möglichen Zustände sind im Anhang B.1 und B.2 zusammengefasst. Besondere Bedeutung hat der jeweils aktuellste LCG-Job eines H1MC-Jobs. Aus seinem LCG-Status wird der H1MC-Status seines Elternjobs berechnet (vergleiche Kapitel 4.4).

4.2 Job-Maker

Der Job-Maker [22] erzeugt sämtliche H1MC-Jobs, die zu einem Request gehören. Er wird einmalig ganz zu Anfang der Produktion ausgeführt. Dazu legt der Job-Maker die Dateien an, die zu den H1MC-Jobs gehören (siehe Tabelle 4.2). Danach werden die neuen H1MC-Jobs in der Datenbank gespeichert. Der Benutzer des Grid-Batch-Systems muss dabei nur eine einzige Konfigurationsdatei manuell anpassen, nämlich die für den Job-Maker. Zum Beispiel werden dort die nötigen Informationen aus der offiziellen H1-MONTECARLO-Datenbank eingetragen. Die im Folgenden beschriebenen Punkte können dann automatisch erledigt werden. Daraufhin ist der Request bereit abgeschickt zu werden.

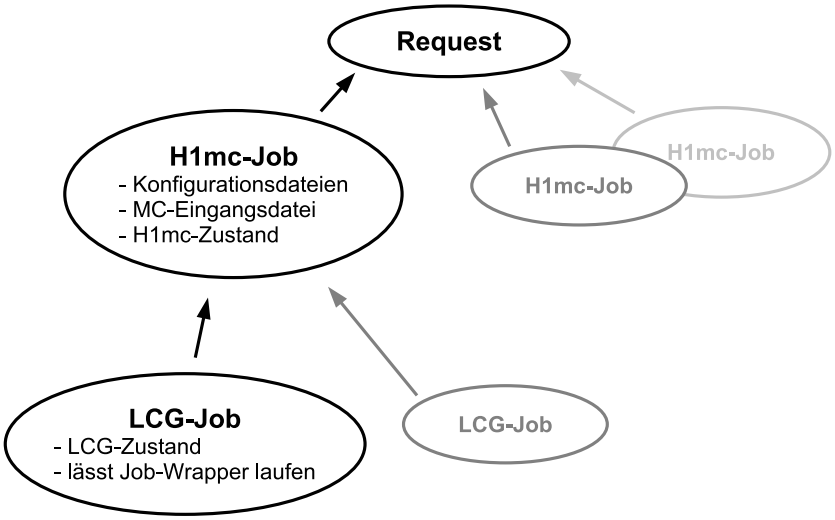

Abbildung 4.2: Ein MONTECARLO-Request wird im Grid Batch System durch drei Typen von Objekten dargestellt, die zueinander in Hierarchie stehen.

Datei	Funktion
MONTECARLO-Eingangsdatei	Enthält die Ereignisse, wie sie vom Ereignisgenerator erzeugt wurden.
`wrapper.conf`	Konfigurationsdatei für den Job-Wrapper
steeringcard	Konfigurationsdatei für H1SIMREC.
JDL-Datei	Beschreibung des LCG-Jobs

Tabelle 4.2: Dateien, die einem H1MC-Job zugeordnet sind.

MONTECARLO-Eingangsdateien vorbereiten

Die Dateien, die von den Analysegruppen mit ihren Ereignisgeneratoren erzeugt werden, enthalten meistens zu viele Ereignisse, um sinnvoll von einem einzigen LCG-Job prozessiert zu werden. Es würde viel zu lange dauern (vergleiche Kapitel 2). Deshalb werden die Dateien zunächst in kleinere zerteilt, die MONTECARLO-*Eingangsdateien*. Dabei hat sich eine Anzahl von 10.000 Ereignissen pro Datei bewährt. Als nächstes werden die zerschnittenen Dateien komprimiert. Je nach ihrem Datenvolumen stehen nun zwei Möglichkeiten zur Auswahl, wie die Eingangsdateien zum Worker Node gelangen: Entweder werden sie vom Job-Maker auf ein Storage Element transferiert (bei großen Dateien oder falls man sie nicht lokal vorhalten will). Oder die Dateien sind klein genug, um mit der Input Sand-

box geschickt zu werden. In erstem Fall berechnet der Job-Maker auch gleich die MD5-Prüfsumme und speichert sie im Katalog (vergleiche Kapitel 3.2.2).

Konfigurationsdateien anlegen

Um die Konfigurationsdateien anzulegen, benutzt der Job-Maker Vorlagen. In diese fügt er an verschiedenen Stellen Daten ein, die sich von Job zu Job ändern – zum Beispiel den Dateinamen der Eingangsdatei. Auf diese Art wird zunächst die JDL-Datei geschrieben, die den Grid-Job definiert. Dann werden sowohl die Konfigurationsdatei für den Job Wrapper als auch die Konfigurationsdatei für H1SIMREC erzeugt.

Datensätze einfügen

Zum Schluss werden für den neuen Request und für seine H1MC-Jobs Datensätze in der Datenbank angelegt. Die Jobs erhalten dabei den Status H1MC *new*.

4.3 Job-Submitter

Der Job-Submitter ist dafür zuständig, die H1MC-Jobs abzuschicken. Dafür durchsucht er die Datenbank nach neuen Jobs, die den Status H1MC *new* haben, und nach Jobs, die resubmittiert werden sollen. Letztere haben den Status H1MC *failed*, und werden genauso wie neue Jobs behandelt. Die zwei Schritte, die notwendig sind um einen Job abzuschicken, sind:

Zunächst muss ein neuer LCG-Job erzeugt und abgeschickt werden. Dafür wird zuerst ein neuer Datensatz in der Datenbank angelegt, der den nötigen Platz bietet für alle wichtigen Daten, die ein Grid-Job erzeugt. Dann wird der LCG-Job mit Hilfe eines Middleware-Kommandos[1] abgeschickt. Die zurückgelieferte LCG-Job-Id wird im zuvor vom Job-Maker angelegten Datensatz gespeichert.

Schließlich wird der Status auf H1MC *running* gesetzt.

4.4 Job-Update-Modul

Das Job-Update-Modul fragt den LCG-Status eines Jobs beim Resource Broker ab und errechnet daraus seinen H1MC-Status. Der Middlewarebefehl, der die eigentliche Arbeit verrichtet[2], liefert zusätzlich zum LCG-Status auch die Zeitpunkte,

[1] `edg-job-submit`
[2] `edg-get-status`

zu denen ein Job den jeweiligen Status erreicht hat. Diese Daten werden in die GBS-Datenbank gefüllt. Außerdem stellt der Befehl den Namen der *queue* zur Verfügung, der ebenfalls in der Datenbank gespeichert wird. Der H1MC-Status wird dabei mit der folgenden einfachen Methode ermittelt: Sobald der Job einen LCG-Status erreicht, der sich „von allein“ nicht mehr ändert, wird der H1MC-Status entweder auf H1MC *failed* oder auf H1MC *done* gesetzt, je nach dem ob ein Gridfehler auftrat oder nicht. „Von allein“ meint dabei die Zustände LCG *done*, LCG *aborted* und LCG *cancelled*, vergleiche auch Abbildung B.1 im Anhang.

4.5 OSB-Receiver

Bevor ein LCG-Job endet, speichert er die Output Sandbox auf dem Resource Broker. Diese muss nun vom Gridbenutzer auf das User Interface kopiert werden, was in unserem Fall der OSB-Receiver erledigt. Er bearbeitet nur solche H1MC-Jobs, die im Status H1MC *done* sind und ändert ihn nach erfolgreicher Arbeit auf H1MC *received*. Allerdings ändert er den Status nicht selbst, sondern ruft das Job-Update-Modul auf. Dieses stellt dann fest, dass sich der LCG-Zustand des Jobs von LCG *done* auf LCG *received* geändert hat, und passt den H1MC-Zustand an. Nachdem die Output Sandbox auf dem User Interface angekommen ist, beginnt der OSB-Receiver mit der Aufbereitung ihres Inhalts. Dazu entpackt er zum Beispiel die Logdateien aus ihrem Tarball (vergleiche Kapitel 3.3.2). Außerdem speichert er den genauen Pfad zu den Dateien in der zentralen Datenbank.

4.6 OSB-Checker

Der OSB-Checker bedient ausschließlich Jobs, die sich im Status H1MC *received* befinden. Seine Aufgabe ist es zu entscheiden, ob der Job erfolgreich gelaufen ist oder erneut abgeschickt werden muss. In diesem Modul steckt die Intelligenz des Grid-Batch-Systems. Als Entscheidungsgrundlage benutzt der OSB-Checker den Wrapper-Exit-State, den er aus der Logdatei des Wrappers herausliest. Es ist aber genauso möglich, weitere Daten des LCG-Jobs mit in die Entscheidung einzubeziehen. Konkret setzt der OSB-Checker den Status eines Jobs entweder auf H1MC *succeeded*, falls alles in Ordnung ist, oder auf H1MC *failed*. In zweitem Fall wird der Job vom Job-Submitter später erneut abgeschickt. Falls schwerwiegende Fehler auftraten, die nicht automatisch behoben werden können, wird der Status H1MC *broken* gesetzt. Eine detaillierte Beschreibung der möglichen Wrapper-Exit-States und der sich daraus ergebenden Aktionen findet sich im Anhang B.3. Damit ein Job nicht endlos resubmittiert wird, gibt es eine maximale

Anzahl an LCG-Jobs, die ein H1MC-Job besitzen darf. Wird sie erreicht, ruft der OSB-Checker um Hilfe, indem er den Status auf H1MC *broken* setzt.

Die zweite Aufgabe des OSB-Checkers ist es, statistisch relevante Daten aus den Logdateien des LCG-Jobs herauszufiltern und sie in der GBS-Datenbank zu speichern. Diese sind dann die Grundlage für detaillierte Statistiken über die Produktion. Hierzu wird die Statistikdatei ausgewertet, die vom Wrapper geschrieben wird (vergleiche Kapitel 3.1).

4.7 DST-Receiver

Nachdem ein Job vom OSB-Checker für gut befunden wurde, ist es die Aufgabe des DST-Receivers, die DST-Outputdateien an eine bestimmte Stelle des Storage Elements `srm-dcache.desy.de` zu kopieren (vergleiche Kapitel 2). Er arbeitet mit Jobs im Zustand H1MC *succeeded* und ändert den Status auf H1MC *finished*, wenn er erfolgreich war. Zuerst werden Dateinamen bestimmt, unter denen die Dateien gespeichert werden sollen. Die Dateinamen müssen einem bestimmten Benennungsschema entsprechen, um von den Analysegruppen verwendet werden zu können. Nun enthält dieses Schema einen fortlaufenden Index über alle Dateien eines Requests. Weiterhin erzeugt ein Job potenziell mehrere DST-Outputdateien, so dass nicht einfach die Jobnummer als Index herhalten kann. Daher muss mit *low level grid tools* auf dem Storage Element geprüft werden, welche Dateinamen schon vergeben sind, und anschließend ein freier gewählt werden. Dieses Vorgehen widerspricht der Grid-Philosophie – was aber verzeihlich ist, da es hier um die Schnittstelle zwischen klassischer Produktion und Gridtechnik geht. Wurde ein Dateiname gewählt, benutzt der DST-Receiver die *replicate*-Funktion, die im Kapitel 3.2 über Datentransfer beschrieben ist. Damit repliziert er die Dateien, die der LCG-Job auf einem beliebigen Storage Element abgelegt hat, zum `srm-dcache.desy.de`.

In einigen Fällen hat der LCG-Job den Rescue Tarball (siehe Kapitel 3.2.2) benutzt, und dieser liegt auf dem User Interface. Dann entpackt der DST-Receiver den Tarball und benutzt die *put*-Funktion, um die entpackten Dateien auf das `srm-dcache.desy.de` hochzuladen. Da es hier um etwa 400 Megabyte geht, ist das eine große Belastung für das User Interface, falls viele Jobs gleichzeitig bearbeitet werden müssen. Es hat sich als sinnvoll erwiesen, vor dem Entpacken des Tarballs auf genügend freien Speicherplatz zu prüfen. Dazu wird die Größe des Tarballs und das bekannte Kompressionsverhältnis, das für DST-Dateien erreicht wird, benutzt.

Zum Abschluss werden die alten Dateien gelöscht – entweder die Replikate auf dem entfernten Storage Element, oder die Rescue Tarballs und die ausgepackten Dateien.

4.8 Der Daemon

Um das Grid-Batch-System in Gang zu setzen, müssen die in den vorigen Kapiteln vorgestellten Module ausgeführt werden. Dabei bearbeitet jedes Modul eine gewisse Zahl von H1MC-Jobs, die einen passenden Zustand haben – aber nur sehr selten alle Jobs eines Requests. Um das Grid-Batch-System in Gang zu halten, müssen die Module also wiederholt aufgerufen werden und zwar solange, bis alle H1MC-Jobs beendet sind (H1MC *finished* oder H1MC *broken*). Genau das erledigt der Daemon[3] automatisch. Nachdem man einen Request mit dem Job-Maker erzeugt hat, übergibt man ihn in die Obhut des Daemons. Im Idealfall muss man nun nichts weiter unternehmen als zu warten, bis alle Jobs erfolgreich beendet sind.

In der Praxis lohnt es sich aber, die Arbeit des Daemons zu überwachen und gegebenenfalls auf veränderte Bedingungen zu reagieren. Man kann zum Beispiel im laufenden Betrieb Module ausschalten (sie werden dann nicht mehr ausgeführt), oder die Pausen zwischen zwei Aufrufen eines Moduls einstellen. Dadurch lässt sich die Leistung des Grid-Batch-Systems an verschiedene Situationen anpassen, zum Beispiel das schnelle Abschicken eines ganzen Requests, wie es in folgendem Kapitel 5.1 beschrieben wird. Abbildung 4.3 zeigt, welche Module zu welchem Zeitpunkt eingeschaltet waren, Abbildung 4.4 dokumentiert die erreichten Raten.

