Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Entwurf und Implementierung eines Plugins für SLURM zum planungsbasierten Scheduling

René Pascal Becker
Matrikelnummer: 5183893
becker29@zedat.fu-berlin.de

Betreuer: Dipl.-Inform. Barry Linnert
Eingereicht bei: Dipl.-Inform. Barry Linnert
Zweitgutachter: Prof. Dr.-Ing. Jochen Schiller

Berlin, 4. Januar 2021

Zusammenfassung

Im Bereich des Cluster-Computings werden diverse Arbeitslast-Verteilungssystem genutzt. Eines dieser Systeme ist SLURM. Bei SLURM werden eingehende Jobs in eine Warteschlange eingereiht und bei verfügbaren Ressourcen abgearbeitet. Ein Job ist hierbei eine Arbeitslast, welche von einem Nutzer an SLURM übergeben wird. Einige Firmen, welche zum Beispiel Computer Assisted Design (CAD) für ihre Produkte nutzen, sind auf Großrechner für Berechnungen zu ihrem Produkt/Teilprodukt angewiesen. Damit Pläne zur Produktentwicklung eingehalten werden können, ist ein planungsbasiertes Scheduling-Verfahren für diese Berechnungen notwendig. Im Rahmen dieser Arbeit werden zwei Plugins für SLURM entwickelt. Das erste Plugin nimmt Jobs vom Nutzer an und übermittelt diese an einen virtuellen Ressourcen Manager, welcher überprüft, ob Ressourcen zu den angegebenen Zeiten zur Verfügung stehen. Das zweite Plugin ermöglicht eine Delegation des Scheduling-Verfahren an den virtuellen Ressourcen Manager, wodurch sichergestellt wird, dass Jobs nach Plan ausgeführt werden können. Die Arbeit setzt sich mit dem Prozess des Entwurfs bis zur Implementierung eines Prototypen auseinandersetzen.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

4. Januar 2021

Inhaltsverzeichnis

1 Einführung

Diese Arbeit beschäftigt sich mit dem Bereich des Cluster Computings. Der Zusammenschluss vieler Rechner wird als Cluster (dt. Rechnerverbund) bezeichnet. Im Wesentlichen gibt es drei Verwendungszwecke eines Clusters. Zum einen werden Hochverfügbarkeitscluster genutzt, um Verfügbarkeit von kritischen Ressourcen zu garantieren. Dies wird durch den Ausschluss von dem sogenannten „Single Point of Failure“ über redundante Computer erreicht. Load-Balancing-Cluster werden in Situationen verwendet, bei denen eine hohe Arbeitslast auf viele Computer verteilt wird. Ein Anwendungsbereich wäre zum Beispiel Webservices, welche leicht parallelisiert werden können. Die für diese Arbeit relevante Kategorie ist die des "High Performance Computing Clusters"(HPC-Cluster). Ein Managementsystem weist verschiedenen Verarbeitungseinheiten Aufgaben zu. Diese Aufgaben können oft aufgeteilt und parallelisiert berechnet werden. Die Verarbeitungseinheiten, welche für das Berechnen der Teilaufgaben zuständig sind, werden Compute Nodes genannt (dt. Rechen-Knoten). Diese Art von Cluster werden oft in der Wissenschaft und bei Renderfarmen verwendet.[17]

Eines dieser Managementsysteme ist die „Simple Linux Utility for Resource Management“ (SLURM), welche die Ressourcen eines HPC-Clusters verwalten kann. SLURM ist hervorzuheben, da es Open Source, flexibel und hochskalierbar ist. Das Managementsystem hat drei Kernkompetenzen. Dazu gehören die Allokation von Compute Nodes für Nutzer, ein Überwachungsmechanismus für die aufgetragenen Arbeitsaufträge (Jobs) und eine Konfliktverhinderung durch das Warteschlangen-Prinzip.[18]

Es existieren jedoch Anwendungsbereiche, bei denen das Warteschlangen-Prinzip zu unpräzise in der Hinsicht ist, als dass der Start- und Endzeitpunkt des eingereihten Jobs nicht immer klar ist. Zudem arbeiten zum Beispiel Unternehmen oft mit Plänen, welches durch das Nicht-Wissen von Start- und Endzeitpunkt erschwert wird. Laut Betreuer war OpenCCS eines der einzigen planungsbasierten Managementsysteme für HPC-Cluster, welches jedoch keine Verwendung mehr findet. SLURM ist ein der populärsten Managementsysteme für HPC-Cluster.[8]

Für Unternehmen, die mit festen Plänen arbeiten und Berechnungen zu bestimmten Zeitpunkten benötigen, sind warteschlangenbasierte Systeme nicht geeignet, da ein Abschluss von Aufgaben zu einer gegebenen Zeit nicht garantiert werden kann. Insofern ist es problematisch, dass SLURM keine integrierte Lösung für dieses Problem anbietet, weshalb das Implementieren eines Plugins, welches SLURM planungsfähig machen soll, Ziel der Arbeit ist. Die Arbeit bettet sich in das vorgeschlagene virtuelle Ressourcen Management System (VRM) aus dem Paper „The virtual resource manager: an architecture for SLA-aware resource management“[1] ein. Der VRM ist vor allem für die Einhaltung von Service-Level-Agreements (dt. Dienstleistungs-Güte-Vereinbarungen) verantwortlich und kann durch Aufteilung von Jobs auf mehrere Cluster zu festgelegten Zeiten die Ausführung nach Plan garantieren. Durch diese Arbeit wird dem VRM die Anbindung an das lokale Ressourcen-Managementsystem SLURM ermöglicht. Dies erlaubt somit ein planungsbasierte Scheduling-Verfahren durch das VRM.

In dieser Arbeit wird das generelle Problem der Anbindung an lokale Ressourcen Managementsysteme, für das in dem Paper beschriebene System nicht gelöst, da

es sich um eine SLURM spezifische Lösung handelt. Für andere HPC Management Systeme könnte der in dieser Arbeit beschriebene Ansatz verwendet werden, falls diese über ähnliche Schnittstellen wie SLURM verfügen. Zudem löst diese Arbeit auch nicht das Problem des automatischen planungsbasierten Scheduling-Verfahren, denn es wird sich auf die Bereitstellung von einem Interface, welches die Kommunikation ermöglichen soll, begrenzt.

Bei der Bereitstellung von einem Interface handelt es sich um eine oft durchgeführte Aufgabenstellung in der Informatik. Eine konkrete Implementierung für ein Interface zum delegierten Scheduling-Verfahren für SLURM an ein Drittprogramm gab es bisher nicht. Dies kann dadurch ergründet werden, dass eine Delegation des Scheduling-Verfahrens nur bei einer zusätzlichen Hierarchiestufe des Scheduling-Verfahren notwendig sein würde und bisher existiert noch kein VRM. Insofern konnten keine konkreten vergleichbaren Arbeiten gefunden werden.

Ziel der Arbeit ist eine Ausführungsreihenfolge nach einem Plan für HPC-Cluster, welche SLURM nutzen, zu ermöglichen. Der existierende Scheduler von SLURM wird abstrahiert werden, sodass ein VRM die Ausführung der Jobs bestimmen kann. SLURM wird somit die Jobs nicht mehr durch das Warteschlangen-Prinzip abarbeiten.

2 Grundlagen

In der Arbeit werden Komponenten von SLURM verwendet und deren Verhalten durch SLURM- und SPANK-Plugins verändert. Für ein Verständnis der Zusammenhänge von Plugins und Komponenten sowie der Grundstruktur von SLURM, werden in diesem Abschnitt diese erläutert.

