Projektbericht

Marina Shvalova und Oliver Bestmann

15. Oktober 2012

Bericht zum Projekt Parallelrechnerevaluation

PySpank: Python Interface zu SPANK

Betreuer: Julian Kunkel

Wissenschaftliches Rechnen
Department Informatik
Universität Hamburg

Inhaltsverzeichnis

1 Einleitung

Dieser Projektbericht ist im Kontext des *Parallelrechnerevaluation*-Projekts im Arbeitsbereich *Wissenschaftliches Rechnen* an der Universität Hamburg entstanden.

1.1 Motivation und Zielsetzung

Im Rahmen dieser Arbeit wird ein Python Interface *PySPANK* zu einer Plug-in Schnittstelle *SPANK* entwickelt. *PySPANK* stellt eine Erweiterung für den das unixbasierte *Ressourcen Management Tool – SLURM –* bereit. *SLURM* hat sich durch seine Einfachheit, Fehlertoleranz und Sicherheit zu einem der populären Werkzeuge für Supercomputer etabliert. Es kann durch verschiedene Plug-ins erweitert werden. *SPANK* ist eine von *SLURM* bereitgestellte Plug-in-Schnittstelle. Er stellt ein generisches Interface für Plug-ins, die den Startvorgang von Jobs beeinflussen können. Auf Basis von *SPANK* wurde während dieses Projekts die Plug-in-Schnittstelle *PySPANK* entwickelt.

1.2 Aufbau der Arbeit

In dieser Arbeit werden zuerst die technischen Grundlagen und die Architektur von *SLURM* erläutert. Es wird ein Überblick über einige Eigenschaften und Erweiterungen für *SLURM* gegeben und einige der verfügbaren Plug-ins genauer beschrieben. Besondere Aufmerksamkeit wird dem Plug-in *SPANK* gewidmet. Nach diesem theoretischen Teil folgt die Idee und Motivation, die unserer Erweiterung *PySPANK* zugrunde liegt. Es wird die Wahl der Programmiersprache Python begründet. Danach wird die Architektur unseres Plug-ins erklärt und auf die drei vorhandenen Schichten eingegangen. Abschließend folgt ein Beispiel, wie ein fertig entwickeltes *real-life* Plug-in in Python aussieht.

Als Letztes folgt ein kurzes Fazit, das unsere Ergebnisse abschließend zusammenfasst und einen kleinen Ausblick gibt.

2 SLURM

Simple Linux Utility for Resource Management (*SLURM*) ist ein einfacher Ressourcen Manager für große und kleine unixbasierte Cluster. *SLURM* hat drei Kernfunktionen [2]:

- Allokieren von Ressourcen wie Rechenleistung und Speicher auf Rechnerknoten für einen User und einen bestimmten Zeitraum
- Bereitstellen eines Frameworks zum Starten, Ausführen und Überwachen der laufenden Jobs auf den reservierten Knoten
- Verteilen von Jobs aus einer Warteschlange auf die Ressourcen

SLURM ist im Vergleich zu den anderen Ressourcen Management Tools wie Portable Batch System (PBS) oder Tera-scale Open Source Resource and Queue Manager (TORQUE), ein relativ junges und frei verfügbares Tool. Er ist skalierbar und besitzt hohe Fehlertoleranz, ist sehr modular gebaut und kann mit vielen Plug-ins erweitert werden. *SLURM* wird auf vielen mächtigen High-Performance-Clustern (HPC) der Welt benutzt, so z. B. auf Dawn [1] am *Lawrence Livermore National Laboratory*, Tianhe-1A [2] in China und auf Tera-100 [3] in Europa. Die Installation und Konfiguration von *SLURM* auf einem kleinen System ist mit wenig Aufwand verbunden und kann so in wenigen Minuten durchgeführt werden.

Bei der Konfiguration von *SLURM* können verschiedene Parameter eingestellt werden. Viele der Parameter können die default-Einstellungen besitzen. Die Konfiguration wird in der Datei `/etc/slurm-llnl/slurm.conf` abgespeichert und muss auf jedem Knoten des Clusters konsistent durchgeführt werden. Folgende Parameter können z. B. konfiguriert werden: die Namen der Knoten und IP-Adressen, die Anzahl der Partitionen auf dem Knoten und die Scheduling Strategien. Für das Vornehmen der Konfigurationseinstellungen gibt es auch ein webbasiertes Tool, das für das Durchführen der komplexeren Einstellungen von Vorteil ist. In Listing 1 sind die Konfigurationseinträge der Knoten- und Partitionseinstellungen zu sehen.

---

[1] Dawn is a BlueGene/P system at LLNL with 147,456 PowerPC 450 cores with a peak performance of 0.5 Petaflops [2].

[2] Tianhe-1A designed by The National University of Defence Technology (NUDT) in China with 14,336 Intel CPUs and 7,168 NVDIA Tesla M2050 GPUs, with a peak performance of 2.507 Petaflops [2].