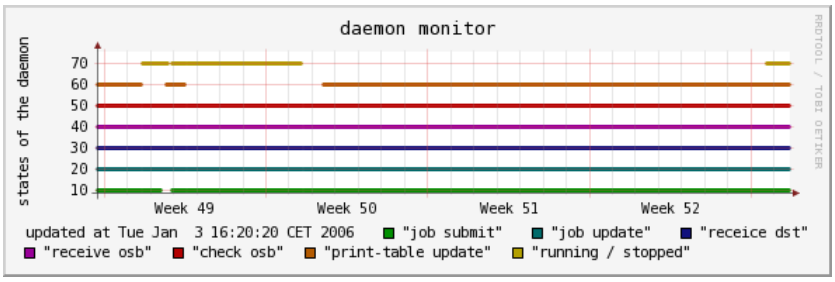

Abbildung 4.3: Monitorplot des Daemons. Die Linien zeigen an, wann das entsprechende Modul aktiv war. Das Modul „print-table update“ verbessert die Leistung des Web-Interface.

Der Daemon legt für jedes Modul eine eigene Logdatei an, in der auch die Fehler, die in dem jeweiligen Modul auftraten, protokolliert werden. Damit man nicht

[3] Mit dem englischen Wort *daemon* bezeichnet man in der Unixwelt einen Prozess, der im Hintergrund läuft, also ohne über eine Konsole mit einem Benutzer zu interagieren. Für gewöhnlich bieten Daemons Dienste an, wie zum Beispiel einen Datenbankserver oder einen Webserver.

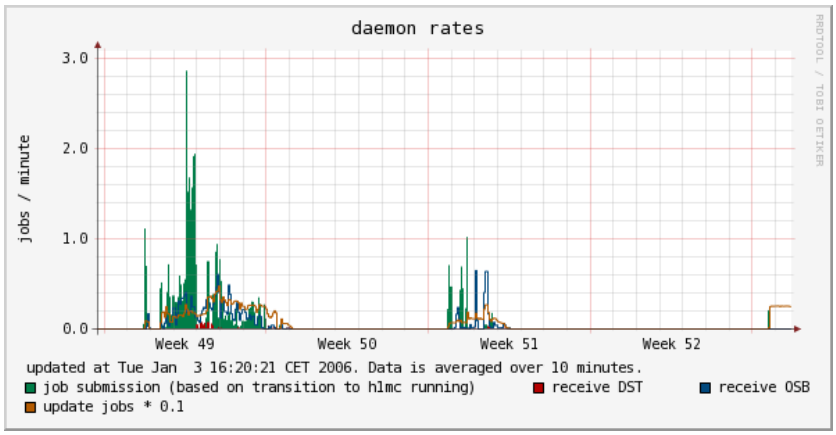

Abbildung 4.4: Ratenplot des Daemons. In den Wochen 50 und 52 fand Entwicklungsarbeit statt. Die Rate, mit der Jobs aktualisiert werden, ist zehnfach größer, als im Plot eingetragen. Zur besseren Übersicht wurde sie skaliert.

manuell alle Logdateien nach etwaigen Fehlern durchsehen muss, hält der Daemon zusätzlich eine Auswahl der zuletzt aufgetretenen Fehler vor. Somit ist der Daemon das Mittel der Wahl, um große Produktionen zu steuern und zu überwachen.

4.9 Web-Interface

Neben dem Daemon ist das in diesem Projekt entwickelte Web-Interface ein zweites mächtiges Werkzeug, um Produktionen zu steuern und zu überwachen. Bei kleinen Produktionen kann es den Daemon sogar vollständig ersetzen, denn das Web-Interface bietet die Möglichkeit, sämtliche Module[4] manuell aufzurufen. Das kann entweder für einen kompletten Request geschehen, oder nur für einzelne Jobs. Weiterhin lässt sich in die laufende Produktion eingreifen, indem der H1MC-Zustand eines Jobs manuell gesetzt wird. Daraufhin wird der Job von anderen Modulen weiterverarbeitet werden, als es ohne die Intervention der Fall gewesen wäre. So lassen sich Fehler auch dann beheben, wenn man die Module nicht selbst ausführen will oder kann[5].

Eine zweite Aufgabe des Web-Interfaces ist es, detaillierte Informationen zu allen Jobs und Requests bereitzustellen. Zunächst ist da die Zahl der Jobs in bestimmten Zuständen, die sofort Auskunft über den Stand eines Requests gibt. Weiterhin erzeugt das Web-Interface viele Histogramme, die die laufende Produktion betreffen und zum Beispiel zeigen, wie die verschiedenen Grid-Sites ausgelastet sind. Schließlich gibt es eine umfangreiche statistische Auswertung vieler Jobs und Requests. Sehr nützlich ist auch die Möglichkeit, problematische Jobs herauszufiltern und sich deren Logdateien direkt ansehen zu können. So lassen sich Fehler schnell finden und beheben.

[4] Alle bis auf den DST-Receiver, da er zu lange dauern würde, und den Job-Maker.

[5] Falls die Jobs zum Beispiel mit dem Proxy eines anderen Gridbenutzers abgeschickt wurden, kann man sie mit seinem eigenen Proxy nicht bearbeiten.

Abbildung 4.5: Die Startseite des Web-Interfaces. In der Tabelle sind die Requests aufgelistet sowie die Zahl der H1MC-Jobs in den verschiedenen H1MC-Zuständen. Request 3924 wird gerade prozessiert.

Job Nr	H1mc State	Grid State	Exit State	Site	LCG Jobs	H1mc State entered	actions
2	running	Running	not yet finished	CSCS	1	2006-03-01 12:45:49	
3	running	Running	not yet finished	CSCS	1	2006-03-01 12:46:00	
10	running	Running	not yet finished	CSCS	1	2006-03-01 12:47:07	
11	running	Scheduled	not yet finished	CSCS	1	2006-03-01 12:47:57	
18	running	Scheduled	not yet finished	CSCS	1	2006-03-01 12:49:03	
8	running	Running	not yet finished	Dortmund	1	2006-03-01 12:46:49	
12	running	Running	not yet finished	Dortmund	1	2006-03-01 12:48:08	
16	running	Running	not yet finished	Dortmund	1	2006-03-01 12:48:46	
21	running	Running	not yet finished	Dortmund	1	2006-03-01 12:50:41	
24	running	Running	not yet finished	Dortmund	1	2006-03-01 12:51:39	
26	running	Running	not yet finished	Birmingham 1	1	2006-03-01 12:51:58	
27	running	Running	not yet finished	Birmingham 1	1	2006-03-01 12:52:07	
4	running	Running	not yet finished	Prag	1	2006-03-01 12:46:10	
7	running	Running	not yet finished	Prag	1	2006-03-01 12:46:40	
34	running	Scheduled	not yet finished	Prag	1	2006-03-01 12:53:52	
43	running	Running	not yet finished	Prag	1	2006-03-01 12:56:28	
45	running	Running	not yet finished	Prag	1	2006-03-01 12:57:17	
46	running	Running	not yet finished	Prag	1	2006-03-01 12:57:26	
49	running	Scheduled	not yet finished	Prag	1	2006-03-01 12:57:54	
50	running	Scheduled	not yet finished	Prag	1	2006-03-01 12:58:05	
9	running	Running	not yet finished	Desy	1	2006-03-01 12:46:58	
28	running	Running	not yet finished	Desy	1	2006-03-01 12:52:17	
33	running	Running	not yet finished	Desy	1	2006-03-01 12:53:43	
37	running	Running	not yet finished	Desy	1	2006-03-01 12:54:19	
47	running	Running	not yet finished	Desy	1	2006-03-01 12:57:35	
29	running	Scheduled	not yet finished	RAL Tier 2	1	2006-03-01 12:52:26	
31	running	Scheduled	not yet finished	RAL Tier 2	1	2006-03-01 12:53:22	
32	running	Running	not yet finished	RAL Tier 2	1	2006-03-01 12:53:33	
38	running	Scheduled	not yet finished	RAL Tier 2	1	2006-03-01 12:54:28	
39	running	Scheduled	not yet finished	RAL Tier 2	1	2006-03-01 12:54:40	
40	running	Scheduled	not yet finished	RAL Tier 2	1	2006-03-01 12:54:54	
41	running	Scheduled	not yet finished	RAL Tier 2	1	2006-03-01 12:56:07	
42	running	Scheduled	not yet finished	RAL Tier 2	1	2006-03-01 12:56:19	
1	running	Running	not yet finished	RAL Tier 1	1	2006-03-01 12:45:02	
14	running	Running	not yet finished	RAL Tier 1	1	2006-03-01 12:48:27	
30	running	Running	not yet finished	RAL Tier 1	1	2006-03-01 12:52:35	

Abbildung 4.6: Eine Detailansicht von Request 3924. Die Tabelle zeigt seine H1MC-Jobs, deren Zustände und Grid-Site sowie den Zeitpunkt, an dem der H1MC-Zustand angenommen wurde.

4.10 Ein typischer Request

Um das in den vorangegangenen Kapiteln vorgestellte Grid-Batch-System gewissermaßen „in Aktion“ zu sehen, begleitet dieses Kapitel einen exemplarischen Request[6]. Anhand von drei Graphen, die vom Web-Interface zur Verfügung gestellt werden, lässt sich das Zusammenspiel der Module gut verfolgen. Zunächst zeigt Abbildung 4.7 die Rate ausgewählter Module. Im Vorfeld wurde der Job-Maker ausgeführt, so dass alle Jobs abschickbereit sind. Um 18:30 Uhr begann der Daemon damit, den Job-Submitter, das Job-Update-Modul, den OSB-Receiver und den DST-Receiver regelmäßig auszuführen. Folglich steigt die Rate, mit der Jobs abgeschickt werden, an.

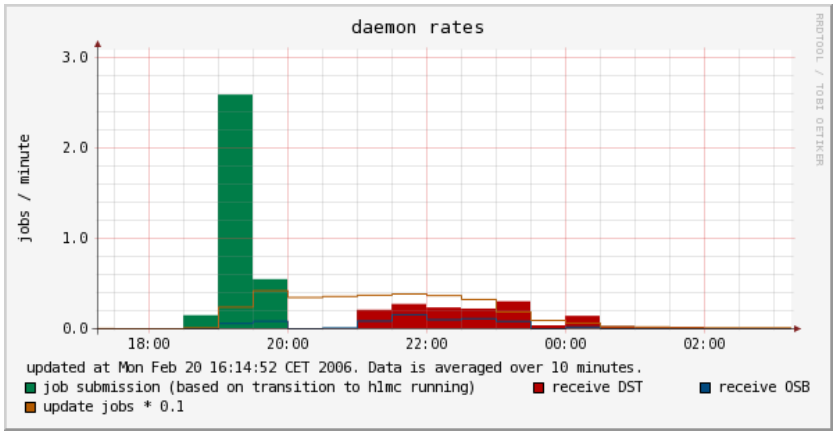

Abbildung 4.7: Raten der verschiedenen Module. Vergleiche auch Abbildung 4.4.

Der zweite Graph (Abbildung 4.8) zeigt die Anzahl der laufenden H1MC-Jobs (im Zustand H1MC *running*), und die Zahl der laufenden LCG-Jobs. Letztere ist zusätzlich aufgeschlüsselt in LCG *scheduled*, womit die meisten LCG-Jobs beginnen, und LCG *running*. Die Zahl der laufenden Jobs steigt umso schneller, je größer die Rate ist, mit der Jobs abgeschickt werden.

Nach einer gewissen Zeit sind die ersten Jobs beendet, weshalb die Zahl der laufenden Jobs gegen 21:30 Uhr zu sinken beginnt. Nun kann der OSB-Receiver die Output Sandboxes abholen, seine Rate steigt. Der dritte Graph (Abbildung 4.9) zeigt die Zahl der prozessierten Ereignisse. Sie wird erhöht, sobald der OSB-Checker einen erfolgreichen Job festgestellt hat. Danach beginnt der DST-Receiver, die DST-Dateien zurückzureplizieren, weshalb jetzt seine Rate ansteigt.

[6]Es wurde ein Request mit 100 Jobs ausgewählt, jeweils mit 20.000 wenig CPU-intensiven elastischen Ereignissen: $ep \rightarrow ep\omega \rightarrow ep\pi^+\pi^-\pi^0$

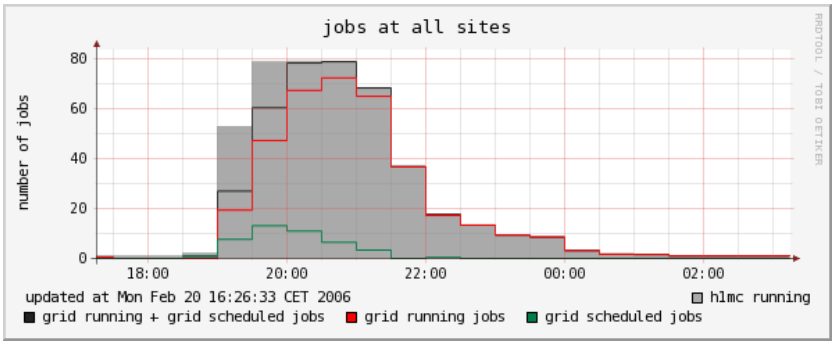

Abbildung 4.8: Laufende Jobs. Zwischen der Summe der LCG-Jobs und den H1MC-Jobs besteht zunächst eine Diskrepanz. Das rührt daher, dass noch nicht alle abgeschickten H1MC vom Job-Update-Modul aktualisiert werden konnten, ihr LCG-Zustand also noch unbekannt ist.

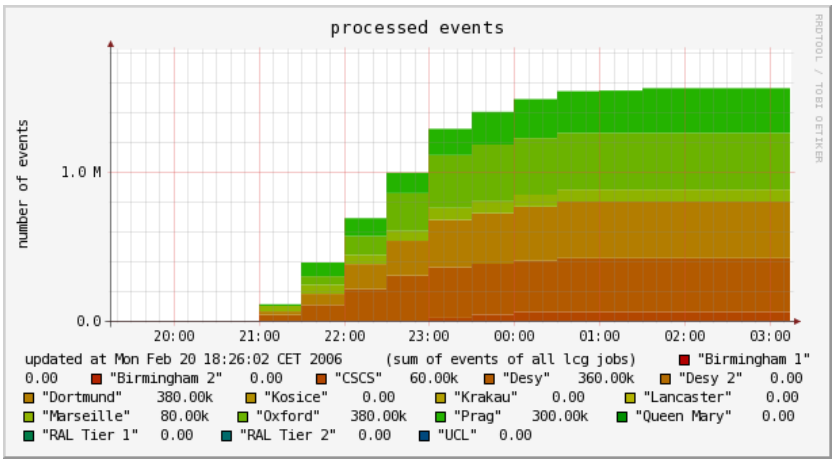

Abbildung 4.9: Prozessierte Ereignisse.