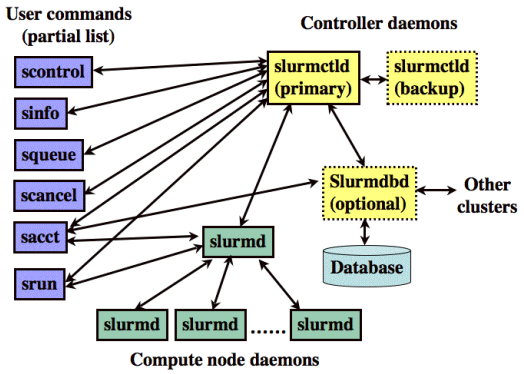

Abbildung 1: Architektur eines SLURM-Clusters
Quelle: https://slurm.schedmd.com/arch.gif

2.1 Komponenten von SLURM

SLURM ist nicht ein einziges Programm, sondern besteht aus mehreren Komponenten. Hierbei sind einige Kommandos für den Nutzer zur Kontrolle von Jobs verfügbar. Darunter: *sacct* für das Anzeigen von Accounting Daten[11], *salloc* für die Allo-

kation von Ressourcen und anschließender Ausführung eines Kommandos[12] und weitere. Ein für diese Arbeit relevantes Kommando ist *sbatch*, welches zur Übergabe von Batch-Skripten genutzt wird. Diese Batch-Skripte sind ähnlich zu normalen Bash-Skriptdateien, welche für die Ausführung von Programmen auf Linux-Systemen genutzt werden. Batch-Skripte werden zum Initialisieren und Starten des Jobs verwendet. Sie enthalten auch Parameter, welche durch das Präfix „#SBATCH“ gekennzeichnet sind. Sie können zur Angabe von Job-Informationen, wie zum Beispiel den Jobnamen, für SLURM genutzt werden. Batch-Skripte werden von *sbatch* zum SLURM-Controller weitergeleitet. Dieser behandelt die Batch-Skripte als Batch-Jobs. Er weist ihnen eine Job-ID und, falls verfügbar, Ressourcen zu. Bei nicht vorhandenen Ressourcen wird der Batch-Job in die Warteschlange von SLURM eingereiht und wartet auf verfügbare Ressourcen. [13]

2.2 SLURM/SPANK Plugin

Durch spezifizierte Schnittstellen bietet SLURM die Möglichkeit, Plugins zu entwickeln, welche für die Anpassung an eine spezifische Infrastruktur genutzt werden können. Diese Flexibilität ist war dadurch motiviert, dass Cluster oft unterschiedliche Verwendungszwecke, Teilkomponenten oder Konfigurationen der Knoten haben. Neben den normalen SLURM Plugins, wird zusätzlich auch noch die Möglichkeit geboten, so genannte SPANK Plugins zu entwickeln. Diese können ohne Quellcode von SLURM entwickelt werden und haben einen geringeren Entwicklungsaufwand, da sie nur ein Interface implementieren und keinen ausgeweiteten Zugriff auf die Datenstrukturen von SLURM haben. Als Beispiel für diesen limitierten Datenzugriff ist die „spank_get_item“ Methode zu nennen, welche nur bestimmte Daten, die fest definiert sind und vom Ausführungskontext abhängig sind, zurückgibt. Diese Methode darf nicht im so gennanten „Allocator Context„ (dt. Allokationskontext) genutzt werden, welcher von *sbatch* und *salloc* aufgerufene Methoden bezeichnet. Zudem können bestimmte Informationen nur nach Abschluss von dem Job abgerufen werden, wie zum Beispiel der Rückgabewert des Jobs. SLURM Plugins haben wesentlich mehr Zugriff auf Datenstrukturen, da sie nicht auf spezifische Methoden für Datenzugriffe (Get-Methoden) angewiesen sind. Bestimmte Methoden von den SPANK Plugins werden zu spezifischen Zeitpunkten ausgeführt, darunter z.B. der Job-Prolog und Epilog, welche für die Vor- und Nachbereitung der Ausführung zuständig sind, weshalb der Ausführungskontext vor Aufruf von Methoden überprüft werden muss.[14]

2.3 Grundstruktur SLURM Cluster

Grundsätzlich besteht ein Cluster, welches SLURM nutzt, aus mindestens einem Controller (slurmctld), welcher die Compute Nodes verwaltet. Der Controller kann an eine Datenbank (slurmdbd), auf der Job Daten persistiert werden können, angeschlossen sein. Diese kann wiederum mit anderen Clustern verbunden werden. Fast alle Kommandos, welche die Nutzer tätigen, werden von dem Controller bearbeitet. Dieser schickt notwendige Informationen an die Head Nodes (dt. Kopf-Knoten) - falls diese vorhanden sind - auf denen eine weitere Instanz von SLURM (slurmd) die Informationen entgegennimmt. (Siehe Abbildung 1)

Die Head Node leitet die Daten anschließend an die untergeordneten Compute Nodes weiter. Falls keine Head Node existiert, werden die Daten direkt an die Compute Nodes weitergegeben. Head Nodes werden als Zugangspunkt für außenstehende Nutzer verwendet. Die Art und Weise, wie die Compute Nodes und Head Nodes untereinander verknüpft sind, wird Cluster-Topologie genannt.[3]

3 Planbasiertes Scheduling Plugin für SLURM

Nach initialer Aufgabenstellung sollte ein Weg gefunden werden, in SLURM ein planungsbasiertes Scheduling-Verfahren zu ermöglichen. Nach einer ersten Beratung mit dem Betreuer wurde die Aufgabenstellung präzisiert. Die Jobs müssten einen Ausführungsplan angeben, welcher von einem Drittprogramm validiert werden müsste, damit ein planbasiertes Scheduling-Verfahren ermöglicht werden kann.

Es ergab sich eine anfängliche Schwierigkeit, die Aufgabenstellung zu verstehen. Konkret hat es sich dabei um das Drittprogramm gehandelt. Es war nicht klar, woher es kommen würde und wie die Schnittstelle aussehen müsste. Durch ein erneutes Beratungsgespräch konnte in Erfahrung gebracht werden, dass das Drittprogramm eine Verwaltungseinheit darstellt, welche eine Gesamtübersicht der Jobs besitzt. Es sollte jedoch nicht im Rahmen dieser Arbeit entwickelt werden, sondern als gegeben gelten. Die Validierung des Ausführungsplans war notwendig, da der Gesamtplan durch das Drittprogramm verwaltet werden würde. Ein Ausführungsplan würde mindestens die Angabe von einen Start- und Endzeitpunkt des Jobs umfassen. Dies ist für die Aufgabenstellung relevant, da dieser angenommen und validiert werden müsste. Im Folgenden wird der Ausführungsplan eines Jobs als Plan bezeichnet und stellt Start- und Endzeitpunkt des Jobs dar.

Es war nicht erkenntlich, welche Schwierigkeiten und Probleme existieren würden und die Aufgabenstellung war relativ abstrakt. Eine vollständige Planung der Implementierung hätte sich daher als schwierig erweisen können, weshalb ein iterativer Entwicklungsprozess gewählt wurde. Ein Vorteil dieser Wahl gegenüber dem Wasserfallmodell ist, dass agiler entwickelt und daher unvorhergesehene Probleme und Schwierigkeiten keine große Abänderung der Planung verursachen.

3.1 Recherche

Aufgrund von diesem Entwicklungsprozess sollte zunächst recherchiert werden, inwiefern eine Implementierung möglich sein könnte. Zudem sollte in Erfahrung gebracht werden, wie SLURM funktioniert und welche Modifikationsmöglichkeiten geboten werden. Ziel war es gewesen, die Aufgabenstellung zu konkretisieren, um eine erste Planung zu ermöglichen.

Im Rahmen erster Überlegungen wurde eine Entwicklung von einem separaten Programm erdacht. Dieses Programm würde die Jobs annehmen und den Zeitplan mithilfe des Drittprogramms validieren und anschließend dem Job mit Start und Endzeitpunkt an SLURM weitergeben. Diese Lösung führt jedoch zu einem neuen Programm, über das die Jobs eingefügt werden müssen.