[3] Tera 100 at CEA with 140,000 Intel Xeon 7500 processing cores, 300TB of central memory and a theoretical computing power of 1.25 Petaflops. Europe's most powerful supercomputer [2].

```
#
# Node Configurations
#
NodeName=DEFAULT CPUs=2 RealMemory=2000 TmpDisk=64000 State=UNKNOWN
NodeName=mcr[0-1151] NodeAddr=emcr[0-1151]
#
# Partition Configurations
#
PartitionName=DEFAULT State=UP
PartitionName=pdebug Nodes=mcr[0-191] MaxTime=30 MaxNodes=32 Default=YES
PartitionName=pbatch Nodes=mcr[192-1151]
```

Listing 1: Ein Ausschnitt der Einstellungen für Knoten und Partitionen aus der `slurm.conf`-Datei

SLURM wird normalerweise von den Benutzern über ein Terminal bedient. Es verfügt jedoch über ein Grafisches User Interface (GUI) für die Administratoren und einfache Nutzer. Durch den Befehl `sview` kann die GUI gestartet werden, durch die laufende Jobs oder Jobs in der Warteschlange überwacht und gesteuert werden können. Nach Eingabe des Administrator-Kennworts kann der Benutzer die Knoten verwalten, aktivieren und deaktivieren und in Partitionen aufteilen.

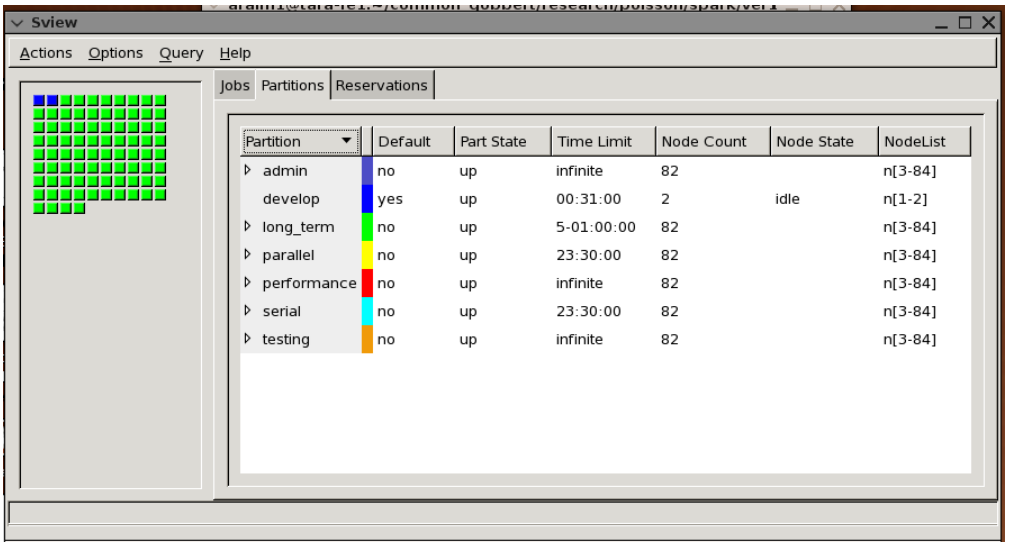

Abbildung 1: Screenshot von `sview`. Links sind die vielen aktiven Knoten und rechts mehrere Partitionen zu sehen.

2.1 Architektur von *SLURM*

SLURM basiert im wesentlichen auf zwei Daemons, `slurmctld` und `slurmd` [3].

`slurmctld` ist der *Controller Master Daemon* und kümmert sich um die systemweite Verwaltung, Annahme und Verteilung von Jobs, persistieren des aktuellen Systemzustandes und ähnliche Aufgaben. Er läuft auf einem zentralen Masterknoten und ist von jedem anderen Knoten im System erreichbar. Ist Ausfallsicherheit gewünscht, kann es Backup-Instanzen des `slurmctld` geben.

`slurmctld` basiert auf drei Bestandteilen:

Job Manager verwaltet die wartenden Jobs in der Queue

Node Manager kontrolliert die Zustandsinformationen des Knotens

Partition Manager gruppiert die Knoten zu Partitionen mit verschiedenen Konfigurationsparametern und allokiert die Knoten

`slurmd` läuft auf jedem Knoten des Systems und stellt eine Möglichkeit für den `slurmctld` bereit, auf das entsprechende System des Knotens zuzugreifen um dort einen Job zu starten. Es ähnelt also einem *Remote-Shell Daemon* wie *SSH*. Weiterhin teilt der Daemon dem `slurmctld` auf dem Masterknoten die Konfiguration des Knotens – CPUs und Speicher – mit.

In der Abbildung 2 ist die *SLURM*-Architektur grafisch dargestellt. Die Grafik zeigt den Aufbau der Architektur und die Interaktion zwischen den einzelnen Komponenten.

