PLANETLAB - EINE ÜBERSICHT

STUDIENARBEIT

VON

MARCUS DRAMBURG
MATRIKELNUMMER: 1568450

20. FEBRUAR 2008

BETREUER:
PROF. T. SCHMIDT

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
FAKULTÄT TECHNIK UND INFORMATIK

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit ist an der Fakultät Technik und Informatik der Hochschule für Angewandte Wissenschaften in Hamburg, im Rahmen einer Studienarbeit, entstanden. Sie soll einen kurzen Einblick in das Arbeiten mit PlanetLab geben und untersuchen, inwieweit PlanetLab als Testumgebung für Echtzeit Peer-to-Peer Anwendungen einsetzbar ist.

Dazu wird in Kapitel 2 PlanetLab kurz vorgestellt und ein kurzer Einblick in den Aufbau des PlanetLab Systems gegeben. Kapitel 3 gibt einen Überblick über angebotene Dienste und Softwarewerkzeuge zum Deployment, Monitoring und Management. Die Tests zur Performanceanalyse werden in Kapitel 4 vorgestellt. Schließlich wird in Kapitel 5 ein Fazit gezogen und ein Ausblick auf die zukünftige Entwicklung PlanetLabs gegeben. In Anhang A findet sich ein kurzes Howto, welches ein paar grundlegende Mechanismen, die den Rahmen der einzelnen Kapitel sprengen würden, zur Benutzung PlanetLabs erklärt.

2 Das Planeten Laboratorium

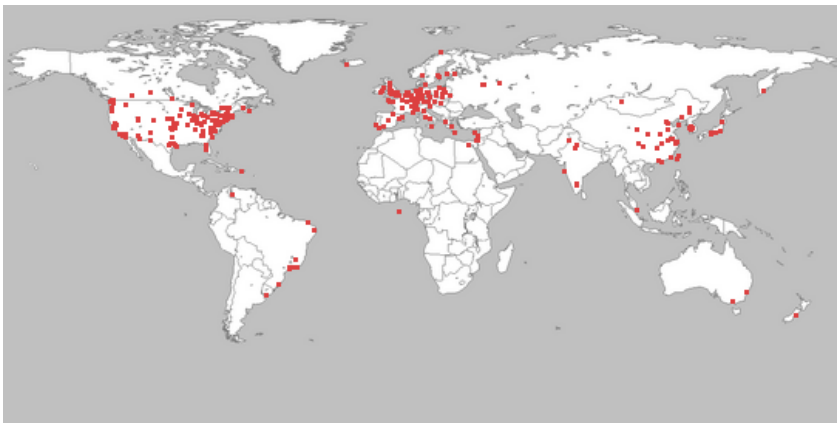

Abbildung 2.1: Verteilung der PlanetLab Knoten. Quelle: [plW08]

PlanetLab ist ein Anfang 2003 an den Start gegangener Rechnerverbund, der zum Zeitpunkt dieser Niederschrift laut Angabe der PlanetLab Homepage[1] aus 836 Knoten an 411 Standorten besteht. Unter den Eindrücken die das Aufkommen einer neuen Technologie von verteilten Systemen, den Peer-to-Peer- Netzwerken, auslöste, trafen sich Anfang 2002 David Culler *(U.C. Berkely)* und Larry Peterson *(Princeton University)*, um über die Möglichkeiten zum Aufbau einer über den Globus verteilten Testumgebung für neue Netzwerktechnologien zu diskutieren. Bis zu diesem Zeitpunkt testeten Netzwerkforscher entweder in Netzwerksimulationsumgebungen wie das von der University of Utah angebotene Emulab oder lokal auf einer Reihe von institutseigenen Rechnern. Im März 2002 trafen sich Culler und Peterson bei einem informellen Workshop zum selben Thema in den Intel Research Labs in Berkeley wieder [Ros05]. Das Ergebnis dieses Zusammentreffens war eine Vision des zu schaffenden PlanetLab und wurde in [PACR02] veröffentlicht. Dieses Dokument geht aber über einen rein visionären Charakter hinaus, da hier auch schon Konzepte vorgestellt wurden, die dann in PlanetLab verwirklicht wurden, so zum Beispiel das Slice-Konzept zur Hardwarevirtualisierung. Die Intel Research Labs forcierten die Realisierung der neuartigen Testumgebung durch Spenden der ersten 100 Knoten, die an weltweit 40 Universitäten ausgeliefert wurden, desweiteren wurde PlanetLab bis 2004 von

[1] http://www.planet-lab.org/

Intel verwaltet. Die Verwaltung ging dann 2004 an das neu gegründete PlanetLab Consortium. Seitdem verzeichnet der PlanetLab Rechnerverbund einen stetigen Zuwachs an Standorten und Knoten. Ursprünglich war der Ausbau des PlanetLab in drei Phasen geplant, siehe hierzu auch [PACR02]. In Phase Eins sollten in PlanetLab nur Hochschulstandorte aufgenommen werden und die Nutzung Mitarbeitern dieser Einrichtungen vorbehalten werden. In Phase Zwei sollte der Teilnehmerkreis auf Forschungsinstitute und die Privatwirtschaft ausgedehnt und Planetlab Gegenstand von Vorlesungen und Seminaren werden. Dabei sollte einer Vielzahl von Studenten die Möglichkeit zur Nutzung gegeben werden. In der dritten Phase schließlich sollte die Benutzung öffentlich werden, also jedem möglich sein, der bereit wäre seinen Rechner via Modem oder DSL an das PlanetLab anzuschließen. Gerade von der Realisierung der dritten Phase ist man aber, auch im Zuge der sich verschärfenden Gesetzeslage bezüglich der Erstellung und Verbreitung von P2P Anwendungen in den USA und weltweit [MS07], wieder abgekommen. Im Folgenden werden nun die PlanetLab zugrunde liegenden Designprinzipien, sowie der Aufbau erläutert und etwas zu den Nutzungsbedingungen der AUP [2] (Acceptable Use Policy) gesagt. Das Kapitel schließt mit einer kurzen Betrachtung zur Verfügbarkeit der Rechnerknoten im PlanetLab.

2.1 Design und Entwurf

Von Anfang an wurde bei der Entwicklung des PlanetLab, neben der Entscheidung den Rechnerverbund als Peer-to-Peer System zu entwerfen, auf die Einhaltung dreier grundlegender Prinzipien Wert gelegt [PR06]. Diese Prinzipien wurden unter den Schlagworten *Distributed Virtualization*, *Application Centric Interfaces* und *Unbundled Management* publiziert, so in [PACR02] und [MS07]. Hier sollen diese Begriffe kurz erläutert werden.

2.1.1 Distributed Virtualization

Mit diesem Designprinzip sollte formuliert werden, dass die zu schaffende Testumgebung, im Gegensatz zu bisherigen Ansätzen, nicht über einen zentralen Schedulingmechanismus verfügt, durch den einem Nutzer für einen gewissen Zeitraum alle Knoten mit ihrer gesamten Rechenleistung für die Durchführung eines Experiments gewährt werden. Vielmehr sollte es die neu zu schaffende Testumgebung ermöglichen, viele Experimente parallel, kontinuierlich und über lange Zeit laufen zu lassen. Um dies zu gewährleisten, musste eine Lösung zur Hardwareabstraktion gefunden werden, die ihrerseits möglichst weniger Ressourcen bedarf und einen Mechanismus zur Verfügung stellt der den verschiedenen Nutzern einen möglichst fairen Zugriff auf die Ressourcen der am Experiment beteiligten Rechner gewährleistet.

[2] siehe unter:
http://www.planet-lab.org/aup

2.1.2 Application Centric Interfaces

Unter diesem Punkt wurde die Forderung erhoben, dass für die neu zu entwickelnde Testumgebung keine neuen und unnötig komplexen Programmierschnittstellen entwickelt werden sollten. Vielmehr wollte man das System auf bekannten und zumindest im Netzwerkbereich weithin benutzten Schnittstellen aufbauen. Der Benutzer sollte in die Lage versetzt werden, mit der neuen Anwendung sofort arbeiten zu können – ohne langwierige Einarbeitungszeit. Hierin begründet liegt z.B. die Entscheidung mit Linux eine Unix-artige Systemschnittstelle zu wählen. Die notwendigen Änderungen am Betriebssystemkern sollten so behutsam erfolgen, dass möglichst alle Systemaufrufe, wie auf einem herkömmlichen Linux-System zu nutzen wären. Auch die Wahl der XML-RPC-Schnittstelle zur PlanetLab Central Datenbank liegt hierin begründet, da hierfür eine große Anzahl an Bibliotheken für fast alle Programmiersprachen zur Verfügung stehen.

2.1.3 Unbundled Management

Hiermit sollte sichergestellt werden, dass das zu entwickelnde System, welches man selbst als langwieriges Experiment mit ungewissem Ausgang betrachtete, auch zukünftige Anforderungen und Entwicklungen mit einbeziehen können sollte. Deshalb sollte die Verwaltung und ein Teil der Funktionalität vom Kern der Anwendung getrennt und in Form von lose an das System gekoppelten Diensten realisiert werden. Davon versprach man sich zum einen die Möglichkeit Teile der Verwaltung und Funktionalität auszulagern und in die Hände einzelner Nutzergruppen zu geben und zum anderen die Freiheit konkurrierende Dienste zuzulassen. Dies sollte dem Nutzer die Möglichkeit zur Wahl des für ihn besten Systems einräumen.