Anschließend wurde sich über SLURM an sich informiert. Das Ziel war, ein Verständnis für die verschiedenen Möglichkeiten einer Umsetzung von der Aufgaben-

stellung zu bekommen. Dabei war die erste Anlaufstelle, die Webseite von SLURM gewesen, da sich dort eine Beschreibung der Funktionsweise befinden müsste. Alternative Informationsquellen wurden zunächst nicht betrachtet. Zunächst wurde sich darüber informiert, wie Jobs angenommen werden würden, denn dort würde der Plan übergeben werden müssen. Es wurde herausgefunden, dass über *sbatch* Batch-Skripte an SLURM übergeben werden.[13] Nach Rücksprache mit dem Betreuer wurden anschließend nach Modifikationsmöglichkeiten gesucht. Hierbei wurde einmal nach einer Möglichkeit recherchiert, wie der Plan an *sbatch* bzw. SLURM über die Job-Skripte übergeben werden könnte und wie in das Scheduling-Verfahren von SLURM eingegriffen werden könnte. Wichtig bei der Planübergabe war zudem, dass dieser validiert werden können. Hierbei wurde sich gegen eine Validierung im veränderten Scheduling-Verfahren entschieden, da dem Nutzer eine sofortige Rückmeldung gegeben werden soll. Zudem würde sich dies ressourcenschonend auf SLURM auswirken, da Jobs direkt bei der Eingabe abgelehnt werden könnten. Für die Planübergabe wurde nach einer Möglichkeit gesucht, wie Parameter übergeben werden können, da diese Art der Übergabe von Informationen oft in der Informatik genutzt wird.

Für die Annahme von zusätzlichen Parametern konnte ein Plugin gefunden werden.[15] Dieses Plugin war ein SPANK Plugin (Slurm Plug-in Architecture for Node and job (K)control) und nach weiterer Recherche auf der Webseite von SLURM, konnte ein Eintrag mit Beispiel zu diesem Plugin-Typen gefunden werden (siehe [14]). Bei der Recherche nach einer Möglichkeit, das Scheduling-Verfahren zu verändern, konnte eine Übersichtsliste auf der Webseite von SLURM gefunden werden, welche alle Arten von SLURM Plugins aufführt (siehe [18]). Darunter fand sich auch ein Scheduling Plugin-Typ. Nach Rücksprache mit dem Betreuer wurden Beispiele zu dieser Art von SLURM Plugin gesucht, wobei der “Thesis Scheduler„ von Alexandros Gialidis gefunden werden konnte (siehe [16]). Dieses Scheduler Plugin nutzt einen zusätzlichen Thread, welcher für das planungsbasierte starten von Jobs vorteilhaft sein könnte. Es bot sich an, mit den Plugins eine Grundstruktur für das Projekt aufzubauen und ein minimalistisches laufendes System zu erstellen, indem spezifischer Code aus den Plugins entfernt werden würde.

Für die Entwicklung eines separaten Programms, welches *sbatch* als Eingabemöglichkeit für Jobs ersetzen würde, konnten zwei Parameter für *sbatch* gefunden werden. Durch “begin„ und “deadline„ könnten SLURM-interne Methoden genutzt werden, damit die Jobs zum gegebenen Zeitpunkt gestartet werden. Damit könnte das separate Programm einen Plan annehmen und validieren und anschließend diese Parameter für *sbatch* setzen und den Job übergeben.

Aus der Recherche konnten somit zwei Ansätze gefunden werden. Der erste würde ein separates Programm beinhalten, welches *sbatch* als Eingabemöglichkeit für Jobs ersetzen würde und dadurch vor der Eingabe einen Plan validieren und durch Parameter den Start- und Endzeitpunkt des Jobs für *sbatch* setzen könnte. Der zweite Ansatz würde zwei Plugins umfassen. Ein SPANK Plugin und ein SLURM Scheduler Plugin. Der Plan würde über *sbatch* als Parameter übergeben werden und das SPANK Plugin könnte diesen zur Validierung nutzen. Das Scheduling-Verfahren würde durch ein SLURM Scheduler Plugin angepasst werden.

Bei der Recherche kamen oft Schwierigkeiten bei der Verständlichkeit der Dokumentation von SLURM auf. Viele Methoden wurden nur oberflächlich erläutert. Für

das Verständnis waren weitere Kenntnisse über den Aufbau von SLURM notwendig, welche zu diesem Zeitpunkt noch nicht vorhanden waren.

3.2 Entwurf - Projektstruktur

Nach den in der Recherche gewonnenen Kenntnissen, konnte eine Evaluation der beiden Ansätze folgen. Der erste Ansatz, welcher ein separates Programm beinhaltet, wurde als weniger aufwendig als der zweite Ansatz eingeschätzt, da sich hier auf ein Programm begrenzt wird. Die Nutzer könnten jedoch durch direkte Übergabe ihrer Jobs an *sbatch* das Programm umgehen. Aus diesem Grund kann auch der Ausführungszeitpunkt von Jobs nicht garantiert werden, da das Scheduling-Verfahren weiterhin SLURM überlassen wird und daher von Grund auf warteschlangenbasiert ist.

Die zweite Lösung beinhaltet zwei Plugins. Ein SPANK Plugin für die Annahme eines Parameters, welcher über *sbatch* übergeben wird, und ein SLURM Scheduler Plugin, welches den Job zu gegebener Zeit ausführt. Diese Lösung hat den Vorteil, dass es sich um eine integrierte Lösung handelt. Der Nutzer kann das gewohnte Interface nutzen und soll nur einen zusätzlichen Parameter angeben. Ein weiterer Vorteil dieser Lösung ist die Flexibilität, welche geboten wird. Rin SLURM Scheduler Plugin könnte zusätzliche Rückmeldungen an das Drittprogramm liefern. Die Entwicklung von zwei Plugins verursacht jedoch auch einen größeren Aufwand.

Es wurde aufgrund der Nachteile des separaten Programms sich für die Lösung mit einem SPANK und SLURM Scheduler Plugin entschieden. Hierbei war die Möglichkeit, das separate Programm zu umgehen, ausschlaggebend und die zusätzliche Flexibilität durch zwei Plugins ein relevanter Grund für diese Entscheidung.

Nach dem iterativen Entwicklungsprozess wurde im Rahmen des ersten Entwurfs ein Ziel für die erste Iteration festgelegt. Die beiden Beispiele, welche in der Recherche gefunden wurden, sollten kompiliert und ausgeführt werden können. Dieses Ziel hat somit die Aufsetzung einer Grundstruktur für das Projekt sowie einer Umgebung für die Ausführung bzw. Inklusion der Plugins benötigt. Nach Absprache mit dem Betreuer würde ein Testcluster zur Verfügung gestellt werden können.

Es wurde sich für dieses Ziel entschieden, da der Aufwand nicht zu hoch oder niedrig erschien. Die Entwicklung der einzelnen Plugins wurde als ähnlich aufwendig eingeschätzt, da Kenntnisse von Unix-Systemen in einer Selbsteinschätzung als nicht angemessen eingestuft wurden.

Das SPANK Plugin, welches im Rahmen dieser Arbeit entwickelt wird, wird folgend als PSPS-Helper (Planbased Scheduling Plugin for SLURM - Helper Plugin) bezeichnet werden, wobei PSPS das SLURM Scheduler Plugin, welches auch im Rahmen dieser Arbeit entwickelt wird, bezeichnet.

3.3 Projektstruktur und Testumgebung

Nach der Zielsetzung der ersten Iteration war der erste Schritt die Entscheidung, welche Projektstruktur genutzt werden sollte. Es wurde sich für zwei einzelne Projekte innerhalb von einem Git-Repository entschieden. In dem einen Projekt würde das PSPS-Helper- und in dem anderen das PSPS-Plugin entwickelt werden. Es wurden

keine Alternativen gesucht und sich daher nur an anderen Projekten aus der Vergangenheit orientiert. Vorteilhaft ist diese Struktur in dem Sinne, dass sie sehr einfach ist.

In das PSPS-Helper Projekt wurde der Beispielcode, welcher auf der SLURM Webseite verfügbar war, eingefügt und in das PSPS Projekt wurde der Code von Alexandros Gialidis eingefügt (siehe [16]). Anschließend wurde der Quellcode, welcher ausgeführt wird, minimiert. Dabei wurden alle Teile entfernt bzw. auskommentiert, die nicht zu der Grundstruktur eines SPANK beziehungsweise SLURM Scheduler Plugins gehört haben. Hierbei wurde sich an die recherchierten Schnittstellen, welche auf der Webseite von SLURM zur Verfügung stehen, gehalten.

Bei dem Versuch, die Projekte zu kompilieren, traten Probleme auf. Es war zunächst unklar, wie die Plugins kompiliert werden sollen. Anlaufstelle war hier die Webseite von SLURM. Durch eine erneute Suche auf der Webseite von SLURM (siehe [10, 14]), konnte in Erfahrung gebracht werden, dass nur eine SPANK Headerdatei (spank.h) zum kompilieren für den PSPS-Helper und der gesamte Quellcode von SLURM für das kompilieren von PSPS nötig sein würde.