Kapitel 5

Optimierung des Grid Batch Systems

Nachdem das in Kapitel 4 vorgestellte Grundsystem aufgebaut worden war, konnte mit dem Testbetrieb begonnen werden. Dabei zeigte sich, dass noch viele unvorhergesehene Herausforderungen anzugehen waren. Über die vorgenommenen Verbesserungen der einzelnen Module wird in diesem Kapitel berichtet.

5.1 Optimierung des Job-Submitters

Der rudimentäre Job-Submitter arbeitet zufriedenstellend mit einer überschaubaren Zahl von Jobs. Es tauchen allerdings eine ganze Reihe von Problemen auf, wenn mehrere hundert Jobs abgeschickt werden müssen. Das offensichtlichste betrifft den Grid Proxy[1]. Bevor man einen Job ins Grid submittiert, muss man darauf achten, dass man einen ausreichend langen Proxy erzeugt. Ansonsten werden die abgeschickten Jobs zu früh von der Middleware abgebrochen. Gerade wenn Jobs automatisch abgeschickt werden, vergisst man leicht, dass zum Beispiel auch solche Jobs, die erst in der Nacht abgeschickt werden, einen ausreichend langen Proxy benötigen. Für Requests mit etwa 300 Jobs hat sich eine Laufzeit von drei Tagen bewährt. Eine mögliche Lösung ist die Nutzung eines *MyProxy Servers*. Auf diesem können besondere Langzeitzertifikate hinterlegt werden, mit deren Hilfe die Jobs auf dem Worker Node automatisch neue Proxies erzeugen können. So wäre es kein Problem mehr, wenn der vor dem Abschicken manuell erzeugte Proxy auslaufen würde, während der Job noch bearbeitet wird. Mit diesem Ansatz wurde zwar schon experimentiert, aber leider konnte noch keine zufriedenstellende Lösung gefunden werden.

[1] siehe Kapitel 1.4.1

Das nächste Problem ist, dass sich Informationen im Grid nur relativ langsam bewegen. So vergehen zwischen dem Zeitpunkt des Abschickens eines Jobs und dem Zeitpunkt, zu dem dieses Ereignis allen Gridsystemen bekannt ist, durchaus fünf Minuten. Schickt man in dieser Zeit mehr Jobs ab, als an eine bestimmte Grid-Site passen, so warten unnötig viele Jobs an dieser Site, während andere unausgelastet sind. Abhilfe bringen zwei Maßnahmen.

- Die Jobs in kleinen Paketen abschicken und in der Zwischenzeit warten. Das wird vom *Daemon* übernommen, beschrieben in Kapitel 4.8.
- Vor dem Submittieren für jeden Job prüfen, ob freie Grid-Sites vorhanden sind. Dieser Test wurde in den Job-Submitter eingebaut.

Es gibt ein weiteres problematisches Verhalten, das aber durch die vorangegangenen Lösungen ebenfalls stark verbessert wird. Werden nämlich Jobs abgeschickt, ohne dass freie Grid-Sites vorhanden sind, so gibt nicht etwa schon das Middleware-Kommando, das den Job abschickt, einen Fehler zurück. Vielmehr wird der Job dennoch submittiert, um kurz danach von der Middleware abgebrochen zu werden (Status LCG *aborted*).

Es hat sich als nützlich erwiesen, während der Produktion zeitnah beeinflussen zu können, an welche Grid-Sites die Jobs gesendet werden. So kann man zum Beispiel in der Konfigurationsdatei des Grid-Batch-Systems solche Sites ausschließen, die spontan ineffizient geworden sind. Ebenfalls kann es gewünscht sein, bestimmte Standorte zu bevorzugen, um sie zu testen. Zu diesem Zweck editiert der Job-Submitter die JDL-Datei unmittelbar bevor er auf freie Grid-Sites prüft. Dabei wird einerseits die `Requirements`-Sektion um eine Positivliste der erlaubten Sites ergänzt. Andererseits kann die `Rank`-Sektion bearbeitet werden. Beide Sektionen beeinflussen, welche von mehreren möglichen Sites den Zuschlag erhält (siehe auch Kapitel 1.4.3).

Die Rate, mit der Jobs abgeschickt werden können, hängt hauptsächlich von zwei Parametern ab. Der erste ist die Zeit, die gewartet wird, nachdem ein Job abgeschickt wurde. Der zweite Parameter ist die Systemlast, die durch das Job-Update-Modul erzeugt wird. Es hat sich bewährt, bei einem neuen Request zunächst das Job-Update-Modul sehr langsam laufen zu lassen, um hohe Submissionsraten zu erreichen. Später, wenn alles abgeschickt ist, kann die Aktualisierungsrate wieder erhöht werden. Diese Zusammenhänge verdeutlicht Abbildung 5.1. Gezeigt ist zweimal der gleiche Request, bestehend aus 100 Jobs. Oben waren der Job-Submitter und das Job-Update-Modul gleichzeitig aktiv, was zu einer hohen Systemlast und einer kleinen Abschickrate führt. Unten wurde das Job-Update-Modul erst eingeschaltet, nachdem alle Jobs abgeschickt waren.

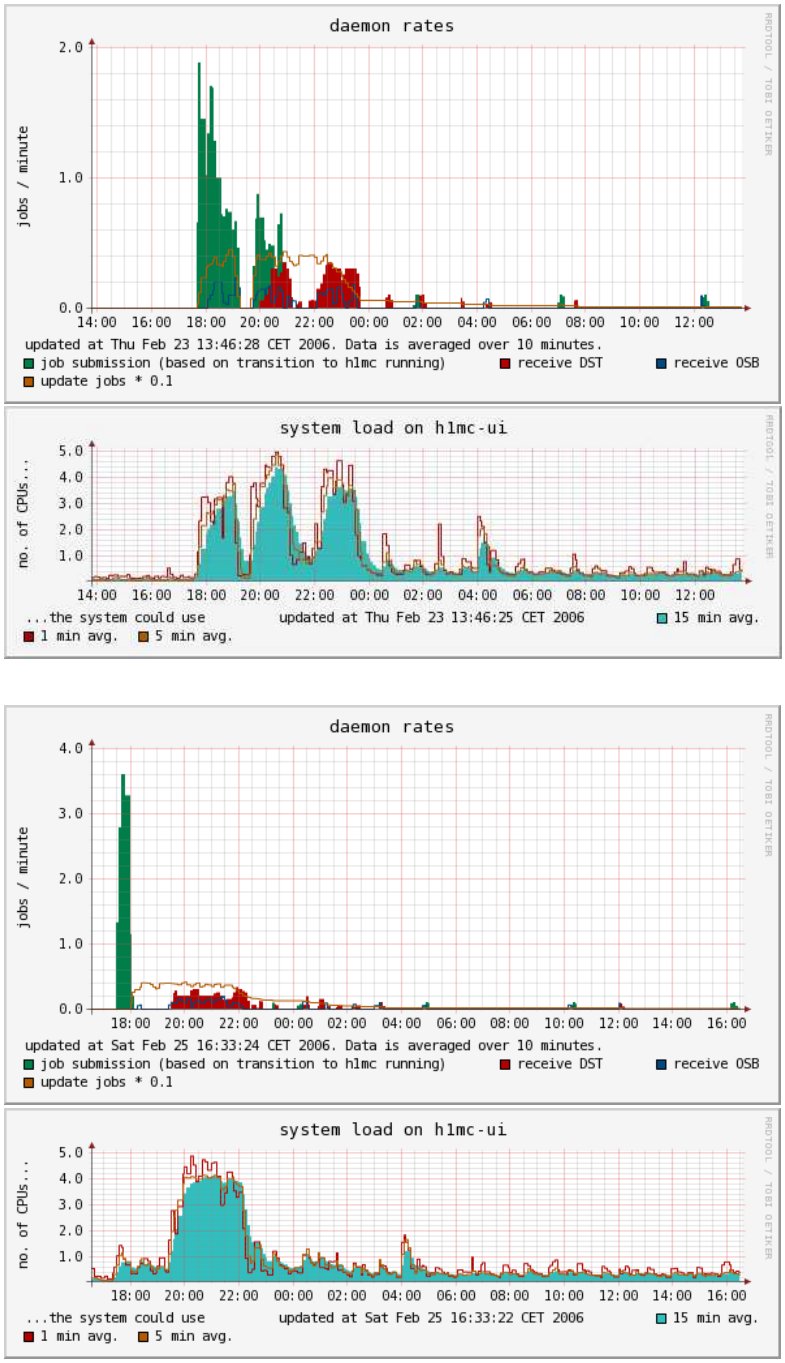

Abbildung 5.1: Zusammenhang zwischen der Rate, mit der Jobs abgeschickt werden, und der Systemlast des User Interface für verschiedene Situationen (siehe Text).

5.2 Optimierung des Job-Update-Moduls

Von Zeit zu Zeit treten Fehler im Batchsystem eines Computing Elements (Abbildung 1.9) auf, die das Computing Element selbst zu beheben versucht, indem es den Job erneut in eine seiner *queues* schickt. Dabei erreicht der Job jedoch kurzfristig den Zustand LCG *done*, bevor er wieder auf LCG *running* gesetzt wird. Nun kommt es vor, dass das Job-Update-Modul gerade in diesem Zeitfenster den Status eines Jobs abfragt. Fortan geht das Grid-Batch-System fälschlicherweise davon aus, dass der Job auch tatsächlich beendet ist. Um diese Falle zu umgehen, überprüft das Job-Update-Modul zusätzlich auch solche Jobs, die im Zustand H1MC *done* sind. Es setzt den Status eines Jobs auf H1MC *running* zurück, wenn es feststellt, dass der Job plötzlich wieder im Zustand LCG *running* ist. Das kann die Zahl der zu überwachenden Jobs beträchtlich erhöhen, da gegen Ende eines Request oft viele Jobs gleichzeitig fertig werden und in den Zustand H1MC *done* übergehen.

5.3 Optimierung des OSB-Receivers

Das Abrufen der Output Sandboxes bereitet erfreulicherweise wenig Probleme. Selbst in belastungsintensiven Situationen traten kaum Fehler im OSB-Receiver auf. Eine solche Situation liegt zum Beispiel vor, wenn viele Jobs vom Rescue Tarball (vergleiche Kapitel 3.2.2) Gebrauch machen. In diesem Fall werden mehrere hundert Megabyte pro Job empfangen, was sehr lange dauert. Außerdem füllen viele Rescue Tarballs schnell die Festplatte des User Interface. Da in so einem Fall der Daemon seine Logdateien nicht weiter schreiben könnte, würde er abstürzen und dabei eventuell Inkonsistenzen in der Datenbank erzeugen. Diese müssten dann mühsam per Hand gefunden und behoben werden. Um dem vorzubeugen, prüft der Daemon regelmäßig auf genügend freien Platz und schaltet sich im Notfall kontrolliert selbst ab.

5.4 Erfahrungen mit dem DST-Receiver

Es hat sich gezeigt, dass dieser Teil der Produktion sehr empfindlich ist. In erster Linie liegt das daran, dass es zum `srm-dcache.desy.de` keine Alternative gibt. Wenn es also ausfällt, kommt die Produktion ins Stocken, weil dann der DST-Output den Analysegruppen nicht zur Verfügung gestellt werden kann, obwohl er unter Umständen schon lange fertig ist. Die Zeit, die die Requests für diesen letzten Schritt benötigen, ist in Abbildung 5.2 gezeigt. Alle Werte über einer Stunde können auf ein zu langsames `srm-dcache.desy.de` zurückgeführt werden.

Im Laufe der Zeit ist es allerdings der DESY-IT-Abteilung gelungen, die Leistung entscheidend zu verbessern. Die im Rahmen dieser Arbeit durchgeführten Produktionen waren dafür durchaus hilfreich, da sie eine Menge Probleme aufzeigten, die sich erst im Großbetrieb ergeben. Abbildung 5.3 zeigt die Verbesserung eindrucksvoll. Ende August und Ende Oktober wurde von der DESY-IT-Gruppe eine neue Version des dem `srm-dcache.desy.de` zugrundeliegenden SRMs installiert, was sich deutlich bemerkbar machte. Aufgrund unterschiedlicher Bedingungen zu den dargestellten Zeiten sowie stark unterschiedlicher Zahl der Jobs in den jeweiligen Bins bieten sich weitergehende Schlüsse aber nicht an.

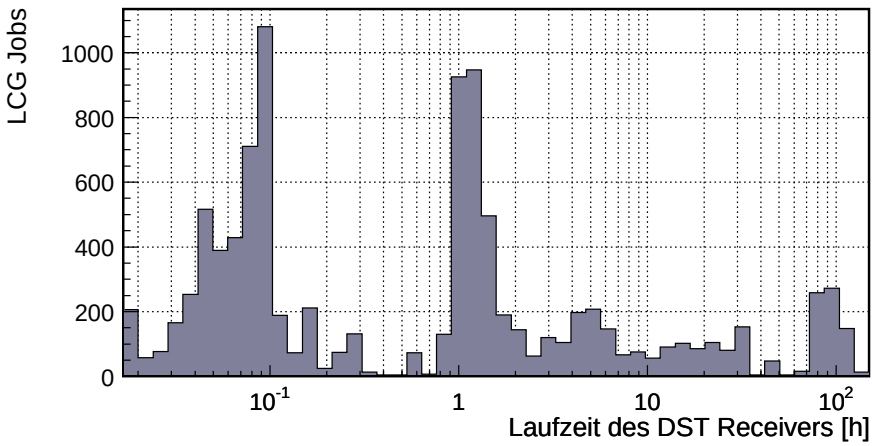

Abbildung 5.2: Die Zeit, die zum Zurückreplizieren des DST-Outputs benötigt wurde

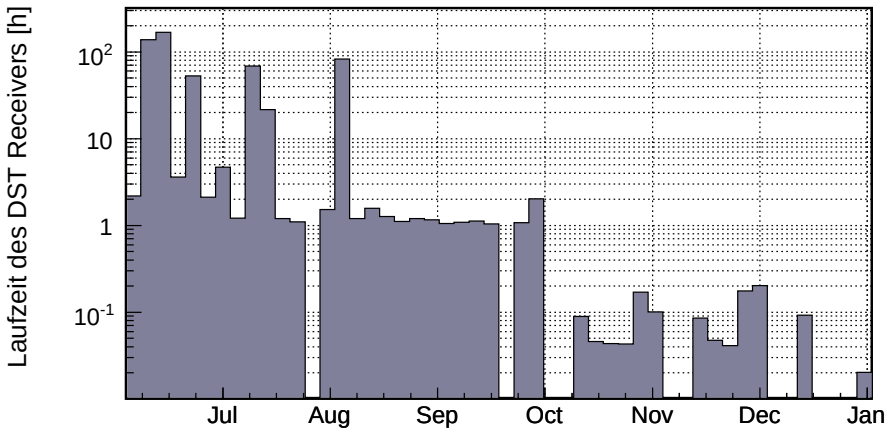

Abbildung 5.3: Die durchschnittliche Zeitspanne, die zum Zurückreplizieren der DST-Dateien benötigt wurde, aufgetragen gegen die Zeit.