2.2 Architektur und Aufbau

Die oben angeführten Konzepte finden sich in allen Bereichen der Realisierung wieder. Die verteilte Virtualisierung im Slice und Sliver Konzept und ihrer Umsetzung mittels der Linux Kernel Erweiterung VServer [3], die anwendungsbezogen Schnittstellen in der Umsetzung der Netzwerkschnittstelle mit VNET und der Systemschnittstelle mit der Wahl eines RedHat basierten Linux als Betriebssystem und die lose an den Kern gekoppelten Dienste in der Umsetzung der Monitoring und Abfrage Dienste, die ausführlich in Kapitel 3 vorgestellt werden sollen. Der PlanetLab Central Dienst findet allerdings schon Erwähnung in diesem Kapitel, da er noch einige fundamentale Aufgaben erfüllt, die für den Aufbau des Systems entscheidend sind.

[3] Detailiertere Informationen hierzu unter: http://linux-vserver.org/Documentation

2.2.1 Slice und Sliver

Slice und *sliver* sind die grundlegenden Abstraktionen der Hardware innerhalb PlanetLabs [PHM+04]. Dabei entspricht ein *slice* der Gesamtheit aller Ressourcen, die einem Benutzer zugeteilt werden, über das Gesamtsystem verteilt. Ein *sliver* hingegen entspricht dabei der Hardwareabstraktion auf einen Rechner bezogen. *Sliver* werden durch VServer realisiert und präsentieren sich dem Nutzer, der sich von außen unter Verwendung des *slice* Namen als Benutzerkennung, mittels ssh in einen Knoten einlogt, wie ein entfernter Linuxrechner. Ein *slice* hat nach Erzeugung nur eine begrenzte Gültigkeitsdauer von vier Wochen. Diese kann aber unbegrenzt auf weitere vier Wochen vom Tag des Setztens verlängert werden. Vier Tage vor Ablauf erhält der Nutzer täglich eine Erinnerungsmeldung per E-Mail.

2.2.2 VServer

VServer können als eine Art Erweiterung der klassischen chroot Umgebung aufgefasst werden, wobei die Erweiterung in der Unterstützung von disk-quotas, einem Mechanismus zum fairen Scheduling zwischen den einzelne VServern, einem eigenem Prozessbaum und einer jeweils eigenen Benutzerverwaltung besteht. Die Gastsysteme verfügen über keinen eigenen Betriebssystemkernel, wie Virtualisierungslösungen die auf einer vollständigen Nachmodellierung der Hardware aufsetzen, so z.B. XEN oder VMWare, sondern über eine private Schnittstelle zum Kernel, der den Hardwarezugriff kapselt und an die VServer verteilt. Dabei wird von einem, Unifikation genannten, Mechanismus Gebrauch gemacht, der es dem Kernsystem ermöglicht Standard Binaries und Libraries mit den Gastsystemen zu teilen. Dies geschieht mittels Hardlinks, die nicht veränderbar, aber löschbar sind, in das Gastsystem. Das Gastsystem kann diese nun benutzen oder durch eigene Anwendungen und Bibliotheken ersetzen. Zur Grundausstattung eines neu erzeugten *sliver*, das via VServer ausgeliefert wird, gehören grundlegende Unix-Kommandos zur Gruppen- und Benutzerverwaltung, zum Datei- und Prozessmanagement, sowie ein *vi* Editor, ein Python und ein Perl Interpreter. Für Python sind viele Programmbiblitheken, für Perl und C/C++ Standardbibliotheken vorhanden. Buildtools, wie *make*, *autoconf* und C/C++ Compiler fehlen, da auf den PlanetLab Knoten aufgrund der eh schon hohen Last, nicht kompiliert werden soll. Weiterhin findet sich der RedHat Package Manager *rpm* in der Installation, mit dessen Hilfe fehlende Softwarepakete nachinstalliert werden können. Das System entspricht mit wenigen Abänderungen der RedHat basierten Linux-Distribution Fedora Core 2.

2.2.3 VNET

VNET [4] ist die innerhalb PlanetLabs benutzte Abstraktion der Netzwerkschnittstelle. Hierzu greift VNET auf auf das Netfilter [5] System des Linux Kernel, ein in den Kernel integrierter Paketfilter, zu. Das wesentliche Funktionsprinzip ist die permanente Ablaufverfolgung aller bestehender Netzwerkverbindung und deren Zuordnung zu einem VServer [Hua05]. Deshalb können von *slices* auch nur Pakete versendet werden, die einer von ihnen initiierten Verbindung zugeordnet werden können und ebenfalls nur Pakete empfangen werden, die von einer dem *slice* zuzuordnenden Verbindung angefragt wurden oder zu einem gebundenen Socket gehen. Unter VNET lassen sich zur Zeit die folgenden Protokolle [6] nutzen: TCP, UDP, ICMP, GRE und PPTP.

2.2.4 Proper

Proper (*privileged operations*) ist ein Dienst, der unter dem root account jedes Knotens läuft und die Nutzung von privilegierten Betriebssystemdiensten auch eingeschränkten Benutzern aus den VServern heraus gestattet , so z.B. die Nutzung von Portnummern unterhalb 1024 oder die Benutzung von RAW Sockets, siehe hierzu auch [MPF+05].

2.2.5 PlanetLab Central

PlanetLab Central ist der zentrale Bootserver, von dem bei Neustart eines Knoten Programmupdates und Informationen über andere Knoten bezogen werden können. Desweiteren fungiert PlanetLab Central als zentrale Datenbank mit Informationen zu den Knoten. Diese Informationen können über eine XML-RPC Schnittstelle [7] abgerufen oder abhängig von der Benutzerrolle verändert werden. So wird z.B. über die XML-RPC Schnittstelle die Zuordnung von Knoten zu einem *slice* vorgenommen. Ein Teil der Funktionalität ist nach einem Login, über die PlanetLab Homepage erreichbar, so das Hinzufügen und Entfernen von Knoten und Benutzern zu einem *slice* oder das Erneuern des Ablaufdatums eines *slice*.

[4] Weitergehende Informationen hierzu unter: http://www.planet-lab.org/doc/vnet
[5] siehe auch: http://www.netfilter.org/
[6] Dies allerdings nur mit Einschränkungen.
genaueres hierzu siehe unter: http://www.planet-lab.org/doc/vnet
[7] siehe hierzu auch: http://www.planet-lab.org/doc/plc_api

2.3 Acceptable Use Policy

In der AUP sind die Regeln für die Benutzung PlanetLabs zusammengefasst. Vor dem Start der eigenen Experimente, sollten diese kurz gelesen und verinnerlicht werden, da eine Nichtbeachtung oder Verletzung dieser Regeln im Extremfall zum Ausschluss von der Nutzung führen kann.

2.4 Verfügbarkeit der Knoten

Als Ergänzung zu den oben angeführten Zahlen an Knoten und Standorten seien hier noch ein paar Daten aufgeführt, die so am 13. Januar 2008 mit Hilfe der Dienste CoMon und CoVisualize des CoDeen Projekts erhoben wurden. Es war feststellbar, dass von den 790 im CoMon-Dienst aufgeführten Knoten insgesamt 295 als *dead* markiert waren, davon waren 47 schon über ein Jahr mit diesem Label versehen, 71 Knoten seit 101 - 365 Tage, 47 Knoten zwischen 31 - 100 Tage und 130 Knoten seit 1- 30 Tagen vollständig unerreichbar. 495 Knoten waren mit dem Label *good* versehen, dass vergeben wird, wenn entweder ssh-login, CoTop-Abfrage oder heartbeat innerhalb der letzten 15 Minuten möglich war. Von den genannten Standorten waren 100 vollständig, d.h. mit all ihren gelisteten Knoten *down*. Und nur 94 Standorte waren vollständig, d.h. mit all ihren Knoten, funktionstüchtig. Die übrigen Standorte hatten zumindest einen Rechener in Betrieb. Nicht unerwähnt bleiben soll auch, dass zu diesem Zeitpunkt an 27 der 100 vollständig funktionsuntüchtigen Standorte PlanetLab mit mehreren für diese Standorte gelisteten Slices in Benutzung war. Diese Untersuchung wurde am 01. Februar 2003 wiederholt und ergab ein ähnliches Bild. An 95 Standorten waren alle gelisteten Knoten down, an 23 dieser Standorte wurde PlanetLab genutzt und 85 Standorte waren mit allen ihren gelisteten Rechnern verfügbar. Zu diesem Zeitpunkt waren 798 Knoten im CoMon Dienst verzeichnet.