Damit dieses Umfeld für die Kompilierung bereitgestellt werden konnte, war es nötig, die Projektstruktur anzupassen. Der Grund für eine notwendige Anpassung ist die grundlegende Projektstrukturdifferenz zwischen SLURM und meiner Projektstruktur gewesen. Es hat sich angeboten, die bereits vorgefertigte Struktur von SLURM zu nutzen und diese um die beiden Plugins zu erweitern. In der Struktur ist ein Ordner für Plugins vorhanden, welcher Unterordner für die einzelnen Plugin-Typen enthält. Das PSPS wurde zu dem Ordner für SLURM Scheduler Plugins hinzugefügt. Der Scheduler-Typ wurde “planbased„ genannt und die Makefile von Alexandros Gialidis konnte in diesem neuen Kontext mit kleinen Anpassungen auch genutzt werden. Für den PSPS-Helper wurde ein separater Ordner angelegt, welcher dennoch in der Kompilierung von SLURM mit einbegriffen wurde. Alternativ hätte der PSPS-Helper in einem eigenen Projekt mit der “spank.h„ kompiliert werden. Dies hätte den Vorteil gehabt, dass die Projektstruktur die unterschiedlichen Umfelder, in denen die Plugins kompiliert werden müssen, berücksichtigt hätten. Allerdings wäre diese Variante mit zusätzlichem Aufwand verbunden gewesen, da so das Kompilierungssystem von SLURM nicht mitgenutzt werden konnte und dadurch eventuell zusätzliche Maßnahmen für die Durchführung von Testfällen notwendig sein könnten. Zudem erschien es sinnvoll, alle Plugins im Ordner von SLURM zu entwickeln, der “plugins„ genannt wurde.

Durch diese Anpassung der Projektstruktur war es nun möglich, SLURM mit den beiden Plugins zu kompilieren. Zu diesem Zeitpunkt war das Testcluster noch nicht verfügbar, weshalb Alternativen gesucht wurden. Bei erneuter Recherche konnte bei der README-Datei von dem Plugin, welches von Alexandros Gialidis entwickelt wurde, der Verweis auf das SLURM-Docker-Cluster gefunden werden (siehe [9]). Mithilfe der Dateien von diesem Projekt könnte ein lokales Testcluster über Docker-Container aufgesetzt werden.

Es ergaben sich Probleme bei dem Versuch, dieses Testcluster einzurichten. Die Dockerdatei musste entsprechend der Projektstruktur angepasst werden. Zudem ist bei dem anschließenden Aufsetzen des Dockerclusters aufgefallen, dass einige Paketabhängigkeiten nicht eingetragen waren, welche anschließend nachgetragen wurden.

Nach dem ersten vollständigen Durchlauf des Erstellungsskripts für das Testcluster wurden die in der Detailseite von SLURM-Docker-Cluster beschriebenen Befehle genutzt, um sich auf den SLURM-Controller auf eine Bash einzuloggen. Dies war erfolglos, denn der slurmctld war inaktiv. Bei der Fehlereingrenzung wurde davon ausgegangen, dass SLURM-Docker-Cluster und SLURM korrekt konfiguriert waren. Dies wurde allerdings nicht überprüft. Es wurde dadurch davon ausgegangen, dass der Fehler in den eingefügten Plugins sein müsste, weshalb die Plugins zuerst angepasst wurden. Nach erneutem Fehlverhalten wurden die Plugins entfernt und damit die Konfiguration überprüft. Dies war erneut nicht erfolgreich. Nach einem Beratungsgespräch mit dem Betreuer kam die Vermutung auf, dass die Verbindung zwischen den einzelnen Komponenten des Clusters fehlgeschlagen beziehungsweise Einschränkungen durch eine Firewall vorhanden waren. Nach Überprüfung beider Möglichkeiten stand fest, dass dies auch nicht möglich sein könnte.

Schließlich wurde versucht, Log-Dateien zu finden, welche die Probleme möglicherweise zeigen könnten. Hier war der erste Ansatz, dass SLURM Logdateien erstellen müsste. Diese konnten nur über den Zugriff von der Bash aufgerufen werden. Ein manuelles starten von dem slurmctld war nicht möglich, da dieser sich sofort geschlossen hat. Über die Anleitung von Docker konnte herausgefunden werden, dass bei Docker auch Log-Dateien einsehbar sind (siehe [2]). Diese wurden eingesehen und es konnte festgestellt werden, dass die Konfigurationsdateien die falsche Berechtigung für die SLURM Nutzer Gruppe haben. Die Veränderung der Berechtigung wurde in der Dockerdatei vorgenommen. Des Weiteren mussten verschiedene Parameter in der Konfigurationsdatei angepasst werden, bis das Testcluster erfolgreich starten konnte.

Die Plugins wurden anschließend wieder eingefügt und sie konnten erfolgreich mit SLURM nach kleinen Anpassungen, welche in den Docker-Log-Dateien als Fehler ausgegeben wurden, gestartet werden.

3.4 Entwurf - Kommunikation, Interface und SPANK

Nach Aufstellung der Grundstruktur konnte die Implementierung der Plugins und deren Kommunikation mit dem Drittprogramm, welches für die Validierung des Plans zuständig sein würde, geschehen. Für die zweite Iteration ergaben sich mehrere Möglichkeiten für die weitere Planung.

Es könnte der PSPS-Helper für die Annahme der Jobs als Erstes implementiert werden. Ein Vorteil davon wäre gewesen, dass die Implementierung dem Pfad folgen würde, den der Job bei den Plugins nehmen würde, weshalb die Entwicklung der nächsten Teilkomponente eindeutig gewesen wäre. Denn steht die Schnittstelle des PSPS-Helper zu Drittprogramm oder PSPS fest, so ist deren Implementierung eindeutig auf die in dem Interface angegebenen Spezifikationen beschränkt. Nachteil dieser Herangehensweise ist, dass bei Implementierung der nächsten Komponente auffallen kann, dass das Interface ungeeignet sein könnte. Bei diesem Fall müsste das Plugin erneut angepasst werden.

PSPS war eine Alternative zu dem PSPS-Helper als Implementierungsziel der zweiten Iteration. Vorteile und Nachteile dieser Möglichkeit sind ähnlich zu den oben genannten. Durch Festlegung von dem Interface des PSPS würden nötige Daten, welche der PSPS-Helper bereitstellen müsste, bereits erfasst werden. Dies kann jedoch

auch dazu führen, dass diese Daten nicht erhoben werden können und das PSPS überarbeitet werden müsste. Hierbei kann noch nicht gesagt werden, ob es sich um ein strukturelles Problem oder ein Implementierungsproblem handeln würde, da dies von der Art der Daten abhängt.

Ein Kompromiss zwischen den beiden Herangehensweisen würde die Entwicklung von einem Interface für alle Komponenten sein. Hierbei wäre die Festlegung der Aufgaben und erwünschten Logik durch ein Interface für die konkrete Implementierung der Plugins nützlich. Grund hierfür ist, dass bei Fehlern von dem Interface, welche während der Entwicklung von den Plugins aufkommen könnten, das Interface angepasst werden könnte und keine Änderung in den Teilkomponenten nötig wäre.

Es wurde der Kompromiss für das Entwicklungsziel gewählt. Es sollte ein Interface für die Kommunikation der beiden Plugins mit dem Drittprogramm festgelegt werden. Dieses Ziel wurde aufgrund der persönlichen Präferenz für die Entwicklung von einem Interface vor der Implementierung der Plugins an sich gewählt. Für den gesamten Entwicklungsprozess würde dieses Entwicklungsziel auch den Entwurf der Plugins im nächsten Schritt einfacher machen, da deren Funktionsweise durch das Interface festgelegt sein würde.

Für die Umsetzung des Entwicklungsziels war zudem relevant, ob das Drittprogramm lokal in denselben Cluster oder sogar auf einem anderen System ausgeführt wird. Nach einem Beratungsgespräch mit dem Betreuer stand fest, dass dies nicht festgelegt ist. Für größtmögliche Abdeckung der Möglichkeiten wurde sich daher für eine Socket-basierte Lösung für das Interface festgelegt. Durch Sockets können sowohl lokale als auch Programme in denselbem System und anderen Systemen erreicht werden.