Kapitel 6

Die Leistungsfähigkeit des Grid-Batch-Systems

Schon während der Entwicklungsphase des Grid-Batch-Systems wurden laufend Produktionen durchgeführt. Dabei wurde ab Juni 2005 eine ausreichende Qualität erreicht, so dass von dem Zeitpunkt an prozessierte Requests in diesem Kapitel ausgewertet werden können. Schon bald zeigten verschiedene Analysegruppen Interesse an den neuen Ressourcen. So wurden später „inoffizielle“ Produktionen im Grid durchgeführt, deren Ereignisse von den Analysegruppen verwendet werden konnten. Das trifft auf etwa die Hälfte aller produzierten Ereignisse zu.

Seit März 2006 ist die in dieser Arbeit entwickelte Produktionsumgebung Teil der offiziellen MONTECARLO-Produktionskette von H1 [27].

6.1 Leistung

Die wichtigste Kenngröße, die die Leistungsfähigkeit eines Produktionssystems für MONTECARLO-Ereignisse beschreibt, ist die Zahl der Ereignisse, die in einer bestimmten Zeit prozessiert werden können. Diese Zahl beträgt für das hier entwickelte Grid-Batch-System knapp 120 Millionen Ereignisse in sieben Monaten, was hochgerechnet auf ein Jahr gut 205 Millionen Ereignisse bedeutet. Das entspricht etwa $50\,\%$ der Leistung der klassischen Produktionskette, die im Jahr 2005 432 Millionen Ereignisse erreichte (vergleiche Abbildung 1.6). Die durch diese Arbeit für H1 neu erschlossenen Ressourcen sind somit vergleichbar mit der klassischen Farm des RAL, die im Jahr 2005 256 Millionen Ereignisse zur klassischen Gesamtproduktion beisteuerte. In Abbildung 6.1 ist die Zahl der im Grid produzierten Ereignisse gegen die Zeit aufgetragen.

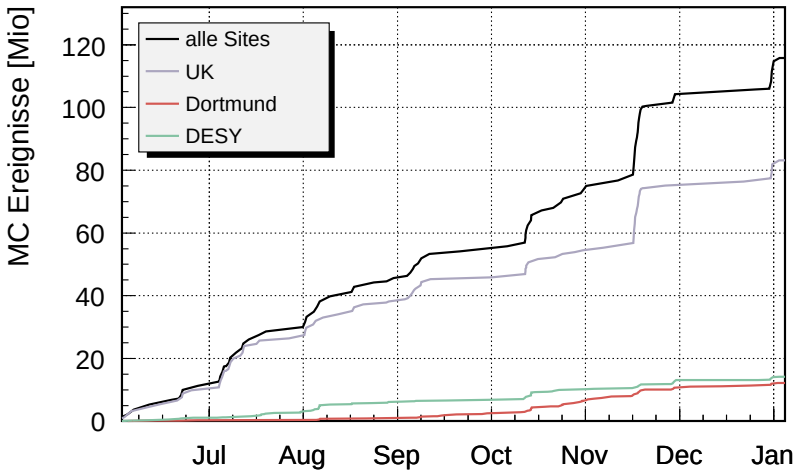

Abbildung 6.1: Gesamtzahl der mittels des Grid-Batch-Systems produzierten Ereignisse und der Beitrag von großen Grid-Sites. Der Eintrag für UK stellt die Summe aller Grid-Sites in Großbritannien dar.

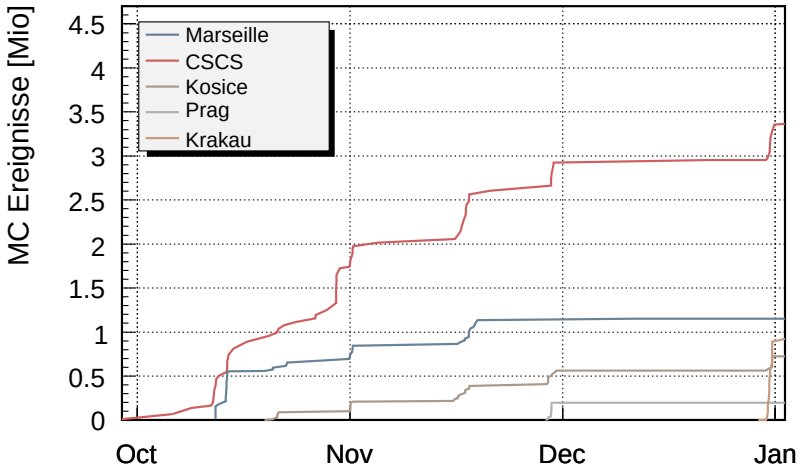

Abbildung 6.2: Gesamtzahl der mittels des Grid-Batch-Systems produzierten Ereignisse und der Beitrag von kleinen Grid-Sites

Es fällt zum einen auf, dass verhältnismäßig viele Plateaus vorhanden sind. Während dieser Zeiten wurde überwiegend Entwicklungsarbeit geleistet. Zum anderen sticht die im November erreichte Maximalleistung ins Auge. Sie beträgt etwa 20 Millionen Ereignisse pro Woche. Abbildung 6.2 zeigt den nicht unerheblichen Beitrag kleinerer Grid-Sites (vergleiche Tabelle 1.4.1), die ab Oktober 2005 das H1-Experiment unterstützen.

Abbildung 6.3 zeigt die insgesamt erreichte Produktionsrate, wobei reine Testproduktionen ausgelassen wurden. Außerdem wurden Requests ausgelassen, bei denen Probleme beim abschließenden Replizieren der DST-Dateien auftraten. Die verbleibenden Requests erreichten eine mittlere Rate von 32.000 Ereignissen pro Stunde. Dabei erreichen Requests mit wenigen Ereignissen (genauer mit wenigen Jobs) niedrigeren Raten als solche mit vielen Ereignissen. Letztere profitieren mehr von der Parallelisierung im Grid.

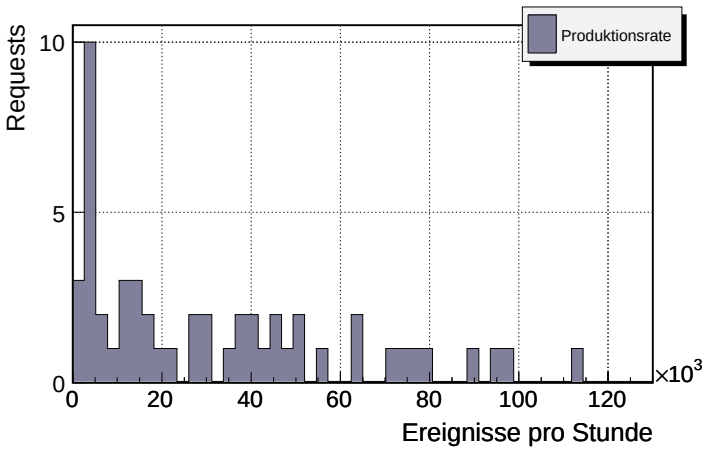

Abbildung 6.3: Die von den Requests erreichte Produktionsrate (52 Einträge).

Die Laufzeit eines Requests schwankt zwischen einem und zwei Tagen, wie Abbildung 6.4 zeigt. Durchschnittlich beträgt sie 34 Stunden. Dabei wurde nur die Zeit gemessen, die der Request tatsächlich „im Grid“ verbrachte, also das Intervall zwischen dem Zeitpunkt, an dem der erste LCG-Job auf LCG *running* ging und demjenigen, an dem der letzte H1MC-Job auf H1MC *finished* gesetzt wurde. Dadurch ist die Laufzeit nicht direkt mit der Laufzeit klassischer Requests zu vergleichen, die oft eine Woche brauchen – das allerdings inklusive sämtlicher Vorlaufzeiten. Bei einem Requests wurde die Laufzeit (119 h) allerdings von der Zeit bestimmt, die nötig war, die DST-Dateien zurück zu replizieren (vergleiche Abbildung 5.2). Zwei weitere Requests haben aufgrund ihrer großen Zahl von 1.000 Jobs und einer niedrigen Job-Wrapper-Effizienz (siehe folgendes Kapitel 6.2) sehr lange gedauert (111 h und 124 h). Bei einem Request schließlich wird die Gesamtlaufzeit von 101 h von wenigen „Nachzüglern“ bestimmt, also von Jobs, die ungleich länger brauchten als die restlichen des Requests.

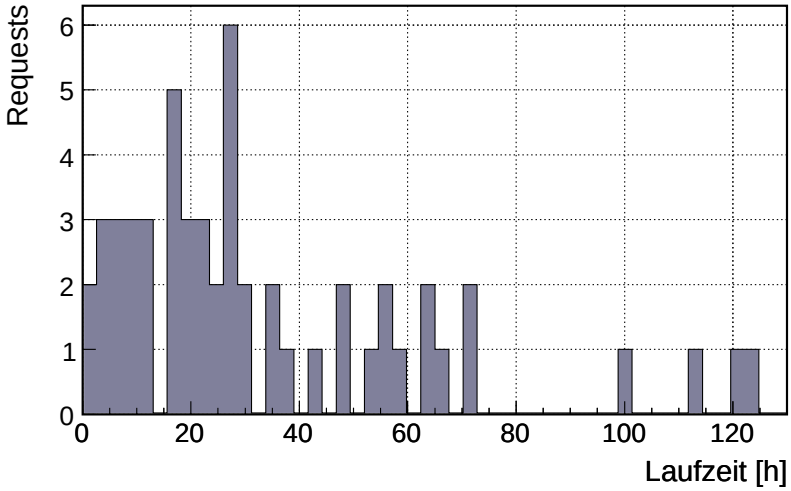

Abbildung 6.4: Laufzeit prozessierter Requests (54 Einträge).

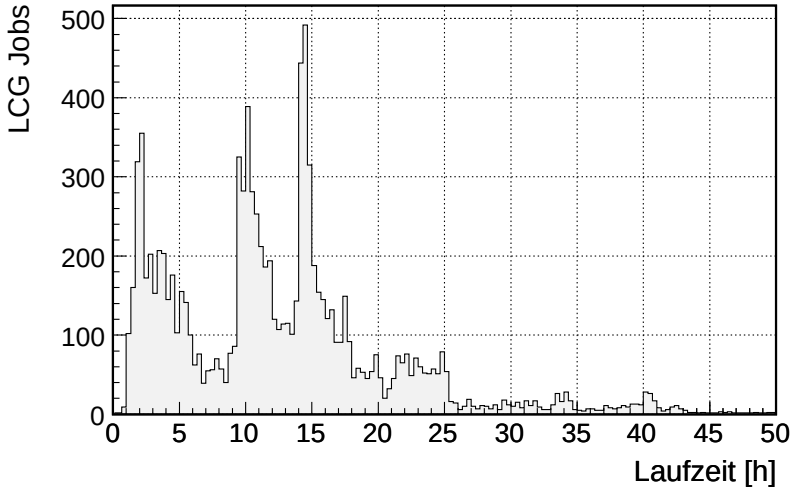

Abbildung 6.5: Laufzeit erfolgreicher LCG-Jobs (9.836 Einträge).

Im Kapitel 2.1, in dem die Direktiven für die MONTECARLO-Produktion im Grid vorgestellt wurden, wurde eine Laufzeit von 15 Stunden pro LCG-Job empfohlen. Abbildung 6.5 zeigt, dass dieses Ziel erreicht werden konnte. Dabei wurden nur Jobs berücksichtigt, die über 1.000 Ereignisse produziert haben, um Testproduktionen auszuschließen. Die scharf ausgeprägten Maxima lassen sich allesamt auf Kombinationen von verschiedenen simulierten physikalischen Prozessen mit verschiedenen Versionen von H1SIMREC (im Wesentlichen HERA-I oder HERA-II) zurückführen. Weder die CPU-Leistung des Worker Nodes noch die Anzahl der prozessierten Ereignisse pro LCG-Job (5.000 bis 20.000) haben ähnlich klaren Einfluss. Die Diskrepanz zwischen der Laufzeit eines LCG-Jobs und der eines ganzen Requests erklärt sich einerseits daraus, dass es eine gewisse Zeit dauert, bis der komplette Request abgeschickt ist. Das kann bei großen Requests durchaus einige Stunden dauern. Zum anderen verlängert es die Laufzeit eines Requests, wenn erfolglose Jobs erneut abgeschickt werden müssen.

Abschließend bietet Tabelle 6.1 einen Überblick über verschiedene Daten der Grid-Produktion von Juni 2005 bis Februar 2006, bevor im folgenden Kapitel die Effizienz des Produktionssystems untersucht wird.

Gesamtzahl prozessierter Requests	109
Zahl der erfolgreichen H1MC-Jobs	11.869
Zahl der produzierten Ereignisse	122, 65 Millionen
Durchschnittliche Laufzeit eines erfolgreichen LCG-Jobs	8:59 Stunden
Durchschnittlich benötigte Zeit, um die DST-Dateien zum `srm-dcache.desy.de` zu replizieren	4:58 Stunden
Durchschnittlich für den Datentransfer benötigte Zeit eines LCG-Jobs	1:03 Stunden
Durchschnittliche Laufzeit eines Requests	34 Stunden
Durchschnittliche Datentransferrate eines LCG-Jobs (gemittelt über die Requests)	3, 39 MB/s
Effizienz des Datentransfers (Zahl der transferierten Dateien dividiert durch die Zahl der benötigten Versuche, gemittelt über die Requests)	74, 8 %
zusätzlich benötigte LCG-Jobs (durch erneutes Abschicken erfolgloser H1MC-Jobs)	4.975
Effizienz des Job-Wrappers (Zahl der erfolgreichen H1MC-Jobs dividiert durch die Zahl der LCG-Jobs)	70, 5 %
Zahl der LCG-Jobs, die vom Rescue Tarball Gebrauch machten	679

Tabelle 6.1: Daten über das Grid-Batch-System und den Job-Wrapper aus der Zeit zwischen Juni 2005 und Februar 2006

6.2 Effizienz

In diesem Kapitel wird die Effizienz des Produktionssystems untersucht. Direkt messbar ist dabei allerdings nur die Effizienz des Job-Wrappers. Sie errechnet sich als Quotient der Zahl der erfolgreichen LCG-Jobs und der Gesamtzahl abgeschickter LCG-Jobs. Viel wichtiger für den laufenden Betrieb des Grid-Batch-Systems ist aber, wieviel manuelle Interaktion letztlich aufgrund einer niedrigen Job-Wrapper-Effizienz nötig ist.