	13.01.2008	**01.02.2008**
Anzahl Knoten (gelistet in CoMon)	790	798
verteilt auf Standorte	315	319
Standorte voll verfügbar	94	85
Standorte nicht verfügbar	100	95
Standorte nutzen PlanetLab ohne Knoten verfügbar zu halten	27	23

Tabelle 2.1: Zustand der Knoten und Standorte am 13. Jan. und 01. Feb. 2008

3 Dienste und Werkzeuge

Ein Großteil der Funktionalität PlanetLabs ist gemäß der Designvorgaben (siehe Kapitel 2.1) in Dienste ausgelagert. Da die Benutzung dieser Dienste für jeden der PlanetLab als Basis eigener Experimente nutzen möchte zumindest teilweise unvermeidbar ist, soll hier kurz auf ein paar der wichtigsten hingewiesen werden. Darüber hinaus existieren eine Unzahl an Anwendungen und Softwarewerkzeugen, die den Umgang mit PlanetLab vereinfachen und effizienter gestalten können, davon sollen ebenfalls ein paar vorgestellt werden. Da viele der hier aufgeführten Softwarewerkzeuge als Sourcecode vertrieben werden sei auch darauf hingewiesen, dass es sich bei vielen der hier vorgestellten Anwendungen als günstiger erwiesen hat, sie auf einer RedHat basierten Linux-Distribution zu kompilieren, da entweder benötigte Bibliotheken nur im *rpm* Format vorliegen und sich nur mit Aufwand oder gar nicht mit Hilfe des für solche Probleme unter Debian-Distributionen üblichen *alien* Skripts in *deb* Pakete oder andere Formate umwandeln lassen. Beispielhaft sei hier *plush* genannt, für das sogar ein fertig kompiliertes Debian-Paket existiert, das explizit ohne Garantie angeboten wird und auf aktuellen Debian basierten Distributionen nicht zum Laufen zu bewegen war, für das eine Vielzahl benötigter Bibliotheken nur im *rpm* Format vorliegen, das aber im Zusammenwirken mit der Java basierten Oberfläche *nebula* ein mächtiges Werkzeug ist und die Benutzung einer RedHat basierten Distribution rechtfertigen kann. Mit der in Anhang A angeratenen Lösung lassen sich aber die meisten solcher Probleme umschiffen und sich die Software nutzen.

3.1 Dienste

3.1.1 CoDeeN

CoDeeN (*Content Distribution Network*) ist ein Projekt der University of Princeton, das eine Verteilungsinfrastruktur über ein Netzwerk aus Proxyservern bereitstellt. Diese Proxyserver, die auf den meisten PlanetLab Knoten laufen, können einerseits als *request redirectors* bei beliebigen Nutzern als Proxy z.B. im Webbrowser eingetragen werden, oder aber als System von Servern, deren Funktionalität über optionale Programmpakete erweitert werden kann. Auf CoDeeN setzten ein paar, besonders für das Monitoring, sehr interessante Dienste auf, die hier kurz besprochen werden sollen.

CoBlitz

CoBlitz ist ein Verteilungssystem für große Datenpakete über http. Mittels CoBlitz werden sechs Fedora Core Mirror Server betrieben, die auch zur Aktualisierung der PlanetLabKnoten, anstelle des PlanetLab Central Servers benutzt werden können. Diese sind aber auch über PlanetLab hinaus nutzbar.

CoDeploy

Hinter CoDeploy verbirgt sich eine kleine Sammlung von einfach zu benutzenden Programmen und Skripten, mit deren Hilfe man beliebige Dateien/Ordner auf alle Knoten eines *slice* verteilen, oder von allen Knoten (z.B. *Logfiles* der eigenen Anwendung) auf seinen Arbeitsplatzrechner laden kann. Hierzu muß lokal ein Webserver laufen und die zu transportierenden Daten sich in einem Verzeichnis des Webservers befinden. CoDeploy bedient sich dann des CoDeeN Proxy-Netzwerks um die Dateien auf die Knoten zu verteilen oder von diesen zu laden.

CoDNS

CoDNS ist ein kooperativer Namensdienst, der Namensauflösungen zwischenspeichert und nur dann DNS Server kontaktiert, wenn ein Name nicht aus dem *Cache* aufgelöst werden kann [PPPW04]. CoDNS läuft auf fast allen PlanetLab Knoten als Dämon-Prozess und ist einfach zu benutzen. Vom Projekt werden eine C-Header-Datei und eine C-Source-Datei zur Verfügung gestellt, die in das eigene Projekt eingebunden werden müssen. Dann ist im eigenen Sourcecode nur noch der Aufruf von *gethostbyname()* durch *CoDNSGetHostByNameSync()* zu ersetzen und schon wird CoDNS genutzt.

CoTop

CoTop ist eine Abänderung des bekannten *top* und setzt auf einem auf allen PlanetLab Knoten installierten Sensor Interface [RPKW03] auf, das bezogen auf den Knoten Processinformationen des root Accounts und aller aktiven slices auf dem Knoten als Prozesstabelle ausgibt. Die Ausgabe des Programms kann sowohl lokal auf den einzelnen Knoten mit einem Aufruf von `cotop` als auch mit jedem http-fähigen Client unter der URL http://FQDN-des-Knoten:3120/cotop abgerufen werden.

CoMon und CoVisualize

CoMon und CoVisualize sind zwei Dienste, die das eben genannte CoTop nutzen, um verschiedene Sichten auf den „Gesundheitszustand“ des gesamten PlanetLab, einzelner Knoten oder eines slices in tabellarischer oder grafischer Form zu generieren und über ein Webinterface zugänglich zu machen. Besonders CoVisualize eignet sich vorzüglich um mit einem Blick eine Übersicht des aktuellen Zustands zu erhalten, siehe hierzu auch die Abbildungen B.1 und B.2 die aus CoVisualize extrahiert wurden.

3.1.2 Sirius

Sirius ist ein Dienst, mit dem man für das eigene Slice einmalig für eine Stunde eine garantierte Prozessorzeit von 25 Prozent und eine garantierte Brandbeite von 2 MBits zugeteilt bekommen kann. Dazu ist eine Anmeldung auf der Website des Projekts[1], mit einer kurzen Schilderung des Experiments und warum man die Ressourcenerhöhung benötigt, notwendig. Die eigene Anwendung muss dann gestartet werden und auf eine Benachrichtigung über einen Unix-Domain-Socket warten[2]. Wird diese Nachricht empfangen kann das eigene Experiment für eine Stunde die garantierten Ressourcen nutzen.

3.1.3 SWORD

SWORD (Scalable Wide-Area Ressource Discovery) ist ein Dienst, der auf fast allen PlanetLab Knoten läuft und der dazu dient, die für die eigenen Anwendungen hinreichend perfomanten Knoten zu finden. Dazu fragt der SWORD-Dienst periodisch vier andere auf den PlanetLab Knoten laufende Dienste ab, u.a. das oben genannte CoTop und ermittelt dabei aktuell 54 Attribute nach denen eine Leistungseinschätzung der Rechner getroffen werden kann. Diese Daten werden auf einem zentralen Server abgelegt und können über eine XML-RPC basierte Query-Sprache abgefragt werden, siehe hierzu [OAPV04]. Dies kann über einen zur Verfügung gestellten Client oder über eine Weboberfläche geschehen. Das Ergebnis ist eine Liste von Rechnernamen oder eine leere Liste, falls keiner der Rechner den geforderten Attributen entspricht. Der Dienst, im besonderen über die Website, ist unkompliziert zu nutzen und unersätzlich zur Bestimmung von hinreichend performanten Rechnern für das eigene Experiment.

[1] https://snowball.cs.uga.edu/~dkl/sirius-expl.php
[2] Beispielcode findet sich unter:
https://snowball.cs.uga.edu/~dkl/code.c

3.2 Einfache Kommandozeilenanwendungen

Hier sollen kurz ein paar einfach zu installierende und benutzende Programme vorgestellt werden, mit denen sich prinzipiell schon alle Aufgaben, die im Zusammenhang mit der Nutzung eines *slice* auf PlanetLab auftauchen, wie z.B. das Kopieren der eigenen Anwendung auf alle Knoten, das Starten und Überwachen der Anwendung, erledigen lassen. Die hier vorgestellten Programme sind robust und auf einer Vielzahl von Plattformen einsetzbar, dafür fehlt ihnen der Komfort einer graphischen Benutzeroberfläche.

3.2.1 plcsh

Die *plcsh* (PlanetLab Central Shell) ist eine Entwicklung der Princeton University. Es ist eine vollwertige Python-Shell mit Builtin-Funktionen zur Interaktion mit der PlanetLab Central Datenbank. Dabei bilden diese Builtin-Funktionen ein einfaches Interface zu den teilweise komplexen XML-Queries. Die *plcsh* ist als Python-Modul auch in eigene Programme einbindbar, was die Verwendung sehr vielseitig macht. Ein Beispiel für eine Benutzung ist in Anhang A auf Seite 21 zu finden. Dort finden sich auch Informationen darüber, wie man an die jeweils aktuellste Version gelangt, da hierüber verschieden brauchbare Aussagen auf der PlanetLab Homepage und in verschiedenen Tutorials existieren und man mit älteren Versionen dieser Anwendung Unannehmlichkeiten erleben kann.