Weiterhin verfügt *SLURM* über zusätzliche Nutzer-Tools, wie `srun`, `scontol`, `sinfo`, `squeue`, `scancel`, `sacct` und mehr. Diese Tools können überall auf den Clustern laufen und sind für folgende Zwecke da:

`sbatch` reiht einen vom Benutzer gewünschten Job in die Warteschlange des Systems ein.

`srun` allokiert die Ressourcen, startet interaktive Jobs und initialisiert die parallelen Aufgaben.

`scontrol` kann von den priviligierten Usern zum Ausführen von priviligierten Aufgaben benutzt werden, wie zum Beispiel das Herunterfahren von Partitionen oder Knoten.

`sinfo` gibt zusammengefasste Informationen über die Partitionen und Knoten. Diese Informationen können auch gefiltert werden.

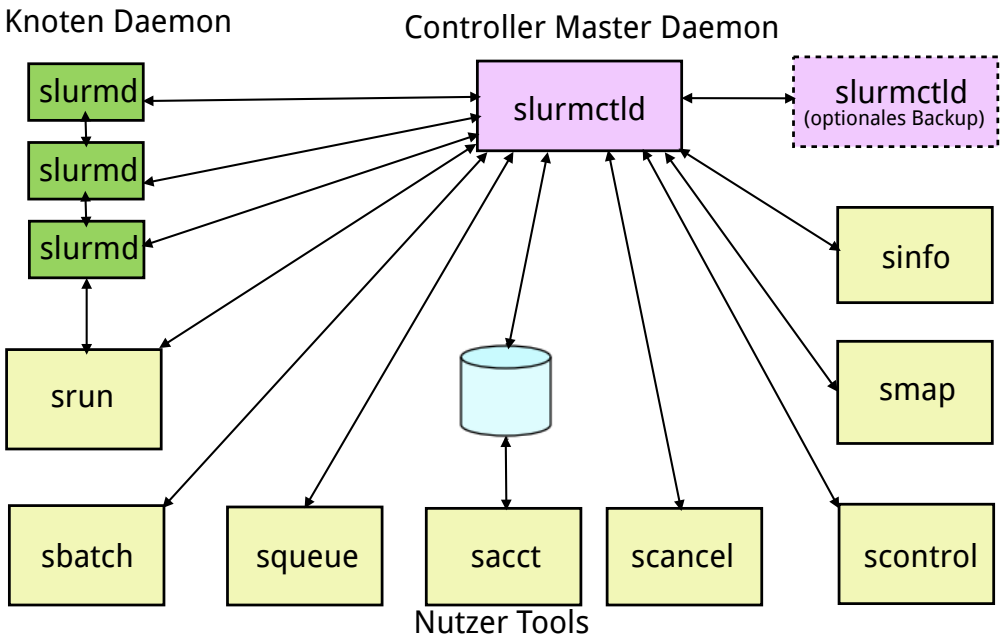

Abbildung 2: Die Architektur von *SLURM*.

`squeue` zeigt den Status aller Partitionen und Knoten an.

`scancel` beendet laufende oder wartende Jobs.

`sacct` zeigt die ausführliche Information zu den durchgeführten Jobs und Jobsteps auf dem Knoten.

`smap` zeigt den Status der Jobs, Partitionen und der Knoten grafisch an.

In Listing 2 abgedruckt ist ein Beispiel, wie `srun` und `sbatch` zusammen verwendet werden können. Die Datei `script.sh` ist ein Shell-Script im Ordner `/home/user`. Wird es ausgeführt, gibt es zuerst den aktuellen Hostname aus und startet dann über srun `/bin/hostname` und `/bin/pwd` auf allen allokierten Ressourcen. Das Skript wird mit `sbatch` an den Master-Knoten mit dem Wunsch, es auf vier CPUs auszuführen, übergeben. Der `slurmctld` teilt dem Job dann je zwei CPUs auf den Knoten `node9` und `node10` zu und führt das Skript dann auf dem ersten Knoten einmal aus. Das Skript startet dann über `srun` weitere Befehle, die auf allen allokierten CPUs ausgeführt werden. So wird `/bin/hostname` zweimal auf `node9` und zweimal auf `node10` ausgeführt. `srun` wechselt vor dem Start des übergebenen Befehls in das gleiche Verzeichnis, in dem `srun` bzw. `sbatch` ausgeführt wurde.

```
node0:~$ cat script.sh
#!/bin/sh
#SBATCH --time=1
/bin/hostname
srun -l /bin/hostname
srun -l /bin/pwd

node0:~$ sbatch -n4 -o script.stdout script.sh
sbatch: Submitted batch job 469

node0:~$ cat script.stdout
node9
0: node9
1: node9
2: node10
3: node10
0: /home/user
1: /home/user
2: /home/user
3: /home/user
```

Listing 2: Ein Beispiel zur Verwendung von `sbatch` und `srun`

2.2 Plug-ins

Wie schon erwähnt verfügt *SLURM* über eine Plug-in Schnittstelle und somit kann die Funktionalität von *SLURM* einfach erweitert werden. Es existiert bereits eine Reihe von fertigen Plug-ins. Da *SLURM* sehr flexibel und modular aufgebaut ist, können fast alle Komponenten, wie zum Beispiel das Authentifikationsmodul oder der Scheduler, ausgetauscht werden. Im folgenden werden einige dieser Plug-ins vorgestellt:

pam-slurm Pluggable Authentication Module (PAM) stellt eine Programmierschnittstelle für Authentisierungsdienste dar. Das Plug-in beschränkt den Zugang zu den Knoten, die ein Job zur Laufzeit zugeteilt bekommt. So kann ein User sich nicht unerlaubt auf allen Knoten einloggen und die zugeteilte Rechenleistung der anderen User verbrauchen.