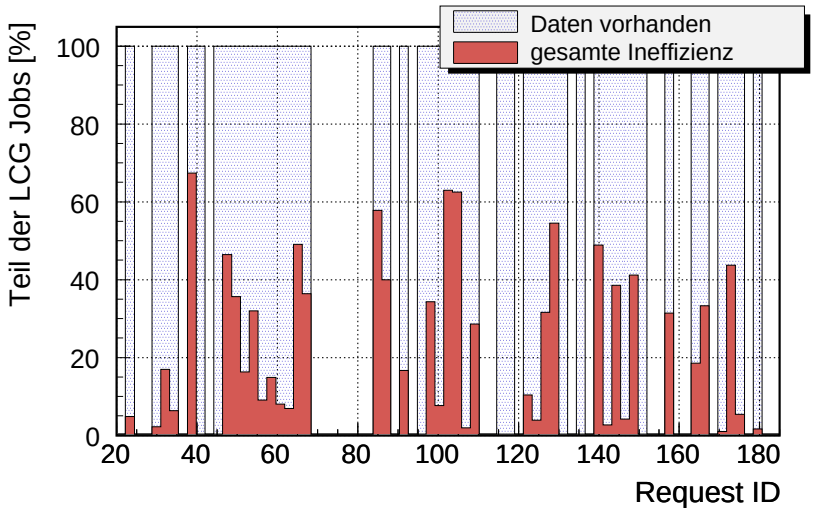

Abbildung 6.6: Anteil der erfolglosen LCG-Jobs bezüglich aller abgeschickten LCG-Jobs.

Abbildung 6.6 zeigt zunächst die Gesamtineffizienz des Job-Wrappers, aufgetragen gegen die interne *Request ID*. Diese identifiziert einen Request eindeutig in der H1MC-Datenbank. Die Request IDs werden fortlaufend vergeben, daher kann die Abszisse auch grob als Zeitachse interpretiert werden. Es ergibt sich eine gesamte Effizienz von $70,5\,\%$. Aus diesem niedrigen Wert, der typisch ist für Gridprojekte dieser Art (zum Beispiel [25]), erwächst die Notwendigkeit, Jobs erneut abzuschicken, wodurch ein H1MC-Job mehrere LCG-Jobs erhält. Abbildung 6.7 zeigt, wieviele es konkret sind – auch diese Verteilung ist typisch [25, 26]. Die Erfahrung zeigt, dass in der Regel erst ab fünf LCG-Jobs manuelle Interaktion erforderlich wird. Mit dieser Annahme ergibt sich aus Abbildung 6.7 ein Wert von ungefähr $97\,\%$ für den Teil der H1MC-Jobs, die keine manuelle Interaktion benötigen. Dieser Wert stellt allerdings eine untere Schranke dar, denn meistens lässt sich ein Problem finden, das alle laufenden Jobs eines Requests betrifft, so dass sich mit einem kleinen Eingriff viele Jobs kurieren lassen.

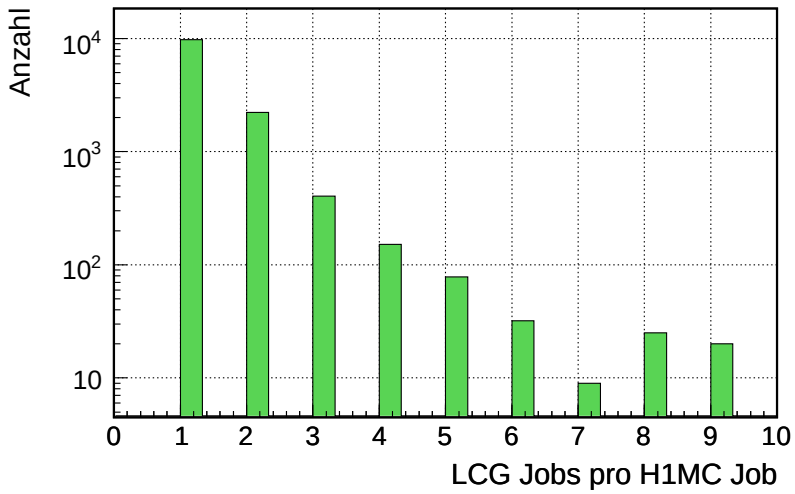

Abbildung 6.7: Die Zahl der LCG-Jobs, die ein H1MC-Job benötigte, bedeutet zugleich, wie oft er abgeschickt werden musste.

Die Ineffizienz des Job Wrappers lässt sich ihren Ursachen nach grob in verschiedene Kategorien aufschlüsseln, wie Tabelle 6.2 zeigt. Die erste Kategorie sind Datentransferfehler im Grid. Die zweite Kategorie bilden die restlichen Fehler im Grid. Außerdem kommen Anwendungsfehler vor, bei denen ein Problem mit H1SIMREC auftrat. Die vierte Kategorie umfasst Benutzerfehler. Als Indikator für die Einteilung werden verschiedene LCG- und Wrapper-Exit-Zustände verwendet, die im Anhang erklärt sind. Dazu kommt der Indikator „LCG-Status blieb stehen“. Er beschreibt LCG-Jobs, deren Zustand sich über lange Zeit nicht mehr veränderte. Die Einteilung in Fehlerkategorien kann allerdings nur grob geschehen, da sich Folgefehler nicht mehr als solche erkennen lassen. Abbildung 6.8 zeigt, wie viele LCG-Jobs an welchem Fehler gescheitert sind.

Fehlerkategorie	Indikator	LCG-Jobs
Datentransferfehler	GET_DATA_ERROR	3.347
	SEVERE_GET_DATA_ERROR	801
andere Gridfehler	LCG *aborted*	1.132
	LCG-Status blieb stehen	299
	SEVERE_ERRORS_OCCURRED	11
	UNKNOWN	47
	MISSING_LIBRARIES	166
Benutzerfehler	LCG *cancelled*	145
Anwendungsfehler	KILLED_BY_WATCHDOG	39
	DST_ERROR	193

Tabelle 6.2: Fehlerkategorien der LCG-Jobs

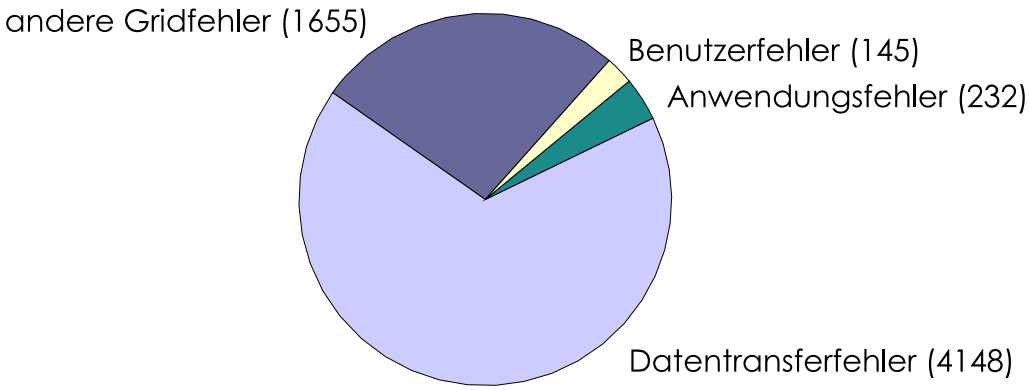

Abbildung 6.8: LCG-Jobs in verschiedenen Fehlerzuständen

Im Folgenden werden die Fehlerkategorien näher untersucht. Zunächst zeigt Abbildung 6.9 die Entwicklung der Datentransferfehler. Für einige Häufungen des Wrapper-Exit-States SEVERE_GET_DATA_ERROR lassen sich einfache Erklärungen angeben. So handelt es sich bei Request 92 um ein statistisches Artefakt, das von zwei Jobs verursacht wird (vergleiche auch Abbildung 6.12). Bei Request 99, 139 und 144 hingegen fehlte die Prüfsumme des H1SIMREC-Pakets, was sich auf einen Benutzerfehler zurückführen lässt.

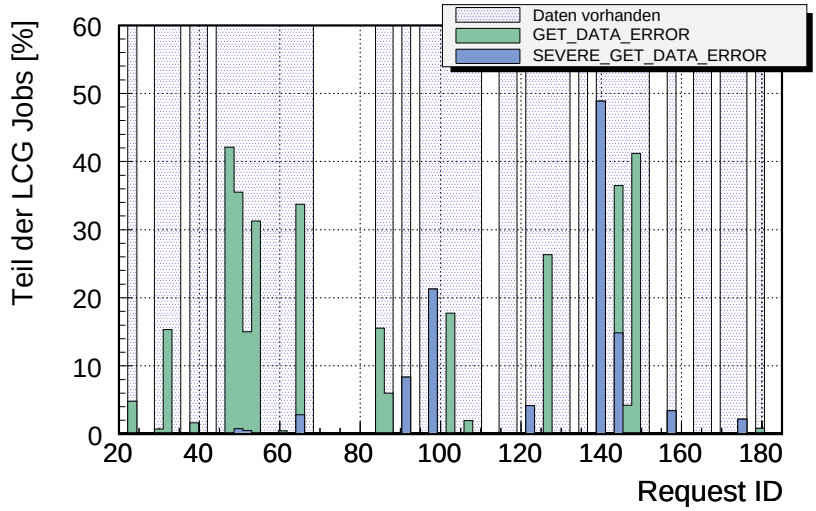

Abbildung 6.9: Anteil von Datentransferfehlern an der Job-Wrapper-Ineffizienz

Abbildung 6.10 zeigt die Entwicklung der anderen Gridfehler und der Benutzerfehler. Die Häufungen des Zustandes LCG *aborted* lassen sich zum einen durch einen zu kurzen Grid Proxy erklären (Request 128 und 129). Zum anderen trat wieder ein statistisches Artefakt auf (Request 63). Zur Zeit von Request 38 allerdings prüfte der Job-Submitter noch nicht auf freie Grid-Sites, weshalb sehr viele Jobs sofort von der Middleware abgebrochen wurden[1]. In der Tat war Request 38 der Anlass, den Test auf freie Sites einzuführen.

Die vierte Fehlerkategorie schließlich wird von Abbildung 6.11 untersucht. Sie zeigt die Entwicklung der Anwendungsfehler, also der Fehler von H1SIMREC. Zunächst gab es noch keine H1SIMREC-Recovery, sie wurde erst mit Request 68 eingeführt. Danach erübrigte sich der Wrapper-Exit-State KILLED_BY_WATCHDOG, weshalb er später nicht mehr auftreten konnte. Allerdings war die Recovery noch bis zur Request 103 fehlerhaft, weshalb hier verstärkt der Wrapper-

[1] vergleiche Kapitel 5.1

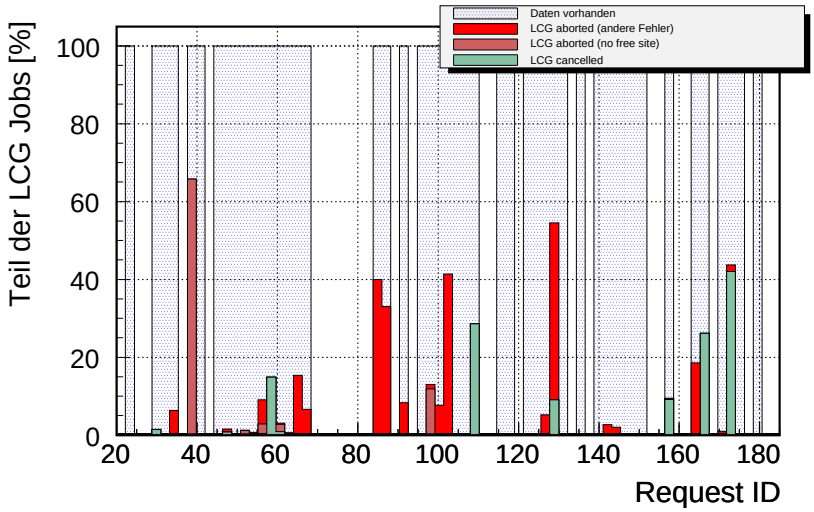

Abbildung 6.10: Anteil von anderen Gridfehlern an der Job-Wrapper-Ineffizienz

Exit-State DST_ERROR auftritt. Die Häufung von nicht behebbaren Fehlern in Request 123 ist ein Statistikartefakt, und bei Request 158 fehlten entscheidende Daten in der Datenbankdatei, was sich auf einen Benutzerfehler zurückführen lässt.

Abschließend zeigt Abbildung 6.12, wieviele LCG-Jobs jeweils zu den Requests gehören, um ein Maß für die statistische Relevanz der zuvor angegebenen Ineffizienzen zu geben.