Weiterhin war es nötig, ein Protokoll für die Datenübertragung festzulegen, welches den Kernaspekt der Interfaces zwischen Plugins und Drittprogramm bildet. Es gibt viele Möglichkeiten, wie Daten übertragen werden können. Für die Entwicklung der Plugins im Rahmen dieser Arbeit wurde menschliche Lesbarkeit für das Debuggen sowie Flexibilität der Datenübertragung priorisiert. Aus diesem Grund wurde sich für das JSON-Format entschieden, wobei eine Variable die Art der Anfrage festlegen müsste, damit die Eindeutigkeit der Anfragen gewährleistet sein könnte.

3.5 Entwicklung - Interface und Parameterannahme

Damit Verbindungen aufgebaut und JSON-Anfragen gesendet und empfangen werden können, müssen diese zunächst durch Sockets etabliert werden. Dazu wurden die Unix Sockets verwendet. Es wurde sich für TCP-Sockets entschieden, da sich dadurch leichtere Fehlerbehandlung versprochen wurde. Für die Implementierung wurden die Beispiele aus der Dokumentation für Unix Sockets genutzt (siehe [7]). Diese wurden in einem eigenen Objekt abgekapselt, sodass eine spätere Implementierung einer Verschlüsselung dort lokal stattfinden könnte, welche bei einem System, bei dem das Drittprogramm in einem externen System vorhanden ist, notwendig sein kann.

Für das Testen der Sockets wurde der PSPS-Helper so erweitert, dass die Eingabe eines optionalen Parameters möglich ist. Hierfür wurde der Beispielcode für SPANK Plugins von der SLURM Webseite eingebunden und angepasst (siehe [14]). Der „renice“ Parameter, welcher in dem Beispielcode als neuer Parameter hinzugefügt wurde,

wurde in „plan“ umbenannt und die entwickelte Verbindungsimplementierung wurde eingebunden. Der PSPS-Helper hat sich mit einem anderen Rechner verbunden und den Parameter gesendet. Per dem Programm “netcat„ wurden die Daten auf den anderen Rechner empfangen.

Für die Erstellung und Decodierung von JSON-Anfragen gab es im wesentlichen zwei Möglichkeiten. Entweder der nötige Quellcode würde selber geschrieben oder eine Bibliothek eingebunden werden. Es wurde sich für die Einbindung einer Bibliothek entschieden, da dies wesentlich weniger Aufwand für die Entwicklung der Plugins bedeuten würde. Zudem ist die Entwicklung eines JSON-Parsers auch nicht Teil dieser Arbeit.

Es existieren einige JSON-Parser für C, welche über die offizielle Webseite von JSON gefunden werden konnten (siehe [4]). Nach Recherche über die existierenden Bibliotheken wurde sich aufgrund der verfügbaren Beispiele auf Git für JSON-C entschieden (siehe [5, 6]). Hier wurde eine einfache und verständliche Nutzung der Bibliothek priorisiert. Es gibt keine weiteren Gründe für diese Entscheidung - es hätten auch andere Bibliotheken genutzt werden können.

Es ergab sich das Problem, dass es nicht bekannt war, wie externe Bibliotheken auf Linux eingebunden werden könnten. Jedoch war in der Dockerdatei ersichtlich, dass einige Development-Pakete (dt. Entwicklungspakete) zur Kompilierung notwendig sind. Analog dazu konnte auch die neue Bibliothek eingebunden werden. Das Development-Paket von JSON-C wurde in der Dockerdatei eingebunden und ähnlich zu anderen Bibliotheken in der Makefile hinzugefügt. Das Development-Paket fügt den Standardpfaden für Entwicklungsdateien neue Dateien hinzu, weshalb diese nach Installation auch ohne weitere Konfiguration der IDE genutzt werden konnten.

Nachdem die Kommunikation zwischen Drittprogramm und Plugins ermöglicht wurde, konnte anschließend die Kommunikation modelliert werden. Die Netzwerkstruktur begrenzt sich hier auf einen Server, auf dem das Drittprogramm läuft, und einem oder mehreren Clients, wobei jeder Client ein Cluster darstellen würde. Es bieten sich daher das Server-Client und Subscribe-Publish Kommunikationsmodell an, da es sich um einen Server handelt. Weitere Kommunikationsmodelle wurden nicht evaluiert. Der Vorteil des Server-Client Modells wäre die Einfachheit gewesen, jedoch könnte damit nicht der Sachverhalt modelliert werden, wenn der PSPS-Helper außerhalb eines Clusters ausgeführt werden würde. Es wäre nicht eindeutig, welches Cluster diesen Job ausführen soll. Dies ist insofern ein Problem, dass *sbatch* diesen Job ohne das Plugin an den konfigurierten Cluster weiterleiten würde. Dieses Problem würde bei einem Subscribe-Publish Modell nicht existieren, da jeder Cluster einem eigenen Kanal zugewiesen sein würde.

Aus diesem Grund wurde sich für das Subscribe-Publish Modell entschieden. Jedes Cluster wäre demnach einem Kanal zugewiesen und die PSPS-Helper würden den Kanal des Clusters nutzen, für den sie konfiguriert wurden. Die Jobs würden auf dem Kanal von dem PSPS-Helper veröffentlicht werden und die Cluster würden diese durch den PSPS, nach Validierung durch das Drittprogramm erhalten. Um diese Funktionalität zu ermöglichen, müssten mindestens Subscribe, Publish und eine Statusanfrage existieren. Letztere wird durch den Validierungsprozess notwendig, da dort eine Rückmeldung an den Nutzer erfolgen soll.

Die Anfragen wurden nach den Überlegungen in einer ersten Grundstruktur im-

plementiert. Die spezifischen Inhalte der Anfragen würden durch die konkrete Implementierung der Plugins bestimmt werden. Auf welche Datenstrukturen zugegriffen werden kann, war zu der Zeit noch nicht erfasst. Es wurde jedoch erwartet, dass von SPANK Plugins aus auf den vom Nutzer eingegebenen Job zugegriffen kann, welcher somit für den PSPS-Helper zur Verfügung stehen müsste.

Für die Durchführung von Testfällen für Anfragen musste ein Dummy-Drittprogramm entwickelt werden. Hier wurde Einfachheit priorisiert, sodass dieses Programm auf jede Anfrage dieselbe Antwort zurücksendet. Bei Durchführung von Tests wurde festgestellt, dass manche Nachrichten nicht in einem Teil ankamen. Dies konnte auf das TCP-Protokoll zurückgeführt werden. Zur Lösung dieses Problems wurde ein 4 Byte Integer vor jede Anfrage angefügt, welcher die Länge der Anfrage repräsentiert. Anfänglich wurde hier falsch mit Zeigern umgegangen und dies spät erkannt, weshalb dieser Schritt mehr Zeit in Anspruch genommen hatte.

3.6 Entwurf - Implementierung der Plugins

Nach der zweiten Iteration konnten bereits Anfragen über eine Schnittstelle von dem PSPS-Helper an das Dummy-Drittprogramm gesendet und beantwortet werden. Die Programmlogik, welche diese Anfragen verwenden soll, war jedoch noch nicht implementiert. Zudem war der PSPS noch nicht an das Dummy-Drittprogramm angebunden.

Für die dritte Iteration ergab sich ein erhöhter Zeitdruck, welcher von der Abgabe-Deadline dieser Arbeit und persönlichen Umständen verursacht wurde. Die Entwicklung der Plugins sollte zeitnah abgeschlossen werden, sodass bei Gleichhaltung der Länge einer Iteration keine vierte Iteration möglich wäre. Es boten sich mehrere Möglichkeiten an.

Zum Einen könnte die Länge der dritten Iteration verkürzt werden und damit die Implementierung der Programmlogik zweiteilig geschehen. In einer Iteration könnte die Programmlogik des PSPS-Helper und in der nächsten die des PSPS implementiert werden. Ein Vorteil dieser Planung wäre, dass die Entwicklung lokalisiert in einer Teilkomponente stattfinden würde. Durch die Lokalität der Entwicklung könnten Fehlerquellen leichter identifiziert werden, weshalb die Entwicklungs- und Testungsdauer reduziert werden würde. Ein Nachteil wäre, dass Interfaceänderungen, welche sich in der vierten Iteration ergeben, nicht in die Entwicklung während der dritten Iteration einbezogen werden könnten. Dies kann zu erhöhtem Aufwand durch die Abänderung bereits existierender Programmlogik führen.