I/O Watchdog ist ein Monitoringtool um abgestürzte oder aufgehängte Jobs abzubrechen. Dafür überwacht der Watchdog die Applikationen auf Schreibzugriffe. Wenn eine Applikation eine längere Zeit keine Schreibzugriffe getätigt hat, wird dies als ein Zeichen dafür angesehen, dass sie sich aufgehängt hat. Sie sollte dafür natürlich

regelmäßig Schreibzugriffe ausführen, wenn sie normal funktioniert, damit sie nicht fehlerhaft als abgestürzt erkannt und beendet wird.

Sqlog *SLURM Query Log*-Tool, mit dem unter anderem die Log-Ausgaben von *SLURM* in eine Datenbank gespeichert werden. Weiter wird protokolliert, wer welche Jobs wann, wie, wo und mit welchem Ergebnis ausführen durfte. An diese Datenbank kann dann mit `sqlog` eine Anfrage gestellt werden. Beispielsweise wird mit dem folgendem Query abgefragt, welche Jobs fehlerhaft zwischen acht und neun Uhr morgens beendet wurden, absteigend sortiert nach Abbruchzeitpunkt.

```
sqlog --all --end=8am..9am --states=F --sort=-end
```

PYSLURM Ein Python-Erweiterungsmodul, um auf die interne API von *SLURM* aus Python heraus zuzugreifen. Dadurch hat man Zugriff auf die Jobs, kann diese auch selbst schedulen, abbrechen oder ähnliche administrative Tätigkeiten durchführen. Auch bekommt man Zugriff auf die Knoten und kann diese verwalten.

***SPANK* Plug-ins** SPANK Plug-ins sind eine eigene Kategorie von Plug-ins, die den Startvorgang von Prozessen durch `srun` und andere Tools beeinflussen können. Sie bieten bestimmte Einstiegspunkte und es werden mehrere dieser Plug-ins nacheinander ausgeführt [4]. Beispiele für SPANK Plug-ins ändern z. B. das *nice*-Level der gestarteten Prozesse oder setzen bestimmte Umgebungsvariablen. Auf SPANK wird im nächsten Kapitel genau eingegangen.

2.3 SPANK

Die Abkürzung *SPANK* steht für *SLURM Plug-in Architecture for Node and job (K)control* und bezeichnet ein generisches Interface für Plug-ins, die zum Kontrollieren und Steuern des Startvorgangs von Jobs in *SLURM* benutzt werden können. Im Gegensatz zu normalen *SLURM*-Plug-ins sind diese besonders lose an *SLURM* gekoppelt, da sie beim Kompiliervorgang nicht gegen dessen Bibliotheken gelinkt werden müssen.

Um ein *SPANK*-Plug-in zu schreiben, reicht es aus, die *C*-Header-Datei `spank.h` einzubinden. Dieser definiert Prototypen für einige Funktionen, die ein Plug-in implementieren muss. Da das Plug-in selbst nicht von *SLURM* abhängig ist, ist es folglich nicht möglich, Funktionen aus der *SLURM*-API zu verwenden.

```
/*  Prototype for all spank plug-in operations
 */
typedef int (spank_f) (spank_t spank, int ac, char *argv[]);
extern spank_f slurm_spank_init;
extern spank_f slurm_spank_init_post_opt;
extern spank_f slurm_spank_exit;
```

Listing 3: Ausschnitt aus der Datei `spank.h`.

In Listing 3 sind drei Prototypen von Funktionen definiert, die von jedem *SPANK*-Plug-in implementiert werden müssen. Die Funktion `slurm_spank_init` wird immer zu Beginn aufgerufen, wenn das Plug-in geladen wurde. Sie kann mit der Funktion `spank_option_register` Kommandozeilen-Parameter für `srun` und `sbatch` registrieren. Wird ein Parameter registriert, übergibt das Plug-in eine *Callback*-Funktion, die später mit dem vom Benutzer gesetzten Wert für den Parameter von *SLURM* aufgerufen wird. Diese kann die Eingabe dann validieren und einen Erfolgs- bzw. Fehlercode zurückgeben.

Wurden alle übergebenen Parameter verarbeitet, wird die Funktion `slurm_spank_init_post_opt` ausgeführt. Abhängig vom Kontext, in dem das Plug-in geladen ist, werden nun neben weiteren Funktionen zum Ende der Ausführung die Funktion `slurm_spank_exit` von *SLURM* aufgerufen.