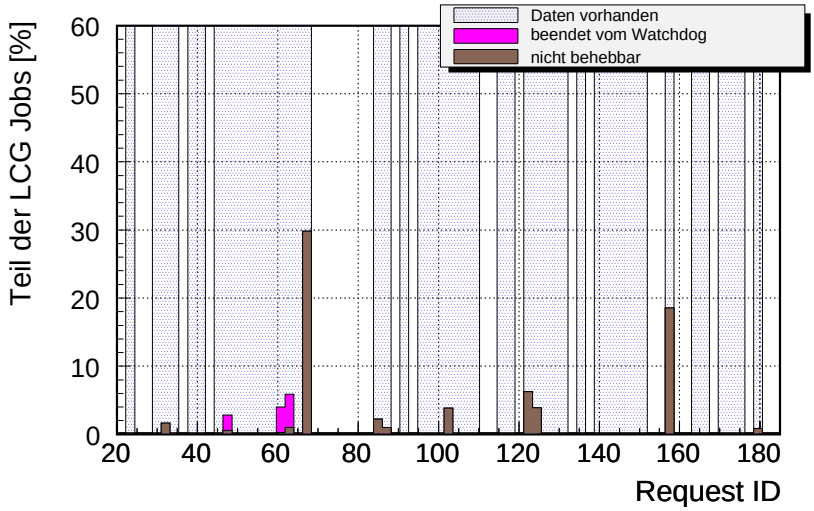

Abbildung 6.11: Der Anteil von Anwendungsfehlern an der Job-Wrapper-Ineffizienz

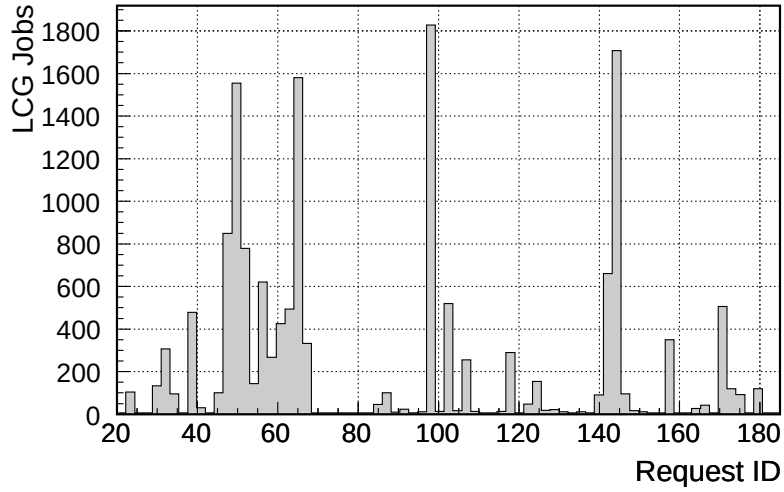

Abbildung 6.12: Zahl der LCG-Jobs der untersuchten Requests

6.3 Validierung gridprozessierten MONTECARLOs

Bevor sich physikalische Analysen auf gridprozessierte MONTECARLO-Ereignisse verlassen können, muss sichergestellt sein, dass die so gewonnenen Ereignisse absolut äquivalent sind zu solchen, die von der klassischen Produktionskette prozessiert wurden. Selbst wenn exakt die gleiche Version von H1SIMREC und exakt die gleiche Datenbank verwendet werden würde, so bleibt doch eine geringe Restunsicherheit. Zum Beispiel könnten geringfügige Unterschiede in den Betriebssystemen der Worker Nodes fatale Folgen haben.

Glücklicherweise ist das nicht der Fall, wie eine Untersuchung von sechs Millionen Ereignissen zeigt. Dazu wurde der gleiche Request einmal von der klassischen Produktionskette prozessiert, und ein weiteres Mal vom Grid-Batch-System. Die Ereignisse wurden dann im Rahmen einer momentan zur Veröffentlichung anstehenden Analyse [28] verwendet, und den gleichen Kontrollprozeduren unterzogen, die auch für klassisches MONTECARLO üblich sind. Die Abbildungen 6.13 und 6.14 stehen beispielhaft für die Ergebnisse dieses Vergleichs: Bis auf statistische Schwankungen, die einer anderen Wahl des Startwerts des Zufallsgenerators geschuldet sind, sind keine Abweichungen zu erkennen.

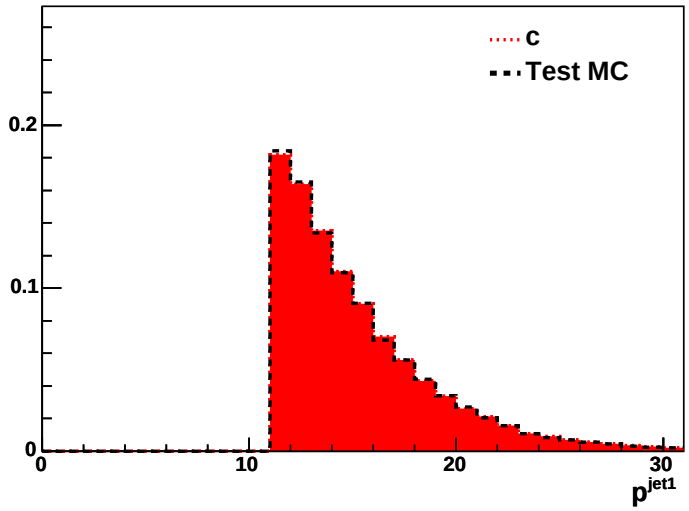

Abbildung 6.13: Transversalimpulsverteilung des ersten Jets, p_t in GeV, Ordinate in beliebigen Einheiten. Mit „c“ ist klassisches MONTECARLO bezeichnet (L. Finke).

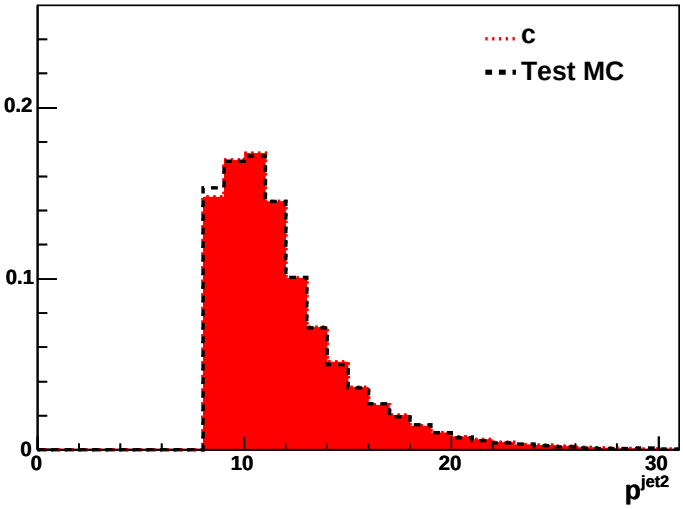

Abbildung 6.14: Transversalimpulsverteilung des zweiten Jets (p_t in GeV, Ordinate in beliebigen Einheiten)

Zusammenfassung

Große Teilchendetektoren wie der H1-Detektor am DESY können nur verstanden werden, wenn den gemessenen Elektron-Proton-Kollisionsereignissen eine ähnliche Zahl an simulierten Ereignissen gegenübergestellt wird. Für diese MONTE-CARLO-Simulationen sind erhebliche Rechnerkapazitäten erforderlich, die bisher ausschließlich von klassischen Rechnerfarmen geliefert werden. Da nach dem Umbau des HERA-Speicherringes im Jahre 2000 und 2001 wesentlich mehr Ereignisse gemessen wurden, stieg der Bedarf an MONTECARLO-Ereignissen und an Rechnerkapazität gleichermaßen.

Eine neue Entwicklung in der Informationstechnologie ist das *Grid Computing*, das Rechenleistung dezentral zur Verfügung stellt. Von besonderer Bedeutung für die Teilchenphysik ist das *LHC Computing Grid* (LCG), das die Rechnerinfrastruktur für die LHC-Experimente am CERN bereitstellt. In der vorliegenden Arbeit wurde eine Produktionsumgebung entwickelt, die das LCG für die MONTE-CARLO-Produktion von H1 nutzbar macht.

Die zugrundeliegende Idee ist, eine große Simulationsaufgabe (einen so genannten *Request*) in viele Teilaufgaben aufzuteilen, die dann als LCG-Jobs auf den Worker Nodes bearbeitet werden.

Die Produktionsumgebung besteht aus zwei wesentlichen Teilen. Der erste Teil ist der *Job-Wrapper*, dessen Aufgabe es ist, auf einem Worker Node das Detektorsimulationsprogramm H1SIMREC auszuführen. Dazu muss er eine Laufzeitumgebung einrichten, wofür ein Datenvolumen von etwa zwei Gigabyte von einem Storage Element kopiert werden muss. Es stellte sich während der Entwicklung heraus, dass der Datentransfer in den aktuellen Versionen der LCG-Middleware noch sehr ineffizient ist. Zusätzlich muss auf etwaiges Fehlverhalten von H1SIMREC automatisch reagiert werden. Der Job-Wrapper implementiert daher umfangreiche Wiederherstellungsmaßnahmen für diese beiden Szenarien, dank derer eine zufriedenstellende Gesamteffizienz erreicht werden kann.

Der zweite Teil ist das *Grid-Batch-System*, das mehrere hundert Jobs gleichzeitig betreuen kann, und zwar mit einem Minimum an manueller Interaktion. Es schickt selbstständig Jobs ab, überprüft regelmäßig ihren Status, holt für beendete Jobs die Output Sandbox ab, wertet diese aus, und leitet abhängig von den

Informationen aus der Output Sandbox weitere Schritte ein. All diese Aufgaben werden von gekapselten Modulen übernommen, die ausschließlich über eine zentrale Datenbank kommunizieren. In ihr ist zu diesem Zweck der Status eines jeden Jobs gespeichert. Während des Testbetriebs wurden weitere Unzulänglichkeiten der LCG-Middleware aufgedeckt, insbesondere die niedrige Geschwindigkeit, mit der Informationen durch das Grid propagieren. Die Module wurden daraufhin verbessert. Zur komfortablen Bedienbarkeit des Grid-Batch-Systems wurde ein Web-Interface entwickelt. Es bietet neben umfangreichen Monitoren auch die Möglichkeit zur manuellen Interaktion mit laufenden Produktionen.

Die Leistungsfähigkeit der entwickelten Produktionsumgebung liegt in der gleichen Größenordnung wie diejenige der klassischen Produktionskette. So wurden im zweiten Halbjahr 2005, während das Grid-Batch-System entwickelt wurde, schon knapp $50\,\%$ der Ereignismenge produziert, die von der klassischen Kette im selben Zeitraum erzeugt wurde. Seit März 2006 ist die in dieser Arbeit entwickelte Produktionsumgebung Teil der offiziellen MONTECARLO-Produktionskette von H1.

Durch die vorliegende Arbeit ist die Rechnerinfrastruktur, die für die virtuelle Organisation *hone* über das LCG verfügbar ist, für die Massenproduktion von MONTECARLO-Ereignissen des H1-Experiments nutzbar geworden.

Anhang A

Portabilität

Bei größeren Softwareprojekten ist es nützlich, sie möglichst portabel und flexibel zu gestalten[1]. Dieser Anhang stellt die Portabilität der in dieser Arbeit entwickelten Produktionsumgebung vor.

Das LHC Computing Grid muss ab 2007 einsatzbereit sein, da es ab diesem Termin für den Betrieb der LHC-Experimente benötigt wird. Daher wird momentan viel an der LCG-Middleware gearbeitet, so dass etwa im Halbjahresrhythmus neue Versionen erscheinen, die auch umgehend von den Grid-Sites installiert werden. Eine Produktionsumgebung, die zum jetzigen Zeitpunkt das LCG nutzen will, muss also leicht an die Veränderungen anpassbar sein. Das in dieser Arbeit entwickelte Grid-Batch-System erreicht die nötige Flexibilität durch die objektorientierte Programmierung, die eine Kapselung der versionsabhängigen Teile ermöglicht. So musste zum Beispiel beim Wechsel von LCG 2.4 auf LCG 2.6 im August 2005 nur diejenige Klasse angepasst werden, die den Datentransfer implementiert. Denn dieser Wechsel brachte im Wesentlichen die Timeout-Option für die Datentransferbefehle, die in Kapitel 3.2 schon angesprochen wurde. Verständlicherweise stellen nicht alle Grid-Sites gleichzeitig auf eine neue Version um. Das macht es nötig, dass der Job Wrapper auf dem Worker Node dynamisch bestimmt, welche LCG-Version installiert ist, und dann die entsprechende Klassenimplementierung wählt.

Eine grundlegende Änderung im Datenmanagement ist jetzt schon absehbar, sie betrifft die Umstellung des File-Catalogs vom EDG-Katalog auf einen neuen, performanteren Katalog: den *LHC File Catalog* (LFC). Dieser soll das Dateisystem des Grids für die Gridbenutzer angenehmer machen, indem er zum Beispiel eine Verzeichnisstruktur und Zugriffsrechte für Dateien einführt, die beide im EDG-Katalog nicht vorhanden sind. Weiterhin sieht der LFC auch eigene Felder für die

[1] Das belegt allein schon der inflationäre Gebrauch der Begriffe „portabel“‘ und „flexibel“ in der IT-Community.

Dateigröße und die MD5-Prüfsumme vor, so dass künftige Versionen der Datentransferbefehle diese benutzen können. Maßnahmen, wie im Kapitel 3.2.1 über die hier entwickelte *get*-Funktion beschrieben, wären dann überflüssig. Das Grid-Batch-System wurde im Rahmen dieser Arbeit auf Kompatibilität mit dem LFC geprüft. Es zeigte sich, dass lediglich geringfügige Anpassungen der Datentransferklasse nötig sind, um erfolgreich auf den LFC zu wechseln. Bei der anstehenden Umstellung werden also keine größeren Probleme erwartet.

In eine gänzlich andere Richtung geht ein Unterprojekt dieser Arbeit, das der klassischen H1-MONTECARLO-Produktion zugute kommt. Bisher muss nämlich der Mitarbeiter, der einen Request auf der (klassischen) DESY-Farm betreut, sämtliche Jobs, bei denen H1SIMREC abgestürzt ist, manuell reparieren. Der Anteil der aus diesem Grund erfolglosen Jobs ist durchaus signifikant; er ist vergleichbar zu den Daten, die aus der Gridproduktion bekannt sind (vergleiche Kapitel 3.3.2). Diese Arbeit würde deutlich erleichtert, wenn die hier entwickelte H1SIMREC-Recovery[2] auch auf klassischen Farmrechnern verwendbar wäre. Daher werden wesentliche Teile der H1SIMREC-Recovery (Kapitel 3.3.2) in die klassische Produktionskette eingebaut.