Eine weitere Möglichkeit für die Aufteilung in zwei Iteration wäre die Entwicklung der Programmlogik des PSPS in der dritten und die des PSPS-Helper in der vierten Iteration gewesen. Die Priorität des PSPS war höher, da dort auch die für das planbasierte Scheduling-Verfahren relevante Programmlogik vorhanden sein würde. Der PSPS-Helper soll ausschließlich den Job annehmen und validieren. Im Gegensatz dazu soll der PSPS den Job erhalten, zur angegebenen Zeit ausführen und bei Bedarf terminieren. Der Vorteil dieser Herangehensweise gegenüber der Ersten wäre, dass das höher priorisierte Plugin zuerst fertiggestellt werden würde.

Ein alternativer Ansatz wäre die Entwicklung der gesamten Programmlogik in einer Iteration. Hierbei wird auf eine zusätzliche Entwurfsphase verzichtet. Ein Vor-

teil dieses Ansatzes wäre, dass durch eine Entwurfsphase weniger Aufwand entsteht als bei zweien. Das resultierende Entwicklungsziel könnte jedoch für die Entwicklung nicht akkurat genug sein. Dies kann zur Folge haben, dass während der Implementierung das Entwicklungsziel weiter ausgearbeitet werden müsste, was wiederum zusätzlichen Aufwand erzeugen würde. Die zusätzliche Flexibilität während der Entwicklung ist allerdings auch ein Vorteil, denn Interfaceänderungen können möglicherweise zu weniger Abänderungen des bestehenden Quellcodes führen.

Es wurde sich für den letzten Ansatz entschieden, bei dem die gesamte Programmlogik in einer Iteration implementiert werden, da sich von diesem Ansatz ein geringerer Zeitaufwand erhofft wurde. Für die Einhaltung der Deadline war ein möglichst geringer Zeitaufwand wichtig, welche der größte Faktor für diese Entscheidung war. Ohne Deadline wäre der Ansatz mit zwei Iterationen, bei dem zunächst die Programmlogik für den Scheduler in der dritten Iteration implementiert werden würde, gewählt worden, da die angenehmere Implementierungsphase durch die Lokalität und einem präziseren Entwurf präferiert werden würde.

Für die dritte Iteration ist demnach die Implementierung der Programmlogik das Entwicklungsziel. Diese besteht aus der Jobannahme, Validierung und Abarbeitung. Bei der Annahme handelt es sich um die Parameterübergabe, welche bereits in der zweiten Iteration für die Testung teilweise implementiert wurde. Die Validierung würde von dem Drittprogramm vorgenommen werden und die entsprechende Anfrage zur Validierung und die Rückgabe müssten von dem PSPS-Helper vorgenommen werden. Bei der Abarbeitung soll der Job zur angegebenen Zeit ausgeführt werden. Hier soll zudem kein Job ausgeführt werden, wenn der Plan es nicht vorsieht. Für eine feinere Teilung der Programmlogik würden Kenntnisse der konkreten Implementierungsmöglichkeiten notwendig sein, welche zu der Zeit der Entwurfserstellung noch nicht vorhanden waren.

3.7 Entwicklung - Programmlogik

Für eine strukturierte Implementierung der Programmlogik war es nötig, diese in Teilkomponenten zu unterteilen. Die Notwendigkeit einer strukturierten Implementierung hat sich aus der manuellen Durchführung von Testfällen ergeben. Die Analyse bei möglichen Fehlern würde sich einfacher gestalten, wenn die vorangegangene Implementierung der Programmlogik lokalisiert stattgefunden hat. Es ergaben sich zwei Möglichkeiten, in welcher Reihenfolge die Teilkomponenten implementiert werden können.

Bei der Implementierung könnte sich an den Pfad, welcher ein Job in dem System durchläuft, gehalten werden. Hierbei würde zunächst die Annahme, anschließend die Validierung und zuletzt die Abarbeitung implementiert werden. Der Vorteil dieser Variante wäre, dass es einfacher wäre, Testfälle durchzuführen, denn diese würde durch einen Durchlauf der Jobs durch das System erfolgen.

Alternativ könnte zunächst die Abarbeitung, anschließend die Validierung und Annahme implementiert werden. Bei dieser Herangehensweise würde die Durchführung von Testfällen mithilfe eines Durchlaufs der Jobs durch das System nicht möglich sein, da die nötige Teilkomponenten für einen Durchlauf fehlen würden. Dies erschien schwieriger, da die nötigen Anfragen und Antworten simuliert werden müssten.

Das PSPS würde die Jobdaten benötigen, welche zum Teil unklar waren.

Aufgrund der vereinfachten Durchführung von Testfällen wurde sich für die Implementierung entlang des Pfades, bei dem der Job das System durchläuft, entschieden.

3.7.1 Jobannahme

Die Jobannahme wurde in der zweiten Iteration teilweise implementiert. Es konnte bereits ein Parameter an den PSPS-Helper übergeben werden, welcher "plan„ hieß. Da ein Plan für einen Job aus Start- und Endzeitpunkt besteht, sollten zwei Parameter anstatt einer übergeben werden. Bei der Erweiterung ergaben sich Schwierigkeiten.

Für die Angabe eines zusätzlichen Parameters war es nötig, eine zweite Methode bereitzustellen, welche bei Verarbeitung der Parameter aufgerufen wird. Es handelt sich hierbei um eine Callback-Methode (dt. Rückrufmethode). Damit sichergestellt wird, dass beide Parameter gesetzt sind, wäre ein Speicher notwendig, auf den die Callback-Methoden zugreifen könnten, welche für die Start- und Endzeitpunkt Parameter registriert wurden. Die Implementierung eines weiteren Parameters würde darauffolgend einen zusätzlichen Aufwand erfordern.

Bei einer Reevaluation wurden weitere Nachteile bei der Übergabe von zwei Parametern erkannt. Bei der Begrenzung auf zwei Zeitpunkte könnten zusätzliche Optionen verhindert werden. Das Pausieren eines Jobs zu bestimmter Zeit wäre nicht möglich. Durch die Angabe eines generischen "plan„-Parameters könnten alle Optionen ermöglicht werden, denn das Format eines Plans würde dadurch von dem Drittprogramm bestimmt werden. Diese noch unbekannten Optionen müssten jedoch auch in der Jobabarbeitung unterstützt werden, weshalb bereits zu diesem Zeitpunkt deutlich wurde, dass dem PSPS Kompetenzen zum eigenständigen Scheduling-Verfahren entzogen und an das Drittprogramm delegiert werden müssten.

Die erhöhte Flexibilität und der geringere Aufwand führten zur Beibehaltung der Jobannahme mit einem Parameter.

3.7.2 Planvalidierung

Bei der Validierung soll sichergestellt werden, dass die Jobausführung zu dem angegebenen Plan möglich ist. Dies wurde durch Erweiterung einer Testanfrage von dem PSPS-Helper an das Drittprogramm aus der zweiten Iteration mit Angabe des "plan„-Parameters ermöglicht. Zudem wurde das Dummy-Drittprogramm insofern erweitert, als dass es auf die Anfrage eine Antwort senden konnte.

Nach dem gewählten Subscribe-Publish Modell, sollte nun der Job per Publish-Ereignis an das zuständige Cluster übertragen werden, weshalb alle nötigen Job-Informationen erfasst und übermittelt werden müssten. Für die Erfassung von Job-Informationen war keine Methode bekannt und nach einer Recherche konnte festgestellt werden, dass die dafür verfügbare Methode in SPANK-Plugins (spank_get_item) nicht in dem Allokatorkontext nutzbar ist. Bis zu diesem Zeitpunkt waren die verschiedenen Kontexte, in denen SPANK Plugins ausgeführt werden können, noch nicht erkannt worden. Die Ausführung der SPANK-Plugins findet bei *sbatch* im Allokatorkontext statt, weshalb das erfassen von Job-Informationen nicht möglich ist.