Ein entsprechendes Plug-in wird in einem von drei Kontexten ausgeführt.

allocate ist als Kontext gesetzt, wenn das Plug-in z. B. in `sbatch` gestartet wird. Es werden nur die drei oben genannten Funktionen `init`, `init_post_opt` und `exit` aufgerufen.

local ist der Kontext, wenn das Plug-in in `srun` ausgeführt wird. Es wird neben den Funktionen, die im `allocate`-Kontext ausgeführt werden, auch die Funktion `local_user_init` nach `init_post_opt` aufgerufen.

remote In diesem Kontext befindet sich ein Plug-in, wenn es als Teil des `slurmd` ausgeführt wird. Dies ist beispielsweise der Fall, wenn ein von einem anderen Knoten initiiertes `srun` einen Prozess auf einer oder mehreren CPUs des aktuellen Knoten ausführen möchte. Ein Plug-in in diesem Kontext wird an mehreren Stellen beim Starten der Prozesse aufgerufen und kann so aktiv in den Vorgang eingreifen. Siehe Listing 4 für einen Auszug aus der Datei `spank.h`, die die Ausführungsreihenfolge in diesem Kontext beschreibt.

```
/*  Plug-in callbacks are completed at the following points in slurmd:
 *
 *   slurmd -> slurmstepd
 *               `-> init ()
 *                -> process spank options
 *                -> init_post_opt ()
 *               + drop privileges (initgroups(), seteuid(), chdir())
 *               `-> user_init ()
 *               + for each task
 *               |       + fork ()
 *               |       |
 *               |       + reclaim privileges
 *               |       `-> task_init_privileged ()
 *               |       |
 *               |       + become_user ()
 *               |       `-> task_init ()
 *               |       |
 *               |       + execve ()
 *               |
 *               + reclaim privileges
 *               + for each task
 *               |     `-> task_post_fork ()
 *               |
 *               + for each task
 *               |       + wait ()
 *               |          `-> task_exit ()
 *               `-> exit ()
 */
```

Listing 4: Beschreibung der Ausführungsreihenfolge der Plug-in-Funktionen im `remote`-Kontext

3 PySPANK

In diesem Kapitel beschreiben wir die Entwicklung und Funktionsweise von *PySPANK*. *PySPANK* bietet die Möglichkeit, *SPANK*-Plug-ins in der beliebten Skriptsprache Python zu implementieren.

3.1 Warum Python?

SPANK-Plug-ins müssen im Normalfall in der Sprache *C* geschrieben werden. Dies ist nötig, da das Plug-in als *Shared-Library* kompiliert wird und bestimmte Funktionen als Einstiegspunkte exportieren muss. Aus diesem Grund ist es nicht möglich, ein Plug-in in einer anderen Sprache zu schreiben (mit Ausnahme von *C++* oder einer anderen maschinennahen Programmiersprache).

Die Aufgaben, die ein *SPANK*-Plug-in erledigt, sind jedoch häufig sehr trivial. Beispielsweise setzt ein Plug-in das *nice*-Level eines gestarteten Prozesses auf einen vom User übergebenen Wert, verändert Umgebungsvariablen oder beschränkt die CPUs, die ein Task nutzen kann, auf die für den User allokierte Anzahl. Obwohl diese Aufgaben sehr einfach gehalten sind, muss in *C* oft viel Code geschrieben werden, da die Sprache sehr maschinennah ist. So muss der Programmierer sich um das Speichermanagement selbst kümmern, was gerade beim Verarbeiten und Zusammensetzen von Zeichenketten aufwendig und fehleranfällig ist. Weiterhin ist es einfach in *C* das Programm durch einen fehlerhaften Speicherzugriff zum Absturz zu bringen. Für ein *SPANK*-Plug-in, das als Teil des `slurmd` ausgeführt wird, bedeutet das, dass der Daemon mit einem Fehler abstürzt.

Unter anderem aus diesen Gründen haben wir uns entschlossen eine Möglichkeit zu schaffen, *SPANK*-Plug-ins in Python programmieren zu können. Bei einem in Python programmierten *SPANK*-Plug-in wird der notwendige Quellcode um einen großen Teil reduziert. Weiterhin wird dem Benutzer die manuelle Speicherverwaltung entnommen und an den Python-Interpreter delegiert. Fehler im Python-Code werden mit einer Python-Exception behandelt und bringen den Interpreter nicht zum Absturz. Abstürze durch Speicherzugriffsfehler sind so nahezu unmöglich. Zusammenfassend kann man also sagen:

- Mit Python ist möglich, schneller und einfacher ein neues Plug-in zu entwickeln, als in *C*, da der Programmierer sich nicht mit Dingen wie Speicher allokieren und freigeben beschäftigen muss, sondern sich direkt auf seine Plug-in-Funktionalität

konzentrieren kann.