3.2.2 pssh

Pssh ist eine von den Intel Research Labs in Berkeley entwickelte Sammlung aus vier Kommandozeilenprogrammen, *pssh*, *pscp*, *pnuke* und *prsync*, die parallele Versionen bekannter Kommandozeilenprogramme sind, wobei *pnuke* ein paralleles *kill* ist. Die Programme sind schnell installiert und sind unkompliziert zu benutzen, wie ist in [Chu03] beschrieben. Diese Programmsammlung stellt das Minimum dessen dar, was benötigt wird, um mit seinem *slice* zu interagieren.

3.2.3 vxargs

Vxargs ist ein an der University of Pennsylvania entwickelter Wrapper in Python, der nahezu aus jedem Unix-Kommando eine parallele Version erzeugt. Inspiriert durch *pssh* und *xargs*, bietet dieses Kommandozeilenprogramm eine curses-basierte Oberfläche zur Präsentation der Ausgaben und des Status der initiierten Prozesse, das Speichern der Standardausgabekanäle zur späteren Analyse, sowie die Parallelisierung einer Vielzahl von Programmen mit ihrem vollständigen Funktionsumfang. Zu beziehen ist das Programm über die Homepage unter

http://dharma.cis.upenn.edu/planetlab/vxargs/, wo sich auch eine Einführung in die Benutzung findet.

3.3 Komplexe Managementsysteme

Die in diesem Abschnitt vorgestellten Mangementsysteme verfügen alle über eine Projektverwaltung. Darüberhinaus bieten sie Möglickeiten zum Monitoring der gestarteten Anwendungen,oder wie z.B. *nebula* über ein Lokalisierungssystem, das wahlweise alle Knoten oder nur die des aktuellen *slice* auf einer stufenlos zoombaren Weltkarte darstellt.Auch eine übersichtliche Verwaltung aller Knoten, sowie Zugriffsmöglichkeiten auf die PlanetLab Central Datenbank werden ermöglicht. So sehr sie die Verwaltung des eigenen *slice* komfortabler und effizienter gestalten können, so aufwendig kann es sein sie aus den Quellen zu kompilieren.

3.3.1 AppManager

Der AppManager aus den Intel Research Labs in Berkeley verfolgt ein Client-Server-Konzept und setzt sich aus einem Satz von bash-Skripten für die Client- und die Server-Seite zusammen. Dabei bilden die PlanetLab Knoten die Clients und ein lokal laufender Webserver mit Anschluss an einen PostgreSQL-Datenbank-Server das Server Frontend. Von einem lokalen Rechner müssen nun zuerst die Client-Skripte an die eigenen Bedürfnisse angepasst werden und dann auf die PlanetLab Knoten des betreffenden *slice* kopiert werden[3]. Dann ist der Webserver mit der Datenbank und den Server-Skripten aufzusetzen[4]. Nun können über die Oberfläche, die vom Webserver ausgeliefert wird, die eigenen Anwendungen gestartet, beobachtet und auch neue Knoten zum *slice* hinzugefügt werden.

3.3.2 PlMan

Der *PlanetLab Experiment Manager* ist an der University of Washington enstanden und bietet eine aufgeräumte Oberfläche mit der sich gut arbeiten läßt. Es ist eine reine Javaimplementierung und läßt sich somit auf vielen Plattformen betreiben. Von den hier vorgestellten Managementsystemen ist es das, welches mit dem geringsten Aufwand zu Benutzen ist. Unter Linux musste nur das fehlerhafte Startskript angepasst werden. Mit der Anwendung lassen sich Knoten in ein *slice* integrieren, Daten auf die Knoten verteilen, Anwendungen starten und beobachten, sowie

[3] Detaillierte Anleitung unter:
http://appmanager.berkeley.intel-research.net/client-README.txt
[4] bzw.:
http://appmanager.berkeley.intel-research.net/server-README.txt

Daten von den Knoten auf einen lokalen Rechner transferieren. Hierbei unterstüzt die Anwendung den Nutzer mit einer Vielzahl von hilfreichen Assistenten für die verschiedenen Aufgaben. Der Kosten-Nutzen Faktor ist hier maximal.

3.3.3 plush und nebula

Das mit Abstand leistungsfähigste der hier vorgestellten Managementsysteme ist das in C++ und Perl implementierte *plush*, das auch allein an der Kommandozeile genutzt werden kann, in Kombination mit der in Java implementierten Oberfläche *nebula*. Leider ist es auch am aufwendigsten zu installieren, hierzu weiter unten mehr. Dieses Managementsystem bietet, über den Funktionsumfang der anderen hier vorgestellten Applikationen hinaus, noch die oben schon erwähnte Darstellung der Knoten auf einer Weltkarte, sowie eine leistungsfähige XML-basierte Projektverwaltung in der sich Prozessabläufe verschiedener Tasks modellieren und anwenden lassen. Um *plush* aus den Quellen[5] zu bauen, sind zuerst eine Vielzahl von C++-Bibliotheken und ein paar Perl-Module zu installieren[6]. Hierzu sollte unbedingt ein RedHat basiertes System verwendet werden, da einige der benötigten Bibliotheken nur als rpm-Pakete erhältlich sind. Es ist ist auch darauf zu achten, dass die in der Installationsanweisung genannten Versionsnummern benutzt werden und, falls man hinter einer Firewall sitzt, die von *plush* und *nebula* benutzten Ports freischaltet, bzw. die Ports weiterleitet, falls hinter dem Modem noch ein Router hängt.

[5] Hierzu ist anzumerken, daß unter:
http://plush.cs.williams.edu/plush-binaries/plush-bin.tar.gz
zwar ein statisch kompiliertes Binärpaket für Debian erhältlich ist,
das aber leider nicht zum funktionieren zu bewegen war

[6] Installationsanleitung unter:
http://plush.cs.williams.edu/users.html

4 Tests und Resultate

4.1 Vorüberlegungen

Wichtig für die Durchführung des Testszenarios sind ein paar Vorüberlegungen bezüglich der Anforderungen von weichen Echzeitanwendungen an Netzwerkperformance und Dienstgüte. Multimedia-Anwendungen zielen meist darauf ab, größere Datenmengen in eine Richtung zu transportieren, deshalb sind Messungen der Umlaufzeit (*round-trip-time*) eher von untergeordnetem Interesse. Wichtiger ist hier die Messung (soweit dies möglich ist) von Einwegverzögerungen *one-way-delay* und Paketverlustraten, sowie der maximale Durchsatz [Sie05]. Da für Multimedia-Anwendungen, wie Videostreaming, ein erneutes Versenden von verlorengegangen Paketen meist nicht in Betracht kommt, ist hier ein besonderes Augenmerk auf Paketverluste und Paketverlustraten zu legen. Da die Paketverluste einen Bezug zur Größe der versandten Pakete aufweisen, kommt für unsere Tests eine sogenannte *Constant-Bitrate Source* nicht in Betracht, sondern es muss ein Sender benutzt werden, der über die Zeit variable Paketgrößen versendet. Weiterhin wichtig ist der maximal erzielbare Durchsatz, der, wie die Tests gezeigt haben, auf den PlanetLab Knoten weniger durch die Netzwerkschnittstelle, als durch die Auslastung der Knoten begrenzt wird. Da die *slices* innerhalb eines separaten VServers laufen, wirkt sich eine große Anzahl von aktiven VServern negativ auf das Scheduling des eigenen VServer aus. Schließlich ist der Jitter noch von einiger Bedeutung für Multimedia-Anwendungen, da diese meist mit einem kontinuierlichen Datenstrom versorgt werden müssen, was bei hohen Schwankungen in den Laufzeiten zunehmend schwieriger wird. Die Messungen zu den Paketverlustraten und den Laufzeitschwankungen können dazu benutzt werden, in Multimedia-Anwendungen den benötigten Zwischenspeicher (*jitter buffer*) zu dimensionieren. Zu den angestrebten Einwegmessungen ist zu sagen, dass dies natürlich eine synchronisierte Zeit auf Sende- und Empfangsseite vorrausetzt. Die PlanetLab Knoten werden über das *Network Time Protocol (ntp)* synchronisert und die Drift, jedes Knotens von der Systemzeit, ist jederzeit abfragbar und somit der dadurch verursachte Fehler jederzeit bestimmbar. Funktionstüchtige Knoten innerhalb des PlanetLab Verbunds weisen im allgemeinen eine Drift von 0,0 bis 0,2 Sekunden auf. Da dies in Summe mit der über ntp herstellbaren maximalen Uhrengenauigkeit von 1-2 ms, gemessen an einer Einwegverzögerung von 40 ms, einen beträchtlichen Fehler darstellt, sollen zur Korrektur auch Umlaufmessungen zur Bestimmung der Einwegverzögerung durchgeführt werden, insbesondere da vorherige Tests auf den Testknoten mit traceroute und tracepath sehr stabile und sehr symmetrische Routen zwischen den PlanetLab Knoten aufgezeigt haben.

4.1.1 Wahl der Implementationsprache

Für die Implementierung des Experiments wurde aus naheliegenden Gründen Python gewählt. Einerseits ist Python eine robuste Sprache mit integrierter Ausnahmebehandlung und leistungsfähigen Datenstrukturen, andererseits ist alles was zur Entwicklung benötigt wird auf den PlanetLab Knoten schon vorhanden. Zudem wird die ganze Unixsystemschnittstelle über Wrapper- Module direkt angesprochen, so z.B. die Socketschnittstelle. Hier ist durch eine C-Implementierung kein nennenswerter Vorteil zu erwarten.