Da keine Job-Informationen erfasst werden konnten, war es nicht möglich, die Jobs auf Clustern auszuführen, welche die Daten nicht von *sbatch* empfangen. Es wurde sich dafür entschieden, statt dem Subscribe-Publish Modells das Client-Server Modell zu verwenden, da die Transmission von Job-Informationen per Publish-Ereignis unmöglich wurde.

3.7.3 Jobübergabe

Die Interfaceänderung hatte zur Folge, dass eine Methode zur Übertragung von nötigen Informationen an den PSPS gefunden werden musste, denn die Jobs müssten zu der korrekten Zeit ausgeführt werden. Durch *sbatch* werden Jobs nach erfolgreicher Eingabe automatisch an den slurmctld weitergeleitet. Allerdings war unklar, ob auf den “plan„-Parameter im PSPS zugegriffen werden kann.

Zu diesem Zeitpunkt war aufgefallen, dass Planänderungen möglich sein könnten. Für die Aktualisierung des Plans gab es mehrere Möglichkeiten. Zum Einen könnte der neue Plan über SLURM angenommen werden, wobei die konkreten Implementierungsmöglichkeiten noch nicht feststanden. Eine Alternative wäre die Delegation dieser Aufgabe an das Drittprogramm gewesen. Hierbei würde das Drittprogramm den aktualisierten Plan an den Scheduler übermitteln müssen. Jedoch müsste jeder Job identifizierbar sein, damit ein neuer Plan zugeordnet werden könnte und auf die Job-ID, welche eine Job-Information ist, kann in dem Allokatorkontext in SPANK-Plugins nicht zugegriffen werden. Dies könnte durch die Zuordnung eines eigenen einzigartigen Identifikators geschehen.

Es wurde sich für die Delegation an das Drittprogramm entschieden, da ein geringerer Aufwand erwartet wurde. Der einzigartige Identifikator könnte von dem Drittprogramm bei der Validierung generiert und übermittelt werden. Bei weiterer Recherche der verfügbaren Methoden in SPANK Plugins wurde eine Methode zur Setzung von Umgebungsvariablen eines Jobs gefunden (spank_setenv). Diese konnten zur Übertragung des Identifikators an den PSPS genutzt werden.

3.7.4 Jobabarbeitung

Die Abarbeitung der Jobs findet bei dem Scheduler statt. Bei der Durchführung von Testfällen für die Jobübergabe wurde erkennbar, dass die Jobs direkt nach der Übergabe abgearbeitet wurden. Es wurde infrage gestellt, ob der PSPS aktiv sei. Zur Testung wurde ein Herzschlag hinzugefügt, welcher einen Text ausgibt. Bei erneuter Durchführung der Testfälle konnte erfahren werden, dass der PSPS aktiv war. Aus den Log-Dateien wurde ersichtlich, dass SLURM den Job in die Warteschlange eingefügt hatte. Anschließend wurde ein Scheduler-Durchlauf gestartet, welcher den Job ausgeführt hatte. Nach der entsprechenden Ausgabe aus der Log-Datei wurde in dem Quellcode von SLURM gesucht und in der SLURM-internen “schedule„-Funktion gefunden.

Ein SLURM Scheduler Plugin war somit für die Sortierung der Warteschlange beziehungsweise die Erweiterung des Scheduling-Verfahren zuständig und nicht, wie vorher angenommen, für das Scheduling-Verfahren an sich. Es handelt sich um einen geschichteten Scheduler. Daraus ergaben sich neue Schwierigkeiten, denn die Jobs müssten an der Ausführung gehindert werden, damit sie zum gegebenen Zeitpunkt

gestartet werden könnten. Bei der Inspektion des Quellcodes von SLURM ist aufgefallen, dass Jobs mit Priorität 0 in den Halten-Status versetzt wurden. Durch Codeanalyse konnte herausgefunden werden, dass Jobs mit diesem Status nicht durch die "schedule„-Funktion gestartet werden.

Für die initiale Setzung der Priorität gab es eine Methode von SLURM Scheduler Plugins, welche von SLURM aufgerufen wird (slurm_sched_p_initial_priorit). Nach Überarbeitung und Durchführung von Testfällen für diese Methode war durch die Log-Dateien erkenntlich, dass die Jobs nicht mehr automatisch gestartet wurden.

Bei Implementierung der Jobübergabe wurde die Planänderung durch Festlegung einer Identifikation ermöglicht. Es ergaben sich mehrere Möglichkeit, eine Planänderung im Scheduling-Verfahren zuzulassen. Nach erfolgreicher Validierung könnte der Plan dem PSPS durch das Drittprogramm mitgeteilt werden. Analog könnten Planänderungen übermittelt werden. Alternativ könnte das Interface insofern abgeändert werden, dass durch Anfragen des Drittprogramms Jobs gestartet und terminiert werden könnten. Dieser Ansatz würde eine größere Flexibilität ermöglichen, da die Interpretation des Plans durch das Drittprogramm geschehen würde.

Es wurde sich für die Alternative entschieden, da sich bereits in der Jobannahme für eine Formatbestimmung des Plans von dem Drittprogramm entschieden wurde. Das Interface wurde dementsprechend erweitert, sodass mit dem Anfragentyp "action„ dem PSPS Instruktionen übermittelt werden konnten. Hierbei war es wichtig, den Identifikator für den Job zu übermitteln, damit dieser von dem PSPS gefunden werden kann.

Ziel war es anschließend, dass ein übermittelter Job durch eine Anfrage gestartet werden kann. Daraufhin wurde der PSPS insofern angepasst, als dass er bei jedem Herzschlag überprüft, ob eine Anfrage empfangen werden konnte. Diese wurden dann auf ihren Typ überprüft. Bei einer Instruktion würde der Job mit dem entsprechenden Identifikator gesucht und die Instruktion ausgeführt werden.

Bei der Implementierung einer Methode zur Suche von einem Job ergaben sich Schwierigkeiten. Über eine Methode, welche auch Alexandros Gialidis in seinem SLURM Scheduler Plugin genutzt hatte (build_job_queue) konnten nur Jobs mit dem Anstehend-Status abgefragt werden. Weitere Methoden zur Abfrage einer Job-Liste konnten nicht gefunden werden, weshalb auf die globale "job_list„ Variable zugegriffen werden musste, aus der auch die "build_job_queue„-Methode die Daten entnommen hat. Über eine Suchfunktion für Listen, welche von SLURM bereitgestellt wurde, fand die Suche nach einem Job mit dem Indikator, welcher in der Instruktion angegeben war, statt. Bei Testläufen konnte festgestellt werden, dass auf nahezu alle Daten nicht zugegriffen werden konnte. Der PSPS hatte invalide Daten gelesen oder war abgestürzt. Es wurde infrage gestellt, ob diese Daten für Jobs mit dem Halten-Status zugreifbar sein würden. Bei weiteren Testläufen konnte festgestellt werden, dass auch nicht auf den Status der Jobs zugegriffen werden konnte, weshalb von invaliden Zeigern auszugehen war. Es wurde anschließend versucht, auf ähnliche Art und Weise wie in der "build_job_queue„-Methode, über die "job_list„ zu iterieren. Die Jobdaten konnten anschließend erfolgreich ausgelesen und auf die Umgebungsvariablen zugegriffen werden.

Es wurden drei Instruktionen implementiert. Eine Möglichkeit zum starten von Jobs wurde durch eine und das Beenden von Jobs durch eine andere Instruktion

ermöglicht. Zur Statusabfrage wurde eine weitere Instruktion hinzugefügt, welche eine periodische Überprüfung des Job-Status durch das Drittprogramm ermöglichen sollte.

3.8 Resultate

Im Rahmen dieser Arbeit entstanden zwei Plugins für SLURM. Zum einen der PSPS-Helper und zum anderen der PSPS. Des Weiteren entstand ein Interface.