- Auch unerfahrene Entwickler können relativ einfach ein Plug-in entwerfen, da sie bei einem Fehler nur eine Exception bekommen und keinen Absturz des Programms verursachen – das vereinfacht auch das *Debugging*.
- *PySPANK* bietet eine einfache Möglichkeit zum Registrieren und Validieren von Kommandozeilenparametern.

3.2 Funktionsweise

PySPANK selbst ist als ein normales *SPANK*-Plug-in implementiert. Als solches bietet es eine Implementierung für die in Kaptitel 2.3 vorgestellten Funktionsprototypen. Das Plug-in initialisiert dann einen eingebetteten Python-Interpreter und lädt ein oder mehrere Python-Skripte, die dann die in Python entwickelten Plug-ins darstellen.

Problem ist, dass Python-Code nicht direkt wie eine C-Funktion aus einer *Shared Library* exportiert und aufgerufen werden kann. Die Einstiegspunkte müssen also weiterhin C-Funktionen sein, die dann bei Aufruf an das geladene Plug-in verweisen. Weiter kann Python-Code auch nicht direkt C-Funktionen aufrufen. Dies ist jedoch für *SPANK*-Plug-ins wichtig, da eine *API* zum Abfragen von Informationen über den aktuellen Job und die Umgebung existieren, sowie Funktionen zum Verändern der Startumgebung. Diese Funktionen müssen aus Python heraus aufrufbar gemacht werden.

Die *PySPANK*-Architektur basiert auf drei Ebenen: SPANK Plug-in, Abstraktion und Übersetzung und den *PySPANK* Plug-ins. Im Folgenden werden diese drei Ebenen näher erläutert.

SPANK Plug-in In dieser Schicht gibt sich *PySPANK* als ein ganz normales in *C* geschriebenes Plug-in. Es sind die in Kapitel 2.3 vorgestellten Funktionen implementiert. Wird die `slurm_spank_init` Funktion aufgerufen, initialisiert das Plug-in einen neuen Python-Interpreter durch Aufruf der Funktion `PyInitialize`. In diesen wird die in *Cython*[4] als Python-Modul entwickelte Abstraktionsschicht `pyspank` geladen. Daraufhin wird in den in der Konfigurationsdatei `plugstack.conf` definierten Parameter ein oder mehrere `script=`-Parameter extrahiert und in das Plug-in geladen. Jedes dieser Skripte stellt ein *PySPANK*-Plug-in bereit.

[4]Cython ist eine Programmierspache, die es ermöglicht teils *getypten Python*-Code in ein C-Modul zu kompilieren und die Interaktion mit *C*-Funktionen vereinfacht

Abstraktion und Übersetzung Die Abstraktions-Schicht ist die aufwendigste der drei Ebenen. Hier werden die Aufrufe aus dem in Python geschriebenen *PySPANK* Plug-in in die entsprechenden Aufrufe an die *SPANK-API* nach *C* übersetzt. Die von der *C-API* zurückgegebenen Werte müssen dann ebenfalls wieder in Python-Typen umgewandelt werden.

```
cdef spank_get_item__str(c.spank_t spank, c.spank_item_t item):
    """ This will call spank_get_item for an argument of type
        char* like S_SLURM_VERSION and returns a string
    """
    cdef char *charptr
    cdef c.spank_err_t result = c.spank_get_item(spank, item,
       ref(charptr))
    if result != c.ESPANK_SUCCESS:
        raise SpankException(c.spank_strerror(result))
    return charptr

cdef class SlurmPlugin:
    # [...]
    property slurm_version:
        def __get__(self):
            return spank_get_item__str(self.spank, c.S_SLURM_VERSION)
```

Listing 5: Beispiel für die Interaktion zwischen Python und *C*-Code

SPANK bietet eine Methode `spank_get_item` zum Abfragen von Informationen über den aktuell zu startenden Job und über die Umgebung. Diese Methode nimmt neben einem `spank_t`-Handle einen Integer-Parameter an, der beschreibt, welche Information abgefragt werden soll. Danach folgt ein weiterer Parameter abhängig von dem Typ der Information. Für diese Methode existiert in der Übersetzungsschicht für jeden möglichen Typen eine eigene Cython-Methode, die sich um die Übersetzung zwischen den Sprachen kümmert. In Listing 5 ist beispielhaft die Umwandlung zwischen `char*` und einem Python-String zu sehen.

Neben der Übersetzung zwischen den beiden verwendeten Sprachen findet auch eine Abstraktion statt. Die *C-API* wird in eine objektorientierte Form gebracht, die in der Sprache Python üblicher ist. Dafür wurde eine abstrakte Basisklasse `SpankPlugin` entworfen, die die Funktionen `init`, `init_post_opt` und `exit` sowie alle anderen möglichen `slum_spank_*`-Prototypen enthält. Ein *PySPANK*-Plug-in erweitert nun diese Klasse und kann alle für das Plug-in relevanten Methoden überschreiben. Wie man in dem gezeigten Quelltext sehen kann, werden von der *C-API* zurückgegebene Fehler mit ihrer

Beschreibung als eine `SpankException` geworfen, die dann an einer geeigneten Stelle mit einem `try/except`-Block bearbeitet werden kann.