4.1.2 Einschränkung der Testergebnisse

Da PlanetLab keine Testumgebung im klassischen Sinne ist [PPSB05], können die Ergebnisse nur als grobe Abschätzung der realen Gegebenheiten auf den Knotenrechnern der PlanetLab Umgebung gelten. Umsomehr gilt dies, da bei diesen Test kein Dienst, wie z.B. Sirius, zur bestimmbaren Ressourcenzuteilung genutzt wurde und die Einschätzung des Einfluss der Schedulingmechanismen des VServers schwierig ist. Das der Einfluss des Schedulers, je nach Last auf den einzelnen Knoten mitunter gravierend sein kann, wird an der Wiederholung des Tests deutlich. Ein großer Teil der Nachrichtenverluste in beiden Testläufen beruht auf dem Auslagern des eigenen Vservers und dem damit verbundenen Ablauf des Timeouts beim Kommunikationspartner.

4.2 Durchführung

Die beiden Testläufe wurden mit einer Dauer von jeweils 2 Tagen im Abstand von einem Tag durchgeführt. Es wurden Nachrichten mit einer Länge von 4 Kilobyte bis 64 Kilobyte vom Testclient-Modul versendet. Bis auf das blockierende Empfangen des Server-Moduls wurden alle Socketoperationen mit einem gesetzten Timeout-Wert durchgeführt. Im ersten Testlauf wurde das Timeout auf 10 Sekunden gesetzt, im zweiten auf 5 Sekunden. Für den Test wurden dem benutzten *slice* 43 Knoten hinzugefügt, davon fielen 5 Knoten während des ersten Testlaufs aus. Hierfür wurde kein Ersatz eingebunden und ausgewertet wurden nur die Daten der 38 über den ganzen Testlauf funktionstüchtigen Knoten. Der zweite Testlauf wurde ebenfalls mit diesen 38 Rechnern durchgeführt, die sich während der ganzen Testphase als sehr stabil auszeichneten. Die zu versendenden Nachrichten wurden mit einer 15-stelligen eindeutigen Nachrichtennummer versehen, um bei der Auswertung der *One-Way-Delay* Messung eine Zuordnung zwischen Sender und Empfänger vornehmen zu können. Die Zeitmessung wurde unmittelbar vor dem Senden, bzw. unmittelbar nach Erhalt der vollständigen Nachricht vorgenommen. Zwischen den Testsequenzen wurden Pausen im ersten Fall von 20 Minuten im zweiten Fall von 15 Minuten

eingebaut. Diese Pausen wurden genutzt um die zwischengespeicherten, erhaltenen oder gesendeten Nachrichten in verwertbarer Form in Logdateien zu schreiben. Dies geschah auch mit allen auftretenden Fehlern. Weiterhin wurde vor Beginn, sowie nach Beendigung des Tests die Drift von der PlanetLab Central Zeit aller beteiligter Knoten bestimmt. Hier wurden Werte zwischen 0 und 0,2 Sekunden bestimmt und festgehalten. Die Drift veränderte sich bei den beteiligten Knoten während des Tests bis auf eine Ausnahme nicht, wo die Drift von der PlanetLab Central Zeit gegen Ende des ersten Tests über 15 Sekunden betrug. Die Ergebnisse dieses Knotens wurden für die Bestimmung des *One-Way-Delay* nicht berücksichtigt.

4.3 Ergebnisse

In beiden Experimenten waren die durchschnittliche Umlaufverzögerung mit 0,281 Sekunden, bzw. 0,139 Sekunden für die Einwegverzögerung, im ersten Test und 0,236 Sekunden für den Umlauf, bzw. 0,143 Sekunden für die Einwegverzögerung, im zweiten Test ziemlich hoch. Siehe hierzu auch die Tabellen C.1 und C.2 auf den Seiten 28 und 29. Die besten gemessenen Zeiten lagen hier bei 0,004 bis 0,005 Sekunden für den Umlauf zwischen zwei Knoten, die an das DFN angebunden sind. Ähnlich gute Werte ließen sich zwischen fast allen deutschen und österreichischen Standorten, sowie zwischen den Standorten in den USA bestimmen. Zwischen den Standorten in Europa und USA wurden durchschnittliche Umlaufverzögerungen erzielt und besonders schlecht war die Anbindungung zu südamerikanischen und einigen süd-ost-asiatischen Standorten. Die Bestimmung der Einwegverzögerung scheint zumindest im ersten Testlauf gut funktioniert zu haben hier weichen die Ergebnisse, von denen der zur Kontrolle bestimmten Einwegverzögerungen mithilfe der Umlaufzeit, maximal um 0,01 Sekunden ab. Dahingegen divergieren die Werte im zweiten Testlauf für die gemessene und die berechnete Einwegverzögerung um bis zu 70 ms. Im zweiten Testlauf waren die gemessen Latenzzeiten etwas besser, aber immer noch vergleichsweise hoch. Auffällig ist, dass in keinem der Testläufe eine Nachricht von 64 kB gesendet werden konnte, hier schlug offensichtlich immer der Scheduler zu, lagerte den Prozess während des Sendens aus, so dass der Timeout verstrich. Ungewöhnlich ist auch die hohe Anzahl an Nachrichtenverlusten von über 30 % im Mittel. Hier war zu beobachten, dass der Nachrichtenverlust bei Nachrichten über 32 kB noch einmal anstieg. Eine eine Woche vorher durchgeführte Probemessung lieferte mit Nachrichten der Länge 1 kB sogut wie keine Paketverluste. Eine nach den beiden Testläufen durchgeführte Probemessung mit Nachrichtenlängen über 64 kB ergab ein ähnliches Bild – es wurde keine Nachricht versand.

5 Fazit und Ausblick

PlanetLab selbst wird und wurde von seinen Initiatoren immer als dauerhaftes Experiment angesehen [PACR02] und kann schon aus diesem Grund keine Testumgebung im klassischen Sinne sein, auf der sich reproduzierbare Ergebnisse erzielen lassen [PPSB05]. Auch läßt sich solch ein Ansatz, wegen der vielen gleichzeitig ablaufenden Experimente, der nicht garantierten Ressourcenzuteilung (begrenzt über den Sirius-Dienst) und der Unmöglichkeit ein bestimmtes Set von Knoten dauerhaft zu nutzen, nicht verfolgen, da schon eine Wiederherstellung der Bedingungen des Experiments nicht realisierbar erscheint. Darüberhinaus bietet sich dem PlanetLab Nutzer aber eine, bis dahin nicht vorhandene Möglichkeit, seine Anwendungen unter echten Verteilungsbedingungen auf Großsystemeffekte, Fehlertoleranz und Selbstheilung zu testen, was auch im Umfang des nutzbaren Rechnernetzes so wohl nur in den wenigsten Forschungseinrichtungen möglich wäre. Somit kann PlanetLab kein Ersatz für exakte Messungen unter wohldefinierten Bedingungen sein, aber eine willkommene und wichtige Ergänzung zu diesen.

Nachdem PlanetLab seit seiner Gründung ein stetiges Wachstum erfahren hat, bleibt die zukünftige Entwicklung etwas ungewiss. Zur Zeit wechseln viele europäische Standorte unter die Kontrolle des europäischen PlanetLab Central. Diese von Beginn so gewollte Entwicklung, eines dezentralen und föderalen Aufbaus, birgt in sich das Risiko einer Zersplitterung. Da z.B. die europäische Zentrale unter der Verwaltung des aus EU Mitteln geförderten OneLab steht, ist denkbar das sich hier im Laufe der Zeit europäische Eigeninteressen kristallisieren, die in Inkompatibilitäten zwischen den einzelnen kontinentalen PlanetLab Teilen münden und so eine globale Nutzung irgendwann unmöglich wird. Dem soll durch eine enge Koordinierung entgegengewirkt werden, so dass PlanetLab auch zukünftig weltumspannend genutzt werden kann.

A Kleines PlanetLab Starter Howto

Hier soll ein kurzer Überblick gegeben werden, welche Schritte notwendig sind, um eigene Anwendungen auf PlanetLab Knoten zu transferieren und zu starten. Zudem sollen ein paar nützliche Informationen für den Umgang mit PlanetLab gegeben werden. Für jede der hier genannten Aufgaben existiert eine Vielzahl von Möglichkeiten und Anwendungen um diese zu lösen. Hier[1] soll jeweils nur eine dieser Möglichkeiten vorgestellt werden und Wert darauf gelegt werden, dass der angebotene Lösungsansatz auf vielen verschiedenen Betriebssystemplattformen funktioniert.