[2]siehe ebenfalls Kapitel 3.3.2

Anhang B

Dokumentation

B.1 LCG-Zustände

LCG-Zustand	Definition
LCG *submitted*	Der Job ist vom User Interface abgeschickt worden, aber er wurde noch nicht vom Resource Broker angenommen.
LCG *waiting*	Der Job ist vom Resource Broker angenommen worden, aber er ist noch keinem Computing Element zugewiesen worden.
LCG *ready*	Der Job ist einem Computing Element zugewiesen worden, wurde aber noch nicht an dieses geschickt.
LCG *scheduled*	Der Job wartet in der *queue* des Computing Elements.
LCG *running*	Der Job läuft auf einem Worker Node.
LCG *done*	Der Job hat selbstständig zu einem Ende gefunden. Die Output Sandbox steht zum Abholen auf dem Resource Broker bereit.
LCG *aborted*	Der Job wurde von der Middleware beendet, weil zum Beispiel die Laufzeit des Jobs ein Limit erreicht hat.
LCG *cancelled*	Der Job wurde vom Benutzer abgebrochen.
LCG *cleared*	Die Output Sandbox wurde erfolgreich zum User Interface transferiert.

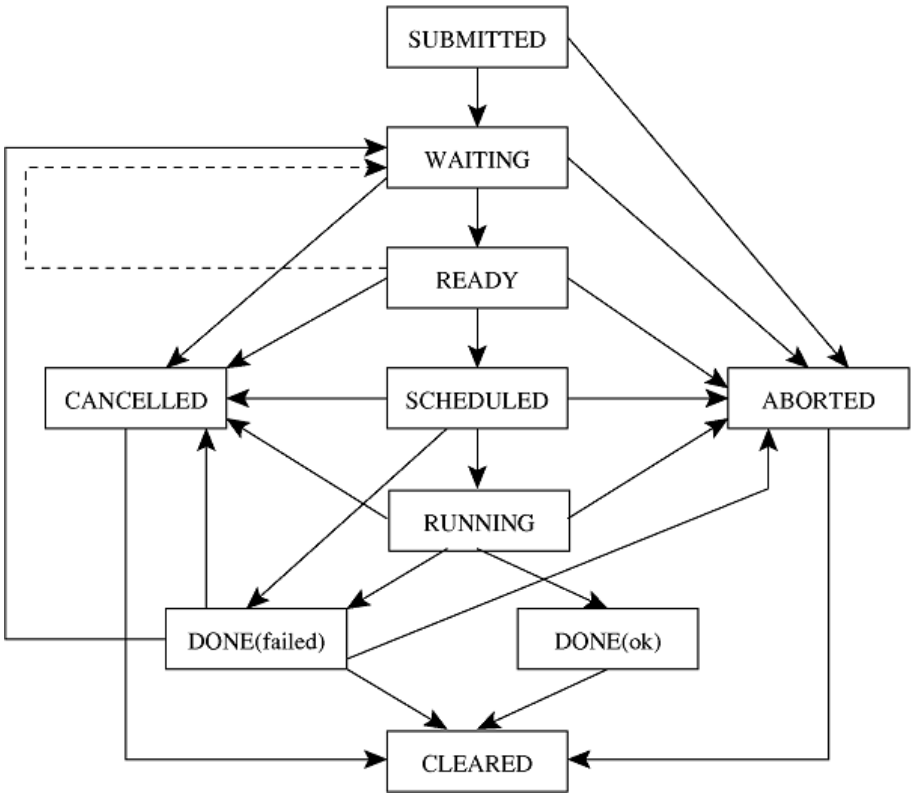

Abbildung B.1: Die verschiedenen LCG-Zustände und die möglichen Übergänge zwischen ihnen [15]. Siehe auch Kapitel 1.4.3.

B.2 H1MC-Zustände

H1MC-Zustand	Definition
H1MC *new*	Der Job wurde vom Job-Maker erzeugt und ist bereit, abgeschickt zu werden.
H1MC *running*	Der Job wurde erfolgreich ins Grid submittiert. Das bedeutet, dass ihn das Grid im Moment bearbeitet, dass also sein LCG-Zustand einer der folgenden ist: LCG *submitted*, LCG *waiting*, LCG *ready*, LCG *scheduled* oder LCG *running*.
H1MC *done*	Der Job hat den Status LCG *done* erreicht und die Output Sandbox steht zum Abholen bereit.
H1MC *received*	Die Output Sandbox wurde erfolgreich vom Resource Broker abgeholt.
H1MC *failed*	Es gab ein Problem mit dem Job, das aber höchstwahrscheinlich dadurch gelöst werden kann, dass der Job erneut abgeschickt wird. Das kann entweder ein Problem sein, das der OSB-Checker entdeckt hat (siehe auch B.3). Oder es kann ein Gridproblem sein, das das Job-Update-Modul gefunden hat (siehe auch 4.4).
H1MC *broken*	Es trat ein Problem auf, das nicht durch Resubmittieren gelöst werden kann. Hier gilt es, manuell einzugreifen und das Problem zu lösen.
H1MC *succeeded*	Alles ist in Ordnung, die DST-Dateien können vom DST-Receiver (siehe 4.7) bearbeitet werden.
H1MC *finished*	Die DST-Dateien befinden sich jetzt im DESY-Tapepool und der Job ist abgeschlossen.

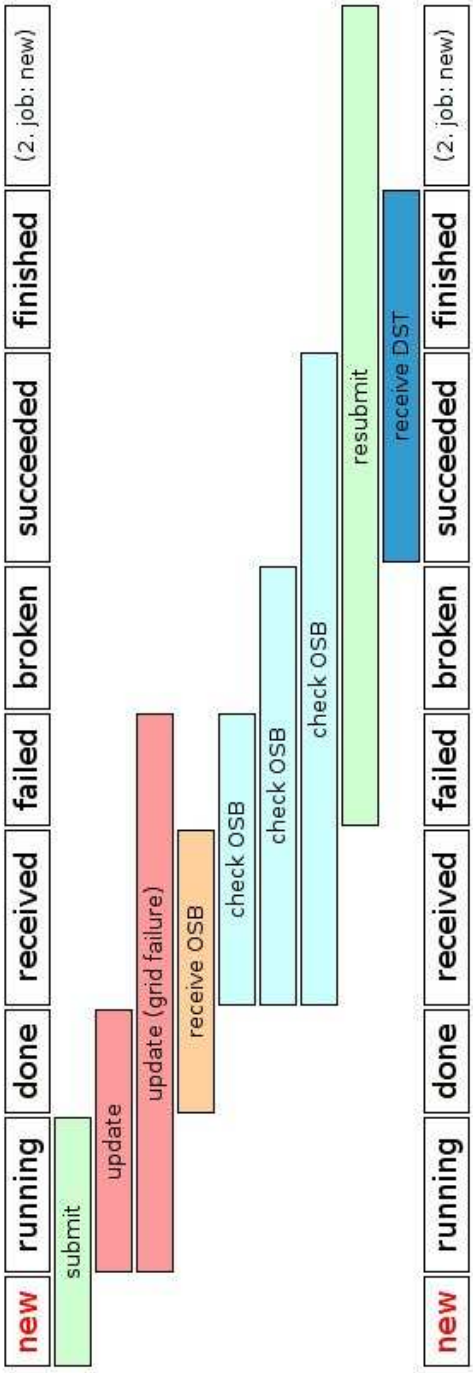

Abbildung B.2: Übergänge zwischen den verschiedenen H1MC-Zuständen

B.3 Wrapper-Exit-States

Wrapper-Exit-State	Definition
SUCCESS	Ok, keine Fehler.
ERRORS_OCCURRED	Ok, aber es traten leichte Fehler traten auf, die vom Job-Wrapper automatisch behoben werden konnten.
PUT_DATA_ERROR	Ok, aber der Rescue Tarball wurde gebraucht.
EXEC_RECOVERED	Ok, aber H1SIMREC wurde neugestartet.
NONRECURRING_ERRORS	Ein Fehler trat auf, der wahrscheinlich nicht wieder passiert. Das kommt zum Beispiel vor, wenn auf dem Worker Node nicht genug freier Festplattenplatz zur Verfügung steht.
MISSING_LIBRARIES	Einer der Anwendungen fehlt eine Systembibliothek. Der Job wird resubmittiert in der Hoffnung, die fehlende Bibliothek auf einem anderen Worker Node vorzufinden.
GET_DATA_ERROR	Der Job konnte benötigte Daten nicht vom Storage Element bekommen. Der Job wird resubmittiert.
KILLED_BY_WATCHDOG	resubmittieren (mittlerweile veraltet, wurde von EXEC_RECOVERED und DST_ERROR ersetzt)
DST_ERROR	H1SIMREC hat trotz Wiederherstellungsversuch keine sinnvollen DST-Outputdateien produziert. Hier ist es nötig, manuell zu interagieren. Daher bekommt der H1MC-Job den Status H1MC *broken*.
SEVERE_ERRORS_OCCURRED	Ein Fehler trat auf, der sich durch Resubmittieren nicht lösen lässt. Manuelle Interaktion ist nötig.
SEVERE_GET_DATA_ERROR	Ein schwerer Fehler beim Datentransfer trat auf. Es fehlt zum Beispiel die MD5-Prüfsumme einer Datei. Manuelle Interaktion ist nötig.
UNKNOWN	Unbekannter Exit State, sollte eigentlich nicht auftreten. Auf Verdacht wird der Job aber resubmittiert.

Anhang C

Glossar

CERN Conseil Européen pour la Recherche Nucléaire. Forschungseinrichtung mit verschiedenen Teilchenbeschleunigern in der Nähe von Genf

Computing Element, CE Ein lokaler Dienst einer Grid-Site, der die Grid-Jobs den freien Worker Nodes zuordnet (Kapitel 1.4.2).

Daemon Programm, das die Module des Grid-Batch-Systems ausführt (Kapitel 4.8).

Datenbankdatei Enthält einen Auszug aus der zentralen H1-Datenbank.

Datenbank, GBS- siehe GBS-Datenbank

Datenbank, H1-Monte-Carlo- siehe H1-Monte-Carlo-Datenbank

Datenbank, zentrale H1- siehe zentrale H1-Datenbank

DESY, Deutsches Elektronen Synchrotron Größtes deutsches Forschungsinstitut für Teilchen- und Synchrotronstrahlungsphysik in Hamburg. DESY ist ein Mitglied der Helmholtzgesellschaft (Kapitel 1.1).

DESY-Tapepool Besonderer Speicherort im Rechenzentrum am DESY. Der Tapepool stellt etwa 120 Terabyte Speicherplatz auf Magnetbändern zur Verfügung, die bei Bedarf von Robotern automatisch ausgelesen werden.

DST-Dateien Enthalten die komplett von H1SIMREC durchsimulierten Ereignisse (Kapitel 1.3.3).

DST-Receiver Modul des Grid-Batch-Systems, das die DST-Outputdateien eines Jobs zum `srm-dcache.desy.de` transferiert (Kapitel 4.7).

EDG, European Data Grid Kapitel 1.4.1

EDG-Katalog Von der VO *hone* verwendeter File Catalog. Soll in naher Zukunft auf den LFC umgestellt werden.

Ereignisgenerator Programm, mit dem die Analysegruppen bestimmte Ereignisse generieren, die in MONTECARLO-Eingangsdateien gespeichert werden. Später wird die Interaktion der in ihnen enthaltenen Teilchen mit dem H1-Detektor durch das H1SIMREC-Programm simuliert (Kapitel 1.3.2).

Ereignis Wenn Elektron und Proton kollidieren, können aus der Kollisionsenergie weitere Teilchen entstehen. Diesen Vorgang nennt man Ereignis (Kapitel 1.1 und 1.2).

File Catalog Ein gridweiter Dienst, der sämtliche Dateien indiziert, die auf den Storage Elements des Grids gespeichert sind. Derzeit verwendet die VO *hone* den EDG-Katalog (Kapitel 1.4.2).

GBS-Datenbank zentrale Datenbank des Grid-Batch-Systems (Kapitel 4.1)

Grid-Batch-System Die in diesem Projekt entwickelte Umgebung zur MONTECARLO-Produktion im Grid.

Grid Computing bezeichnet Methoden, die Rechenleistung vieler Computer innerhalb eines Netzwerks so zusammenzufassen, dass über den reinen Datenaustausch hinaus die (zeitlich parallele) Lösung von rechenintensiven Problemen ermöglicht wird [24] (Kapitel 1.4).

Grid-Job Ein Rechenauftrag, der von einem Worker Node im Grid erledigt wird (Kapitel 1.4.3).

Grid-Site Institut, das dem Grid Rechenkapazität und Speicherplatz zur Verfügung stellt (Kapitel 1.4.2).

GUID, Grid Unique Identifier Eindeutiger Bezeichner einer Datei, die im Grid gespeichert ist (Kapitel 1.4.4).

H1 Die H1-Kollaboration betreibt das H1-Experiment am DESY, einen großen Teilchendetektor (Kapitel 1.2).

H1mc-Zustand Zustand eines H1MC-Jobs (Anhang B.2).

H1-Monte-Carlo-Datenbank Datenbank, in die der MONTECARLO-Koordinator einen Request speichert, den er von einer Analysegruppe erhalten hat (Kapitel 1.3.4).

`https://www-h1.desy.de/icgi-mc/viewRequest` (nur DESY-intern)

H1simrec Ein Programm, das zunächst die Wechselwirkung von Ereignissen, die in MONTECARLO-Eingangsdateien gespeichert sind, mit dem H1-Detektor simuliert. Danach wird aus der simulierten Detektorantwort das Ereignis rekonstruiert (Kapitel 3.3).

H1simrec-Recovery Methode, die H1SIMREC neu startet, falls es zuvor abgestürzt ist (Kapitel 3.3.2).

HERA, Hadron Elektron Ring Anlange Teilchenbeschleuniger für Elektronen und Protonen am Forschungszentrum DESY in Hamburg (Kapitel 1.1).

Input Sandbox, ISB Ein Bündel von Dateien, das zusammen mit dem Grid-Job zuerst auf den Resource Broker, und von dort auf den Worker Node gelangt (Kapitel 1.4.3).

ISB Siehe Input Sandbox.

JDL, Job Definition Language Beschreibungssprache der Grid-Jobs (Kapitel 1.4.3).

Job Ein Rechenauftrag, der von einem Farmrechner erledigt wird.

Job-Maker Modul des Grid-Batch-Systems, das die Jobs vorbereitet (Kapitel 4.2).

Job-Submitter Modul des Grid-Batch-Systems, das die Jobs in das Grid schickt (Kapitel 4.3).

Job-Update-Modul Modul des Grid-Batch-Systems, das den LCG *Status* der Jobs abfragt (Kapitel 4.4).