Zugriff auf die *SPANK-API* bekommt das Plug-in über die Exemplarvariable `spank`, die nach dem Muster in Listing 5 Eigenschaften für die relevanten Informationen enthält. Die Versionsnummer von *SLURM* kann aus einem Plug-in dann beispielsweise über den Ausdruck `self.spank.slurm_version` ausgelesen werden.

Zum Setzen und Auslesen von Umgebungsvariablen müssen in *SPANK* abhängig vom Kontext unterschiedliche Methoden verwendet werden. Dies wird von *PySPANK* ebenfalls abstrahiert, sodass der Entwickler eines *PySPANK* Plug-ins immer die Methode `self.spank.setenv` verwenden kann. Die Abstraktionsschicht bestimmt dann die zu verwendende Methode und ruft diese mit den gewünschten Parametern auf. Da Python das Überladen von Operatoren erlaubt, wär es hier denkbar, eine Klasse zu implementieren, die die Schnittstelle des Typs `dict` – ein assoziatives Array – emuliert, um so sogar von den Funktionen `setenv` und `getenv` zu abstrahieren.

Eine Besonderheit von *PySPANK* ist die Abstraktionsschicht für Kommandozeilen-Parametern. Jedes *PySPANK* Plug-in besitzt eine Instanz der Klasse `OptionTable` und die Methode `init_options`. Sollen für ein Plug-in Parameter registriert werden, kann in dieser Methode mit der Funktion `create` ein neuer Parameter in der `OptionTable`-Instanz erzeugt werden. Dafür bekommt die `create`-Methode verschiedene Parameter (vergleiche auch Listing 6):

`name` Name des Parameters, mit dem er auf der Kommandozeile übergeben werden kann. Der Name muss über allen *SPANK* Plug-ins hinweg eindeutig sein.

`usage` Beschreibung des Parameters als ein kurzer Text, der angezeigt wird, wenn z. B. `srun` mit dem Parameter `-help` aufgerufen wird.

`argtype` Erwartet der Parameter ein Argument, muss hier ein Objekt übergeben werden, das den als Zeichenkette übergebenen Wert validiert und in den gewünschten Argument-Typ übersetzt. Dafür muss eine Methode `convert` bereitgestellt werden. Weiterhin muss ein Attribut `info` existieren, das das Argument identifiziert, damit es im Hilfetext referenziert werden kann. Beispielsweise hat `info` den Wert `N` für eine Integerzahl. Die Ausgabe kann dann lauten:

```
--nice N    sets the nicelevel to N
```

`default` Wird kein Argument für einen Parameter übergeben, wird dieser Wert als Argument angenommen.

`required` Ein boolscher Wert mit dem spezifiziert wird, ob es zwingend notwendig ist, den Parameter anzugeben. Dies ist nur sinnvoll, wenn der Parameter ein Argument erwartet.

Das Argument wird dann von *PySPANK* mit *SPANK* registriert. Wird es auf der Kommandozeile übergeben, wird das Argument validiert und in der entsprechenden mit `create` erzeugten Instanz gespeichert. In der `init_post_opt`-Methode wird zusätzlich überprüft, ob alle erforderlichen Parameter übergeben wurden.

Wird die von `create` zurückgegebene Referenz gespeichert, kann später über das Feld `provided` geprüft werden, ob das Argument gesetzt wurde und über das Feld `value` der gesetzte Wert ausgelesen werden. Dies ist auch im Beispiel in Listing 8 umgesetzt.

```
def init_options(self):
    self.prio = self.options.create(
        name = "renice",
        usage = "sets the priority for the job to N",
        argtype = pyspank.IntegerArgumentType(min=0, max=19))
```

Listing 6: Erzeugen einer Option für das `nice`-Plugin

PySPANK Plug-ins Jedes in Python geschriebene Plug-in muss als eine einzelne `.py`-Datei vorliegen und eine `create`-Methode mit zwei Parametern – *SPANK*-Handle und Argumente aus der `plugstack.conf` – annehmen. Diese Methode wird aufgerufen um eine neue Plugin-Instanz zu erzeugen. Über das mitgegebene *SPANK*-Handle kann der aktuelle Ausführungskontext ermittelt werden und die Initialisierung des Plug-ins abhängig von diesem gemacht werden. Ein Beispiel dafür ist in den letzten Zeilen von Listing 8 zu sehen. Das `nice`-Plugin soll ausschließlich im Kontext des *slurmd*-Daemons ausgeführt werden. Dies wird mit `spank.remote` geprüft.

Das zurückgegebene Objekt sollte von der Klasse `SpankPlugin` aus dem Modul `pyspank`, das mit `import` eingebunden werden kann, erben.