Vorbereitung

Zuerst sollte man sich einen Entwicklungs- und Testrechner mit einer Fedora Core 2 oder Core 4 Installation aufsetzen, da auf den PlanetLab Knoten keine *buildtools* vorhanden sind und die Knoten laut AUP (Acceptable Use Policy) auch nicht zum Entwickeln, Testen und Debuggen benutzt werden sollen. Dann muß ein PlanetLab Account angelegt werden. Hierzu geht man auf die PlanetLab Homepage unter `http://www.planet-lab.org`, wählt den Punkt *create an account*, füllt das Formular aus, und wartet auf die Bestätigungsmail. Dann kann der für den eigenen Standort zuständige PI[2] kontaktiert werden und eine Zuordnung des neu angelegten PlanetLab Benutzers zu einem *slice* vornehmen. Nun muß ein SSH Schlüsselpaar im *open-ssh* Format (Schlüssel anderer SSH Varianten wie z.B. *putty* müssen in das *open-ssh* Format umgewandelt werden) erzeugt werden. Nach erfolgtem Login auf der PlanetLab Homepage muß dann unter dem Menüpunkt *My Account* und dort unter *Manage Keys* der erzeugte öffentliche Schlüssel (id_rsa.pub) eingetragen werden. Der private Schlüssel (id_rsa) sollte auf einem USB-Stick ausgelagert werden. Bei längerer Nutzung PlanetLabs sollte man von Zeit zu Zeit ein neues Schlüsselpaar erzeugen.

[1] Hierzu finden sich noch zwei HOWTOS, die in den Umgang mit diversen Projekt-Verwaltungstools einführen. Siehe unter:
`http://www.cs.huji.ac.il/labs/danss/planetlab/PlanetlabProjectHowto.pdf`
bzw. unter:
`http://www.cs.princeton.edu/~jbagdis/planetlab/hello_world.pdf`

[2] *PI - Principal Investigator* ist der für die *slice*- und Benutzerverwaltung Zuständige eines Standortes

Schlüsselerzeugung mit *open-ssh* :

Mit folgendem Aufruf wird ein Schlüsselpaar erzeugt:

```
me@test:~$ ssh-keygen -t rsa -f ~/.ssh/id_rsa
```

ssh-keygen verlangt das Setzen einer Passphrase für den privaten Schlüssel, die zwischen 10 und 30 Zeichen lang sein sollte. Um dann nicht vor jedem ssh-login die Passphrase eingeben zu müssen, oder um *scp* im Batchmode betreiben zu können, kann am Anfang einer Sitzung der *ssh-agent* gestartet und mit *ssh-add* ein Schlüssel übergeben werden:

```
me@test:~$ ssh-agent
me@test:~$ ssh-add ~/.ssh/id_rsa
```

Hier muss dann nur einmal für die aktuelle Sitzung die Passphrase angegeben werden.

Erstes Login:

Nach der Zuordnung eines *slice* zum Benutzer, kann, unter der Annahme der slice Name sei hawh_slice und der Rechner pl1.haw-hamburg.de gehört zum slice, ein erstes login versucht werden mit:

```
me@test:~$ ssh -l hawh_slice -i ~/.ssh/id_rsa pl1.haw-hamburg.de
```

Schlägt der Anmeldeversuch fehl, sollte das Kommando noch einmal mit der Option -v aufgerufen werden. Aus den ausgiebigen Meldungen des ssh-clients kann dann meistens schnell auf die Fehlerursache geschlossen werden. Ein häufiger Grund warum der Anmeldeversuch fehlschlägt, ist das sich der Schlüssel des Knoten geändert hat und somit dem Schlüssel in `~/.ssh/known_hosts` widerspricht. Von der generellen Lösung, wie sie in einigen Tutorials und Howtos propagiert wird, den Wert der Variablen *StrictHostKeyChecking* in `~/.ssh/config` auf no zu setzen, sei dringend abgeraten. Stattdessen können unter `https://www.planet-lab.org/db/nodes/known_hosts.php` die aktuellen Schlüssel abgerufen und mit einem simplen *copy & paste* in die known_hosts-Datei übertragen werden.

Knoten zum Slice hinzufügen:

Nun müssen weitere Knoten dem *slice* hinzugefügt werden. Hier existieren verschiedene Möglichkeiten dies zu bewerkstelligen. Zum einen lassen sich Knoten über die Pla-

netLab Seite nach erfolgtem Login dem eigenen *slice* hinzufügen, oder aber man nutzt *plcsh*, dies jeweils im Zusammenwirken mit einem *Health-Monitoring-Service* wie CoMon oder einer Abfrage der SWORD-Datenbank. Gerade beim Hinzufügen vieler Knoten ist die effektivere Vorgehensweise wohl die Benutzung von *plcsh* in Kombination mit einer SWORD-Abfrage unter http://sword.cs.williams.edu/sword_gui2.html. Das Programm *plcsh* erhält man, entgegen der Informationen im Userguide unter http://www.cs.princeton.edu/~jbagdis/planetlab/hello_world.pdf, mit Hilfe folgenden Aufrufs

```
svn checkout https://svn.planet-lab.org/svn/PLCAPI/trunk PLCAPI
```

in einer aktuellen Variante. Ein einfacher Aufruf von `make` im Quellverzeichnis baut dann die Anwendung zusammen. Die *plcsh* ist eine in Phyton implementierte *shell* für die PlanetLab Central Datenbank. Diese läßt sich auch als vollwertiger Phyton-Interpreter nutzen oder als Modul in eigene Phyton Anwendungen einbinden. Mit der *plcsh* läßt sich die XML-RPC Schnittstelle sehr unkompliziert nutzen, da die teilweise sehr komplexen XML-RPC Aufrufe, der PlanetLab Central API, in einfache Kommandoaufrufe gekapselt sind. Außerdem verfügt die shell über eine Kommandozeilen-History, was die Benutzung effizient gestaltet. Mit Hilfe der SWORD Oberfläche auf oben genannter Seite kann nun eine Abfrage generiert werden, in der die Grenzen für eine Vielzahl von Parametern, wie freier Festplattenplatz, Anzahl benötigter Knoten, Anzahl auf dem Knoten laufender Sliver, durchschnittliche Speicher- und Prozessorauslastung, etc., gesetzt werden können und die als Ergebnis eine Liste mit Knoten zurückgibt, die diese Kriterien erfüllen. Diese Liste kann man nun in eine Datei *hostlist* schreiben. Nun startet man die *plcsh*:

```
me@test:~$ plcsh -h https://www.planet-org/PLCAPI -u pl_user
```

Nachdem man dann sein PlanetLab Passwort eingegeben hat muss man in der *plcsh* nun nur folgendes aufrufen:

```
[pl_user]>>> file = open('hostlist', 'r')
[pl_user]>>> s = file.read()
[pl_user]>>> AddSliceToNode("hawh_slice", s.splitlines())
```

und schon sind die Knoten dem angegebenen Slice hinzugefügt, was mit einem Aufruf von

```
[pl_user]>>> GetNodes()
```

überprüft werden kann. Nutzt man eine der komplexeren Anwendungen mit graphischer Benutzeroberfläche, wie *plush* in Kombination mit *nebula* oder *PlMan*, so lassen sich die Knoten selbstverständlich auch damit zum Slice hinzufügen.

Anwendung auf die Knoten verteilen:

Hat man seine Anwendung fertiggestellt und ausgiebig lokal getestet, taucht die Frage auf, wie man sein Experiment auf die Knoten des eigenen *slice* transferieren soll. Im einfachsten Fall läßt sich dies mit dem *scp* Programm aus *open-ssh* Paket bewerkstelligen. Vorrausgesetzt der *ssh-agent* ist wie oben beschrieben gestartet und der Ordner `tests` soll ins Heimatverzeichnis des Knoten `a_plab_node` des *slice* kopiert werden sieht der Aufruf folgendermaßen aus:

```
me@test:~$ scp -B -i ~/.ssh/id_rsa -r test hawh_slice@a_plab_node:
```

Dies ist natürlich für mehr als fünf Knoten völlig indiskutabel. Abhilfe schafft hier ein kleines Python-Skript.

```
1  #!/usr/bin/python
2  # pscp.py
3
4  import sys
5  import os
6  import commands
7  from optparse import OptionParser
8
9  scp = '/usr/bin/scp'
10 home = os.getenv('HOME')
11 # hier gegebenenfalls den Pfad zu meinem privaten Schluessel
12 # eintragen, alternativ ist das auch als Parameteruebergabe
13 # beim Programmaufruf moeglich
14 defaultKeyFile = home + '/.ssh/id_rsa'
15 # Hier meinen slice Namen eintragen
16 username = hawh_slice
17 # Hier den Pfad zu meinem hostfile angeben
18 # hostfile: Textdatei mit Hostnamen oder IP-Adresse.
19 #           Ein Name pro Zeile
20 hostfile = home + ' /hostfile'
21
22 parser = OptionParser(usage='\n%prog [OPTIONS] (FILE|DIR)')
23 parser.add_option("-r", "--recursive", dest="recursive", default=False,\
24                 help='copy dir recursively or not, default is %default')
25 parser.add_option("-i", "--key-file", dest="keyfile",\
26                 default=defaultKeyFile,\
27                 help='the key-file to be used, default is %default')
28 parser.add_option("-d", "--destination", dest="destination", default='',\
29                 help='the directory the file(s) will be copied to,' +\
30                 'default is %default')
31
32 (options, args) = parser.parse_args()
33
```

```
34 if not(len(args) == 1):
35   parser.print_help()
36   sys.exit(256)
37
38 file2scp=args[0]
39 f = open(hostfile, 'r')
40 s = f.read()
41 hostlist = s.splitlines()
42
43 if not options.recursive:
44   command = scp + ' -B -i ' + options.keyfile +' '+\
45             file2scp +' '+ username +'@'
46 else :
47   command = scp + ' -B -i ' + options.keyfile +' -r '+\
48             file2scp +' '+ username +'@'
49
50 for host in hostlist:
51   cmd = command + host +':'+ options.destination
52   (status, output) = commands.getstatusoutput(cmd)
53   print host
54   print output
55   print status
56   print "\n"
57
58 sys.exit(0)
```