Job-Wrapper Ein Skript, das auf einem Worker Node eine bestimmte Umgebung bereitstellt, und dann die eigentliche Anwendung startet (Kapitel 1.4.3).

klassische Monte-Carlo-Produktion Bezeichnet die Monte-Carlo-Produktion des H1-Experiments ohne Verwendung von Grid-Technologie (Kapitel 1.3.4).

lcg-cp Middlewarekommando, mit dem sich eine Datei von einem Storage Element auf einen Worker Node oder auf ein User Interface kopieren lässt.

lcg-cr Middlewarekommando, mit dem sich eine Datei von einem User Interface oder von einem Worker Node auf ein Storage Element kopieren lässt.

LCG-Job Ein anderer Name für einen Grid-Job, der vom Grid-Batch-System abgeschickt wurde.

LCG, LHC Computing Grid Ein Computing Grid, das entstand, um die nötige Infrastruktur für die LHC-Experimente zu schaffen. Dieses Grid wird in der vorliegenden Arbeit verwendet, weshalb der Kurzbegriff „Grid“ synonym zu LCG verwendet wird (Kapitel 1.4.1).

LCG-Version Version der Middleware. Zu Beginn dieses Projekts war Version 2.4 aktuell, später 2.6 und zuletzt 2.7.

LCG-Zustand Zustand eines LCG-Jobs (Anhang B.1).

LFC LHC File Catalog

LFN, Logical File Name Beliebig wählbarer Dateiname einer Datei, die im Grid gespeichert ist (Kapitel 1.4.4).

LHC, Large Hadron Collider Großer Teilchenbeschleuniger, der ab 2007 am Genfer Forschungsinstitut CERN betrieben werden soll.

low level grid tools Befehle, mit denen tief in das Datenmanagement des Grids eingegriffen werden kann (Kapitel 1.4.4).

MD5, Message Digest Algorithm 5 Eine kryptographische Hash-Funktion, die einen 128-Bit-Hashwert erzeugt. Die errechneten MD5-Summen werden weitverbreitet zur Integritätsprüfung von Dateien eingesetzt [24].

Middleware Mittelschicht zwischen der Hardware des Grids und der Software des Benutzers. Enthält auch sämtliche Befehle, die auf einem User Interface bereitstehen.

Monte-Carlo-Eingangsdateien Werden von einem Ereignisgenerator erzeugt. Dienen als Eingangsdaten für das Programm H1SIMREC.

Monte-Carlo Numerisches Simulationsverfahren, das sich des Zufalls bedient (Kapitel 1.3).

Monte-Carlo-Produktion Massenhafte Durchführung von MONTECARLO-Simulationsrechnungen.

Noise Files Werden vom Programm H1SIMREC benötigt. Enthalten Informationen über das elektrische Rauschen der Kalorimeter und Spurkammern (Kapitel 1.3.3).

OSB-Checker Modul des Grid-Batch-Systems, das die Output Sandbox eines Jobs auswertet und über das weitere Vorgehen entscheidet (Kapitel 4.6).

OSB-Receiver Modul des Grid-Batch-Systems, das die Output Sandbox eines Jobs abruft (Kapitel 4.5).

OSB Siehe Output Sandbox.

Output Sandbox, OSB Ein Bündel von Dateien, das nach dem Ende des Grid-Jobs zuerst auf den Resource Broker kopiert wird. Von dort kann es der Gridbenutzer zurück zum User Interface holen (Kapitel 1.4.3).

queue Bezeichnet eine Warteschlange für Grid-Jobs, die von einem Computing Element angeboten wird (Kapitel 1.4.2).

RAL, Rutherford Appleton Laboratory Großes Institut in Großbritannien, das das H1-Experiment unterstützt.

Rechnerfarm Ein Zusammenschluss von vielen Einzelrechnern zur parallelen Berechnung großer Rechenaufträge.

Request Der Auftrag, eine große Zahl von Ereignissen mittels des Programms H1SIMREC zu prozessieren (Kapitel 1.3.4).

Rescue Tarball Eine komprimierte Datei, die im Ausnahmefall vom Worker Node mit der Output Sandbox zum User Interface gelangen kann (Kapitel 3.2.2).

Resource Broker, RB Ein gridweiter Dienst, der die Jobs entgegen nimmt, die vom User Interface abgeschickt werden, und passende Grid-Sites sucht, die die Jobs bearbeiten können (Kapitel 1.4.2).

RGMA Relational Grid Monitoring Architecture. Ein datenbankähnliches System, in das von Worker Nodes aus Informationen geschrieben werden können, die dann global zur Verfügung stehen. Siehe auch [19].

srm-dcache.desy.de Ein spezielles Storage Element am DESY, das direkt an den DESY-Tapepool angeschlossen ist.

Storage Element, SE Ein lokaler Dienst einer Grid-Site, der den Zugriff auf Speichermedien wie etwa Festplattenserver regelt (Kapitel 1.4.2).

SURL, Storage URL Eindeutiger Bezeichner einer Datei, die auf einem Storage Element gespeichert ist (Kapitel 1.4.4).

URL Uniform Resource Locator

User Interface, UI Ein Rechner, auf den die LCG-Benutzer Zugriff haben, und auf dem alle nötigen Befehle installiert sind, um mit dem Grid zu interagieren (Kapitel 1.4.2).

Virtuelle Organisation, VO Benutzer des Grids sind in VOs zusammengefasst. Die H1-Kollaboration benutzt die VO *hone* (Kapitel 1.4.1).

Wachhund Ein Überwachungsprozess für H1SIMREC, der es bei grobem Fehlverhalten beendet (Kapitel 3.3.1).

Worker Node, WN Ein Farmrechner einer Grid Farm, auf dem ein Grid-Job ausgeführt wird. Er wird von einem Computing Element angesteuert. Es ist nicht möglich, direkt auf einen Worker Node zuzugreifen (Kapitel 1.4.2).

Wrapper-Exit-State Endzustand des Job-Wrappers (B.3 im Anhang)

zentrale H1-Datenbank Wird vom Programm H1SIMREC benötigt. Enthält im Wesentlichen Kalibrierungskonstanten für den H1-Detektor (Kapitel 3.3).

Literaturverzeichnis

[1] W. HEISENBERG: *Über quantentheoretische Umdeutung kinematischer und mechanischer Beziehungen,* Zeitschrift für Physik **33** (1925), S. 879

[2] ANGELA MEYER: *Nützliche Abfälle – Vom WWW zum Grid* c't **22** (2004), S. 50, `http://www.heise.de/ct/04/22/050/`

[3] H1 COLLABORATION, I. ABT ET AL.: *The H1 Detector at HERA,* Nucl. Instrum. Meth. A **386** (1997), S. 310

[4] H1 COLLABORATION, I. ABT ET AL.: *The Tracking, Calorimeter and Muon Detectors of the H1 Experiment at HERA,* Nucl. Instrum. Meth. A **386** (1997), S. 348

[5] H1 SPACAL GROUP, R.-D. APPUHN ET AL.: *The H1 Lead/Scintillating-Fibre Calorimeter*, Nucl. Instrum. Meth. A **386** (1997), S. 397

[6] H1 COLLABORATION: *A Fast Track Trigger with High Resolution for H1* (1999), Proposal submitted to the Physical Research Commitee PRC 99/06 und interne H1 Note H1-06/99-573

[7] V. BLOBEL, E. LOHRMANN: *Statistische und numerische Methoden der Datenanalyse,* Teubner Studienbücher (1998), S. 171

[8] [7], S. 176

[9] T. SJÖSTRAND ET AL.: *Pythia 6.3 – Physics and Manual,* High Energy Physics – Phenomenology (2003), hep-ph/0308153, Seite 45, `http://www.thep.lu.se/~torbjorn/Pythia.html`

[10] D. WEGENER: *Einführung in die Kern- und Elementarteilchenphysik,* Vorlesungsskript, Dortmund WS 2002/2003, `http://www.physik.uni-dortmund.de/E5/`

[11] R. BRUN ET AL.: *GEANT3, Technical Report*, CERN-DD/EE/84-1, CERN (1987)

[12] I. FOSTER, C. KESSELMANN: *The Grid – BluePrint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, Inc. (1998)

[13] THE GLOBUS ALLIANCE: `http://www.globus.org`

[14] `http://lcg.web.cern.ch/lcg/`

[15] A. DELGADO PERIS ET AL.: *LCG-2 User Guide, Manuals Series*, CERN-LCG-GDEIS-454439, (2005)

[16] `http://goc02.grid-support.ac.uk/googlemaps/`

[17] `https://www.scientificlinux.org/`

[18] `http://www.d-grid.de/`, HEP-Grid

[19] `http://www.r-gma.org/`

[20] A. CYZ: *Persönliche Mitteilung: H1simrec Runner*, (2006)

[21] D. LÜKE: *Short Manual for Monte-Carlo-Production on the H1-computer-farm at DESY,* internes H1 Dokument, (2004)

[22] M. VOROBIEV: *Persönliche Mitteilung: Implementation Details on the H1mc Job Maker*, (2005)

[23] M. ERNST: *Persönliche Mitteilung: Leistungsverbesserung des* `srm-dcache.desy.de`, (2005)

[24] WIKIPEDIA: *Die freie Enzyklopädie,* `http://de.wikipedia.org`

[25] J. CABALLERO, J. M. HERNANDEZ, P. GARCIA-ABIA: *CMS Monte Carlo Production in the Open Science and LHC Computing Grids*, Proceedings of the CHEP06 Conference, Mumbai, India, (2006)

[26] CH. WISSING, M. KARBACH, M. VOROBIEV: *The LCG based mass production framework of the H1 Experiment*, Proceedings of the CHEP06 Conference, Mumbai, India, (2006)

[27] M. KARBACH: *H1 MonteCarlo in the Grid – Staus Report*, Software Plenary, H1 Collaboration Meeting, (Februar 2006)

[28] L. FINKE, A. MEYER: *Measurement of Charm and Beauty Dijet Cross Sections in Photoproduction using the* H1 *Vertex Detector at* HERA*,* to be submitted to Eur.Phys.J, (2006)

[29] H1 COLLABORATION: *Guide for the H1 simulation program H1SIM*, internal H1 note, (1989)

Danksagung

An dieser Stelle möchte ich mich bei allen Professoren, Mitarbeitern und Freunden bedanken, die zum Gelingen dieser Arbeit auf die eine oder andere Weise beigetragen haben. Mein besonderer Dank gilt zuerst Herrn Prof. Dr. D. Wegener. Er ermöglichte es, dass ich diese Arbeit in seiner Arbeitsgruppe am DESY anfertigen konnte. Herrn Prof. Dr. B. Spaan danke ich dafür, dass er als Zweitgutachter zur Verfügung stand. Herzlichen Dank auch an die DESY-IT-Gruppe, besonders an Andreas Gellrich, Matthias de Riese und Michael Ernst, die in beeindruckend kurzer Zeit die vielen Probleme mit der Grid-Infrastruktur am DESY zu lösen vermochten. Zu Dank verpflichtet bin ich auch Max Vorobiev, der mit dem Job-Maker einen Teil zu diesem Projekt beisteuerte: Thank you, Max, for the pleasant time we had not only while developing code but also in the sauna and the gym.

Dann möchte ich mich bei meinem Betreuer Herrn Dr. Christoph Wissing bedanken, der mir mit seiner Erfahrung eine große Hilfe war. So war es für ihn ein Leichtes, kurzerhand „Probleme“ in „Herausforderungen“ umzudefinieren. Auch war er eine stete Quelle süffisantesten Ruhrgebietdeutschs, wobei mir ein Ausspruch besonders im Gedächtnis blieb: *„Tja. Da hast du ja schön am Eimer vorbeigemolken!“*

Danke auch den weiteren Mitgliedern der Dortmund-Heidelberger Arbeitsgruppe, Olaf, Marc-Oliver, Andrea, Katja, Hans-Christian, Victor und Kristin, ohne die meine Zeit am DESY nur halb so schön gewesen wäre.

Großer Dank gebührt meinen sechs Mitbewohnern in meiner ersten Hamburger Wohnung, die mir freundlich ihre Besenkammer vermieteten: Dirk und Jesko danke ich für die Lehrstunden am WG-Kickertisch, die mich auf Hamburger Kneipenabende vorbereiteten. Es kam (durchaus nicht kneipentauglich) vor, dass ein geglückter angeschnittener Kraftschuss mit einem *„Was ein p_t!“* gelobt wurde. Klaus danke ich dafür, dass er mir auch meine letzten Bedenken gegenüber dem „Flow“[1] austrieb und mich so Gelassenheit lehrte. Danke an Jörg dafür, dass er

[1] Englisch etwa für *im Fluss*. Bezeichnet die durch moderne Kommunikationsmittel aufgeweichte Verbindlichkeit, Verabredungen einzuhalten.

mich zum Sportklettern mitnahm, das so mein neues Hobby wurde; und für sein Mantra, das durch trockene Zeiten half: *„Was wir hier machen / ist total wichtig / für die Forschung.“* Bei Stephan bedanke ich mich für die eindrucksvolle Art, mit der er seine Diplomarbeit anfertigte, und die ich beim Schreiben dieser Arbeit immer im Hinterkopf hatte. Dem großen Andi danke dafür, dass er mir trotz meiner ständigen Neckereien ein guter Freund blieb, mit dem sich vorzüglich über die Muse der Wissenschaft[2] philosophieren ließ.

Ebenso gilt mein Dank meinen Mitbewohnern in meiner zweiten Hamburger Wohnung: Danke für den selbstgefangenen Ostseefisch!

Ich bedanke mich bei meinen Freunden vom Kilimanschanzo, mit denen ich beim Freeclimbing Ausgleich zu unzähligen Bürostunden fand. Ganz besonders bei dir, Georg, der du auch über das Klettern hinaus ein guter Freund wurdest.

Danke auch an meinen Dortmunder Freund Dennis Real, der mich in ungezählten Chatsessions in die Geheimnisse des Linux-Betriebssystems einweihte, und dessen kraftvoller Ausspruch *„Das ist doch alles Ranz!“* schnell im ganzen Büro zum willkommenen Ventil wurde, durch das angestauter Druck entweichen konnte.

Schließlich möchte ich mich bei meinen lieben Eltern bedanken, die mich während meines gesamten Studiums in jeglicher Hinsicht unterstützt haben: Danke dafür, Jutta und Walter.

Diese Arbeit wurde mit Mitteln des Bundesministeriums für Bildung und Forschung unter der Projektnummer 05H14BEA/6 gefördert.

[2] Zynische Zungen behaupten, hiermit sei die Zeitverschwendung gemeint.