3.3 Beispiel

In Listing 8 ist ein vollständiges *PySPANK* Plug-in gedruckt. In der ersten Zeilen werden alle notwendigen Module importiert. Daraufhin wird eine Referenz auf die *C*-Bibliothek

über das `ctypes`-Modul erfragt. `ctypes` ist eine Python-Bibliothek, die es erlaubt *C* Funktionen in einer *SharedLibrary* aus Python heraus aufzurufen.

Nun beginnt die eigentliche Definition des Plug-ins. Es wird eine Klasse `NicePlugin` definiert, die die Klasse `SpankPlugin` aus der Abstraktionsschicht erweitert. Das Plug-in überschreibt zuerst die Methode `init_options` um einen eigenen Parameter `renice` zu registrieren. Dieser erwartet eine Ganzzahl zwischen null und neunzehn. Die von `create` zurückgegebene Referenz wird als Feld `prio` für das spätere Auslesen des Parameterwertes gespeichert.

Die zweite definierte Methode ist `task_post_fork`, die von *SPANK* aufgerufen wird, nachdem ein neuer Prozess für einen Task abgespalten wurde, aber vor das auszuführende Programm mit `exec` gestartet wird. Nun wird geprüft, ob der oben erzeugte Parameter gesetzt wurde. Wenn nicht, beendet das Plug-in hier seine Aktivität. Ist der Parameter gesetzt worden, wird die *SLURM*-interne *Task-Id* und die *Prozess-Id* im Betriebssystem des abgespalteten Prozesses erfragt. Weiter holt sich das Plug-in in Zeile 21 den im Parameter übergebenen und bereits validierten Wert als Integer. Mit diesen Werten wird dann in Zeile 24 die `setpriority`-Funktion aus der *C*-Bibliothek aufgerufen.

In Zeile 26 ist die `create`-Methode definiert, die eine Instanz des Plug-ins für den `remote`-Kontext erstellt. Für jeden anderen Kontext wird keine Instanz und `None` zurück gegeben.

Das Plug-in muss nun in die *SPANK*-Konfigurationsdatei `plugstack.conf` eingefügt werden und kann dann benutzt werden.

```
node0:~$ cat /etc/slurm-llnl/plugstack.conf
required    pyspank.so      script=/usr/lib/pyspank/renice.py

node0:~$ sbatch --renice 10 myjob.sh
sbatch: Submitted batch job 314
```

Listing 7: Verwendung des `nice`-Plug-ins

```
import pyspank, ctypes, ctypes.util

# we need to load libc, because python has no setpriority
libc = ctypes.cdll.LoadLibrary(ctypes.util.find_library("c"))

class NicePlugin(pyspank.SpankPlugin):
    def init_options(self):
        self.prio = self.options.create(
            name = "renice",
            usage = "sets the priority for the job",
            argtype = pyspank.IntegerArgumentType(min=0, max=19))

    def task_post_fork(self):
        super(NicePlugin, self).task_post_fork()
        if not self.prio.provided:
            pyspank.info("--renice option not set")
            return

        task = self.spank.task_global_id
        pid = self.spank.task_pid
        prio = self.prio.value

        pyspank.info("set prio for task %d (pid=%d) to %d" % (task,
            pid, prio))
        libc.setpriority(0, pid, prio)

def create(spank, args):
    """ Erzeugt eine neue Instanz des Nice-Plugins """
    return NicePlugin(spank, args) if spank.remote else None
```

Listing 8: Das vollständige `nice`-Plug-in zum Setzen der Priorität eines Tasks

4 Fazit

Unser Vorhaben eine Plug-in Schnittstelle auf Basis von *SPANK* für in Python geschriebene Plug-ins konnten wir verwirklichen. Dies vereinfacht es Entwicklern und insbesonders Einsteigern einfache *SPANK*-Plug-ins zu entwerfen und zu implementieren.

Wir konnten feststellen, dass trotz Verwendung des Python-Interpreters die Startdauer eines Jobs nur etwa 50 ms bis 100 ms länger dauert. Da es sich in Cluster-Systemen im Normalfall ohnehin um länger laufende Prozesse handelt, stellt die verlängerte Startdauer kein Problem dar.

PySPANK stellt folglich eine gute Alternative zur direkten Entwicklung in *C* dar, wenn der Entwickler bereit ist, einen Teil seiner Kontrolle an den Python-Interpreter abzugeben. Dies scheint jedoch keine Hürde darzustellen, da es bereits eine inoffizielle *LUA*-Anbindung an *SPANK* gibt [1].

Literatur

[1] Lua spank. http://code.google.com/p/slurm-spank-plugins/source/browse/#git%2Flua. abgerufen am 22.09.2012.

[2] Slurm: A highly scalable resource manager. https://computing.llnl.gov/linux/slurm/slurm.html. abgerufen am 05.10.2012.

[3] Slurm: Architecture. https://computing.llnl.gov/linux/slurm/overview.html. abgerufen am 08.10.2012.

[4] Slurm spank plugins. http://code.google.com/p/slurm-spank-plugins/. abgerufen am 08.10.2012.