Angenommen das Skript wird in einer Datei namens `pscp.py` innerhalb unseres Pfades gespeichert, der *ssh-agent* ist wie oben beschrieben gestartet und die Variablen im Skript wurden auf unsere Bedürfnisse hin angepasst, dann funktioniert obiger Kopiervorgang auf alle Knoten unseres *slice* folgendermaßen:

```
me@test:~$ pscp.py -r test
```

Parallel Kommandos ausführen:

Als nächstes steht nun an, unser Projekt das wir auf die PlanetLab Rechner transferiert haben, auch zu starten und den Ablauf zu verfolgen. Dies geht wiederum im einfachsten Fall mit der Kommandooption des Programm *ssh* und sieht, unter der Annahme `command` wäre der Aufruf zum Starten unseres Projekts, wie folgt aus:

```
me@test:~$ ssh -i ~/.ssh/id_rsa hawh_slice@a_plab_node command
```

Auch das ist für mehr als eine Handvoll Knoten nicht praktikabel, weshalb auch hier wieder ein kleines Python Skript für Abhilfe sorgt.

```
#!/usr/bin/python
# pcmd.py

import sys
import os
import commands
from optparse import OptionParser

ssh = '/usr/bin/ssh'
home = os.getenv('HOME')
# hier gegebenenfalls den Pfad zu meinem privaten Schluessel
# eintragen, alternativ ist das auch als Parameteruebergabe
# beim Programmaufruf moeglich
defaultKeyFile = home + '/.ssh/id_rsa'
# Hier meinen slice Namen eintragen
username = hawh_slice
# Hier den Pfad zu meinem hostfile angeben
# hostfile: Textdatei mit Hostnamen oder IP-Adresse.
#           Ein Name pro Zeile
hostfile = home + ' /hostfile'

parser = OptionParser(usage='\n%prog [OPTIONS] -c COMMAND')
parser.add_option("-i", "--key-file", dest="keyfile",\
                default=defaultKeyFile,\
                help='the key-file to be used, default is %default')
parser.add_option("-c", "--command", dest="command", default='',\
                help='the command to execute on remote host,' +\
                'default is %default')

(options, args) = parser.parse_args()

if command == '':
  parser.print_help()
  sys.exit(256)

f = open(hostfile, 'r')
s = f.read()
hostlist = s.splitlines()

command = ssh + ' -i ' + options.keyfile +' '+\
          username +'@'

for host in hostlist:
  cmd = command + host +' '+ options.command
  (status, output) = commands.getstatusoutput(cmd)
  print host
```

```
47    print output
48    print status
49    print "\n"
50
51  sys.exit(0)
```

Wieder unter der Vorraussetzung, dass das Skript unter dem Namen `pcmd.py` innerhalb des Pfades abgespeichert wurde, der *ssh-agent* wie oben beschrieben gestartet und die Variablen im Skript auf unsere Bedürfnisse hin gesetzt wurden, sieht der Aufruf zum Starten unseres Projekts auf allen Knoten unseres *slice* wie folgt aus:

```
me@test:~$ pcmd.py -c "command"
```

Die Anführungszeichen sind Pflicht, falls unser Kommando mit Argumenten aufgerufen wird, da die Argumente sonst vom *OptionParser* verschluckt werden.

B Ergänzende Bilder

Hier als Beleg für die Anmerkung auf Seite 8 die tabellarischen Übersichten des CoVisualize Dienstes des CoDeen Projekts.

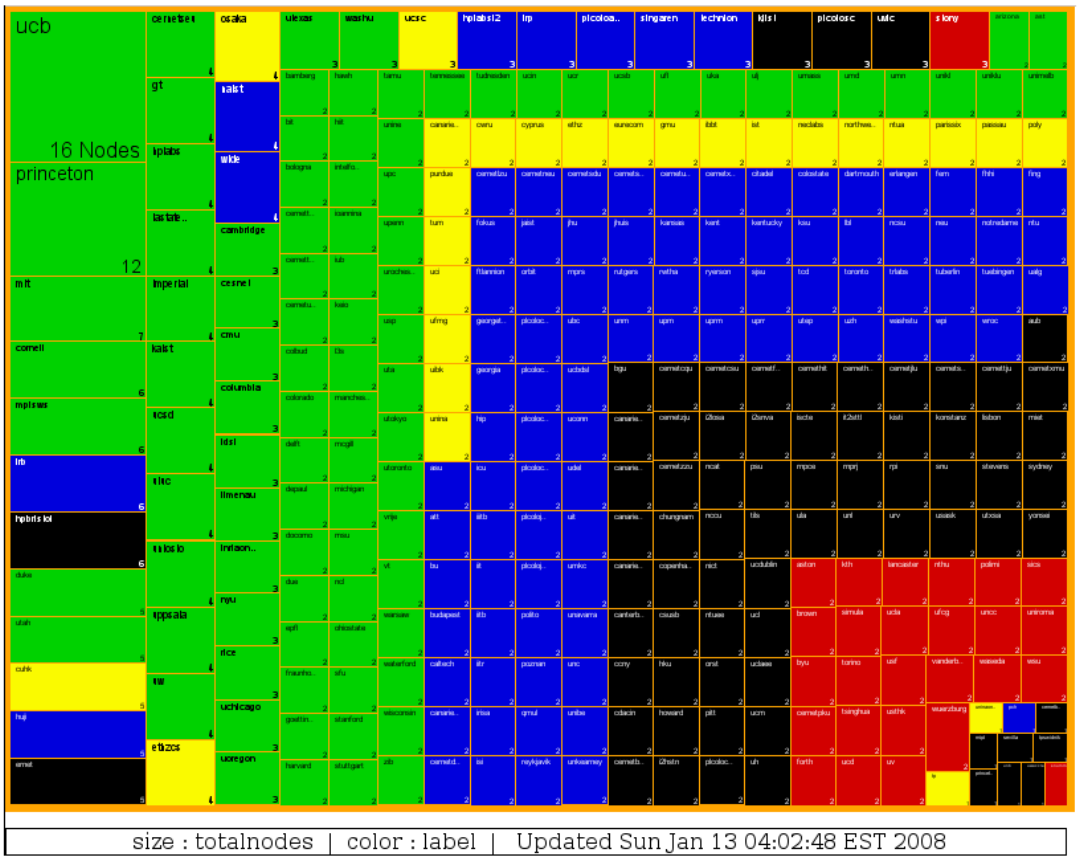

Abbildung B.1: Zustand der PlanetLab Knoten und ihrer Standorte am 13. Jan. 2008

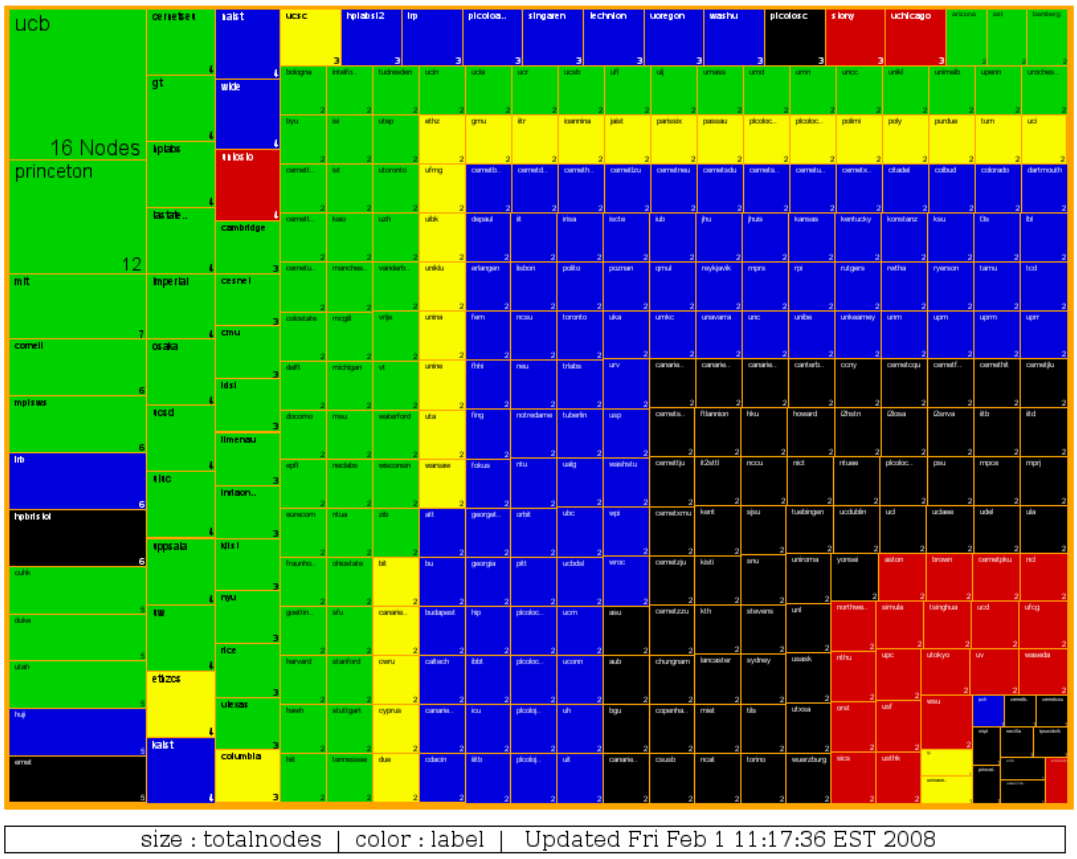

Abbildung B.2: Zustand der PlanetLab Knoten und ihrer Standorte am 1. Feb. 2008

Legende:

- Grün: Standort ist voll funktionstüchtig.
- Gelb: Standort verfügt nur über eingeschränkt nutzbare Knoten.
- Blau: Standort verfügt über nutzbare Knoten, nutzt selbst aber PlanetLab nicht.
- Schwarz: Standort ist verfügt nur über tote Knoten, nutzt selbst aber PlanetLab nicht.
- Rot: Standort verfügt nur über tote Knoten und nutzt darüber hinaus aktiv PlanetLab Ressourcen.

C Tabellen zu den Messergebnissen

Die gemessene Zeiten in Tabelle C.1 und Tabelle C.2 sind in Sekunden angegeben.
Im folgenden bedeuten: rtt: round-trip-time und owd: one-way-delay.

Nachrichtenlänge	rtt	owd	rtt/2
4 kB	0.28185	0.13913	0.14092
8 kB	0.34440	0.18709	0.17220
12 kB	0.33310	0.16894	0.16655
16 kB	0.35749	0.18226	0.17874
20 kB	0.36783	0.18979	0.18392
24 kB	0.38763	0.20139	0.19381
28 kB	0.40984	0.21491	0.20492
32 kB	0.43066	0.22415	0.21533
36 kB	0.46285	0.23729	0.23142
40 kB	0.47574	0.24507	0.23787
44 kB	0.50306	0.26006	0.25153
48 kB	0.51595	0.26638	0.25797
52 kB	0.54027	0.27973	0.27013
56 kB	0.55411	0.28691	0.27705
60 kB	0.58783	0.30305	0.29392
64 kB	–	–	–

Tabelle C.1: Erster Testlauf über 48 Stunden

Nachrichtenlänge	rtt	owd	rtt/2
4 kB	0.23608	0.14396	0.11804
8 kB	0.29209	0.20941	0.14605
12 kB	0.27556	0.17895	0.13778
16 kB	0.29267	0.19043	0.14633
20 kB	0.30795	0.19647	0.15398
24 kB	0.32374	0.20678	0.16187
28 kB	0.33584	0.21547	0.16792
32 kB	0.35212	0.22299	0.17606
36 kB	0.37936	0.23941	0.18968
40 kB	0.38240	0.24651	0.19120
44 kB	0.39414	0.25333	0.19707
48 kB	0.39983	0.25971	0.19991
52 kB	0.40148	0.27017	0.20074
56 kB	0.41072	0.27442	0.20536
60 kB	0.42316	0.27906	0.21158
64 kB	–	–	–

Tabelle C.2: Zweiter Testlauf über 48 Stunden

Nachrichtenlänge	Test 1				Test 2			
		Verlust				Verlust		
	Gesendet	Hin	Rück	Total	Gesendet	Hin	Rück	Total
4 kB	47891	7102	2299	9401	103779	17338	7908	25246
8 kB	47891	7880	2927	10807	103779	19360	9541	28901
12 kB	47891	8067	3159	11226	103779	19877	9926	29803
16 kB	47891	8233	3304	11537	103779	20151	10380	30531
20 kB	47891	8664	3047	11711	103779	21508	9610	31118
24 kB	47891	8770	3196	11966	103546	21662	9711	31373
28 kB	47891	8887	3277	12164	103514	21939	9995	31934
32 kB	47891	8954	3323	12277	103455	22181	10183	32364
36 kB	47891	13877	3153	17030	103404	33839	9639	43478
40 kB	47891	14059	3254	17313	103368	34363	10134	44497
44 kB	47891	14131	3331	17462	103329	34593	10162	44755
48 kB	47891	14667	3606	18273	103282	35982	10687	46669
52 kB	47891	16711	3748	20459	103190	40388	10581	50969
56 kB	47891	16709	3821	20530	103146	40480	10642	51122
60 kB	47890	17771	4467	22238	103098	42646	12029	54675
64 kB	0	0	0	0	0	0	0	0
Total	718364	174482	49912	224394	1552227	426307	151128	577435

Tabelle C.3: Nachrichtenverluste während beider Testläufe

Literaturverzeichnis

[Chu03] Chun, Brent N. *pssh-HOWTO*, 2003. URL http://www.theether.org/pssh/docs/0.2.3/pssh-HOWTO.pdf.

[Hua05] Huang, Mark. *VNET: PlanetLab Virtualized Network Access*. *Technischer Bericht PDN–05–029*, PlanetLab Consortium, 2005. URL http://www.planet-lab.org/files/pdn/PDN-05-029/pdn-05-029.pdf.

[MPF+05] Muir, Steve, Peterson, Larry u. a.. *Proper: Privileged Operations in a Virtualised System Environment*. In *Proceedings of the USENIX 2005 Annual Technical Conference*, Seiten 367–370. Anaheim, California, 2005. URL http://www.planet-lab.org/files/pdn/PDN-04-022/pdn-04-022.pdf.

[MS07] Mahlmann, Peter und Schindelhauer, Christian. *Juristische Situation*. In *Peer-to-Peer-Netzwerke*, eXamen.press, Seiten 252–257. Springer-Verlag, Berlin, 2007. ISBN 978-3-540-33991-5. ISSN 1614-5216.

[OAPV04] Oppenheimer, David, Albrecht, Jeannie, Patterson, David und Vahdat, Amin. *Distributed resource discovery on PlanetLab with SWORD*, 2004. URL http://sword.cs.williams.edu/pubs/worlds04.pdf. First Workshop on Real, Large Distributed Systems (WORLDS '04).

[PACR02] Peterson, Larry, Anderson, Tom, Culler, David und Roscoe, Timothy. *A Blueprint for Introducing Disruptive Technology into the Internet*. In *Proceedings of HotNets–I*. Princeton, New Jersey, 2002. URL http://www.planet-lab.org/files/pdn/PDN-02-001/pdn-02-001.pdf.

[PHM+04] Peterson, Larry, Hartman, John u. a.. *Evolving the Slice Abstraction*. *Technischer Bericht PDN–04–017*, PlanetLab Consortium, 2004. URL http://www.planet-lab.org/files/pdn/PDN-04-017/pdn-04-017.pdf.

[plW08] *World50.png*, 2008. URL http://www.planet-lab.org/generated/World50.png.

[PPPW04] Park, KyoungSoo, Pai, Vivek S., Peterson, Larry und Wang, Zhe. *CoDNS: Improving DNS Performance and Reliability via Cooperative Lookups*. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI 2004)*. San Francisco, CA, 2004. URL http://www.cs.princeton.edu/nsg/papers/codns_osdi_04/paper.pdf.

[PPSB05] Peterson, Larry, Pai, Vivek, Spring, Neil und Bavier, Andy. *Using PlanetLab for Network Research: Myths, Realities, and Best Practices*. *Technischer Bericht PDN–05–028*, PlanetLab Consortium, 2005. URL http://www.planet-lab.org/files/pdn/PDN-05-028/pdn-05-028.pdf.

[PR06] Paterson, Larry und Roscoe, Timothy. *The Design Principles of PlanetLab*. *Operating Systems Review*, Band 40(1):Seiten 11–16, 2006. URL http://www.planet-lab.org/files/pdn/PDN-04-021/pdn-04-021.pdf.

[Ros05] Roscoe, Timothy. *The PlanetLab Platform*. In Ralf Steinmetz und Klaus Wehrle (Herausgeber), *Peer-to-Peer Systems and Applications*, Nummer 3485 in LNCS, Seiten 567–581. Springer-Verlag, Berlin, 2005. ISBN 978-3-540-29192-3. ISSN 0302-9743.

[RPKW03] Roscoe, Timothy, Peterson, Larry, Karlin, Scott und Wawrzoniak, Mike. *A Simple Common Sensor Interface for PlanetLab*. *Technischer Bericht PDN–03–010*, PlanetLab Consortium, 2003. URL http://www.planet-lab.org/files/pdn/PDN-03-010/pdn-03-010.pdf.

[Sie05] Siemens, Eduard. *Verteiltes Messen der Dienstgüte und Netzwerk-Performance in IP-Netzen*. Doktorarbeit, 2005. URL www.tib.uni-hannover.de/spezialsammlungen/dissertationen/hannover/uebersichten/diss_05.pdf.