3.8.1 Plugins

Der PSPS-Helper wird im Allokatorkontext genutzt und empfängt einen Plan über den “plan„-Parameter im Job Skript, welches über *sbatch* an SLURM übergeben wird. Dieser Plan wird an das Drittprogramm übermittelt und dort validiert. Das Plugin wartet auf die Antwort vom Drittprogramm und gibt eine positive Rückmeldung bei einem Erfolg. Bei Misserfolg wird der Prozess abgebrochen und das Job Skript wird nicht weiter bearbeitet. Bei der positiven Rückmeldung sendet das Drittprogramm zudem einen einzigartigen Identifikator für diesen Job mit. Dieser wird vom PSPS-Helper als eine Umgebungsvariable für den Job gesetzt und vom PSPS zur Identifikation von planungsbasierten Jobs genutzt. Der PSPS startet einen eigenen Thread bei der Initialisierung. Dieser Thread überprüft nach einem festgelegten Intervall die Verbindung zum Drittprogramm auf neue Instruktionen und sendet einen Herzschlag. Bei einer neuen Instruktion wird diese ausgeführt. Das Drittprogramm hat damit die Kontrolle darüber, wann Jobs ausgeführt und gestoppt werden, was das planungsbasierte Scheduling-Verfahren ermöglicht. Es kann zudem der Status von bestimmten Jobs abgefragt werden.

3.8.2 Interface

Für die Kommunikation zwischen den Plugins und dem Drittprogramm wurde ein Interface entwickelt, welches das JSON-Format nutzt. Über einen Schlüssel, welcher “request„ genannt wurde, kann der Typ einer Anfrage festgelegt werden. Es existieren drei verschiedene Anfragetypen: status, action und plan. Eine Statusantwort wird zur Übermittlung von Statusinformationen genutzt. Der Anfragetyp “action„ wird für die Übergabe der Instruktionen von dem Drittprogramm an das PSPS genutzt. Die Plananfrage wird zur Validierung des Jobs verwendet und wird von dem PSPS-Helper an das Drittprogramm versendet. Folgende Schlüssel werden bei den Anfragen benötigt:

Tabelle 1: Schlüsselnutzung der Anfragen und Antworten

Schlüssel	Beschreibung	Datentyp	status	action	plan
status	Statuscode (Tabelle 2)	int	✓	✗	✗
action	Instruktion (Tabelle 3)	int	✗	✓	✗
plan	Planparameter von PSPS-Helper	string	✗	✗	✓
JobID	Identifikator eines Jobs	int64	✓	✓	✗

Tabelle 2: Statuscodes

Interne Darstellung (enum)	Statuscode	Beschreibung
STATUS_VALID	0	Antwort bei validem Plan
STATUS_INVALID	1	Antwort bei invalidem Plan
STATUS_ALIVE	2	Herzschlag des PSPS
STATUS_JOB_PENDING	3	Jobausführung steht aus
STATUS_JOB_RUNNING	4	Job wird ausgeführt
STATUS_JOB_CANCELLED	5	Job wurde abgebrochen
STATUS_JOB_COMPLETE	6	Jobausführung war erfolgreich
STATUS_JOB_UNDEFINED	7	Nicht abstrahierter interner Jobstatus

Tabelle 3: Instruktionen

Interne Darstellung (enum)	Statuscode	Beschreibung
ACTION_RUN_JOB	0	Starte Job
ACTION_ABORT_JOB	1	Breche den Job ab

4 Konklusion

Im Rahmen dieser Arbeit wurde ein Prototyp für die Anbindung des lokalen Management System SLURM an ein virtuellen Ressourcen Manager entwickelt. Der Prototyp enthält zwei Plugins, welche über ein vorgeschlagenes Interface an ein VRM angebunden werden können. Es wurde eine Möglichkeit zur Eingabe eines Ausführungsplans für Jobs ermöglicht, welche in Batch-Skripten für *sbatch* angegeben werden können. Die Ausführungsgarantie zum angegebenen Startzeitpunkt von Jobs wurde durch Abänderung des Scheduling-Verfahren ermöglicht, sodass durch Delegation dies von dem VRM durchgeführt werden kann. Hierbei kann SLURM über das vorgeschlagene Interface Scheduling-Instruktionen übermittelt werden. Durch den im Rahmen dieser Arbeit entwickelten Prototyp eines SLURM Scheduler Plugins, werden ausschließlich diese Instruktionen ausgeführt.

Infolgedessen wurde ein planbasiertes Scheduling-Verfahren für SLURM ermöglicht, da ein VRM durch Übermittlung von Instruktionen einen planungsbasierten Ablauf ermöglichen kann. Die Aufgabenstellung wurde somit erfüllt - sie erwies sich als umsetzbar.

Für die Entwicklung der Plugins wurde ein iterativer Entwicklungsprozess gewählt. Die größten Entwicklungsverzögerungen wurden durch manuelle Durchführung von Testfällen und das Debuggen generiert. Hierbei war die Lokalisierung von Fehlern durch Ausgaben in eine Log-Datei nicht immer hilfreich, wodurch viel ausprobiert werden musste. Besonders hervorzuheben sind hierbei die Schwierigkeiten, welche in der Implementierung der Jobabarbeitung entstanden waren. Für den Zugriff auf Jobs in der Job-Liste konnte der Fehler erst nach unerwartet langer Zeit gefunden werden. Nach einer Selbstreflexion wurde erkannt, dass automatisierte Tests genutzt hätten werden sollen.

Es ist zu beachten, dass es sich bei dem Resultat um einen Prototyp handelt, welcher noch nicht ausgereift ist. Die Kommunikation zwischen dem in der Entwicklung

genutzten Dummy-Drittprogramm und den Plugins verläuft ohne Verschlüsselung, welche für sichere Kommunikation notwendig sein würde. Des Weiteren erfordert das Interface eine Erweiterung um die Angabe eines Clusters für die Ausführungsplanübergabe. Dies ist notwendig, wenn der VRM mehrere Cluster verwaltet. Der VRM würde Kenntnisse davon haben müssen, an welchen Cluster Instruktionen zu dem übergebenen Ausführungsplan versendet werden müssen.

Literaturverzeichnis

[1] L. . Burchard, M. Hovestadt, O. Kao, A. Keller, and B. Linnert. The virtual resource manager: an architecture for sla-aware resource management. In *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.*, pages 126–133, 2004.

[2] Docker kommandos. https://docs.docker.com/engine/reference/commandline/docker/. Eingesehen am 03.01.2021.

[3] Understanding the head node. https://zhanglab.ccmb.med.umich.edu/docs/node9.html. Eingesehen am 03.01.2021.

[4] Introducing json. https://www.json.org/json-en.html. Eingesehen am 03.01.2021.

[5] json-c. https://github.com/json-c/json-c. Eingesehen am 03.01.2021.

[6] json-c beispiele. https://gist.github.com/alan-mushi/19546a0e2c6bd4e059fd. Eingesehen am 03.01.2021.

[7] bind(2) — linux manual page. https://man7.org/linux/man-pages/man2/bind.2.html. Eingesehen am 03.01.2021.

[8] Slurm® commercial support and development. https://www.schedmd.com/index.php. Eingesehen am 03.01.2021.

[9] Slurm docker cluster. https://github.com/giovtorres/slurm-docker-cluster. Eingesehen am 03.01.2021.

[10] Slurm programmer's guide. https://slurm.schedmd.com/programmer_guide.html. Eingesehen am 03.01.2021.

[11] sacct. https://slurm.schedmd.com/sacct.html. Eingesehen am 03.01.2021.

[12] salloc. https://slurm.schedmd.com/salloc.html. Eingesehen am 03.01.2021.

[13] sbatch. https://slurm.schedmd.com/sbatch.html. Eingesehen am 03.01.2021.

[14] Spank. https://slurm.schedmd.com/spank.html. Eingesehen am 03.01.2021.

[15] Slurm spank gpu compute mode plugin. https://github.com/stanford-rc/slurm-spank-gpu_cmode. Eingesehen am 03.01.2021.

[16] Thesis scheduler plugin von alexandros gialidis. https://github.com/alexgialidis/slurm-scheduler-plugin. Eingesehen am 03.01.2021.

[17] Rechnerverbund. https://de.wikipedia.org/wiki/Rechnerverbund. Eingesehen am 03.01.2021.

[18] Slurm overview. https://slurm.schedmd.com/overview.html. Eingesehen am 03.01.2021.

A Anhang

Die Resultate können hier eingesehen werden https://git.imp.fu-berlin.de/becker29/planungsbasierter-slurm.