Projektarbeit

Fachbereich Informatik

Eventbasierte Simulation von Middleware Plattformen

Betreuer: Mathias Dalheimer, AG Verteilte Algorithmen

Technische Universität Kaiserslautern

Alexander Petry *

Juni 2006

*E-Mail: <petry@itwm.fhg.de>, Matrikelnummer: 345466

Erklärung an Eides Statt

Ich versichere hiermit, dass ich die vorliegende Projektarbeit mit dem Thema „Eventbasierte Simulation von Middleware Plattformen" selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

(Ort, Datum)	(Name, Unterschrift)

In dieser Arbeit stelle ich eine eventbasierte Simulationsumgebung für *Calana* vor. *Calana* ist eine Grid-Scheduler Architektur, die mittels Auktionen die Aufträge an die einzelnen Ressourcen verteilt. Dabei werden die Anbieter der Ressourcen durch Agenten repräsentiert, die während einer Auktion auf die Aufträge bieten können. Ziel dieses Schedulers ist es, die *Wünsche der Benutzer* zu berücksichtigen. Die Wünsche werden durch Präferenzen realisiert — zum Beispiel die Zeit-Präferenz, die besagt, dass der Auftrag *möglichst* schnell ausgeführt werden soll.

Als erstes werde ich einen kurzen Überblick über das Grid-Umfeld geben. Dabei spreche ich verschiedene Scheduling-Verfahren (sowohl für Cluster, als auch für Grids) an und werde die verwendeten Präferenzen genauer erläutern. In dieser Arbeit gehe ich von reinen *computational Grids* aus — in späteren Erweiterungen des Simulator können jedoch zusätzliche Komponenten (Netzwerke, Datenbanken, etc.) hinzugefügt werden.

Als nächstes werden das *Calana*-Protokoll und seine Implementierung in Java ausführlich beschrieben. Mit diversen Experimenten, die sowohl eine kleine Zahl an Anbietern, als auch ein Grid mit 50 Anbietern umfassen, soll gezeigt werden, dass der Scheduler das gestellte Ziel (die Benutzer-Präferenzen zu erfüllen) erreichen kann. Es zeigte sich dabei, dass die Average Response Time für die Benutzer, die schnell ihre Ergebnisse wollen, *unabhängig von der Load* ist. Die Average Response Time ist in dieser Arbeit die maßgebliche Metrik, um Simulationsläufe zu vergleichen.

Zuletzt stelle ich anhand der *Koallokation* eine Möglichkeit vor, das System zu erweitern ohne bestehende Teile verändern zu müssen. Dabei zeigt sich der Vorteil des agentenbasierten Ansatzes, da nur ein neuer (jedoch spezieller) Agent dem System hinzugefügt werden musste. Experimente, die mit verschiedenen Anteilen an Koallokation (dem gleichzeitigen Reservieren verschiedener Ressourcen) in den Workloads durchgeführt wurden, zeigten, dass auch hier die ART für die Benutzer, die schnell ihre Ergebnisse benötigen, *unabhängig* war.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einführung

Wie in "The Anatomy of the Grid" ([10]) beschrieben wird, konzentriert sich "Grid computing" auf die koordinierte *gemeinsame Nutzung* von hochgradig verteilten Ressourcen, innovative Applikationen und auf den high-performance Bereich. Die diversen Ressourcen, wie zum Beispiel Rechenzeit, Speicherplatz oder Netzwerkbandbreite, werden von verschiedenen Organisationen oder Individuen bereitgestellt. Unterschied zu Cluster-Systemen ist die extrem lose Kopplung über das Internet, die globale Verteilung der Ressourcen und die Heterogenität der teilnehmenden Systeme (Cluster, Workstations, Desktop PCs, verschiedene Betriebssysteme, unterschiedliche Dateisysteme, diverse Datenbanken etc.). All das macht es erforderlich, dass eine Zwischenschicht (auch Middleware genannt) geschaffen wird, welche eine einheitliche Sicht auf die Ressourcen bietet (z.B. Globus Toolkit). Die gemeinsame Nutzung der Ressourcen muss zwischen Anbieter und Konsument wohldefiniert sein. Insbesondere bedeutet das, dass folgende Aspekte geregelt werden müssen (aus [10]):

- was genau wird für die gemeinsame Nutzung bereitgestellt
- wer darf die gemeinsamen Ressourcen benutzen und
- unter welchen Bedingungen werden Ressourcen gemeinsam nutzbar.

Institutionen oder Individuen, die anhand dieser Bedingungen gemeinsame Ressourcen teilen, werden "virtual organizations" (VO) genannt.

Die Bedingungen fallen auf beiden Seiten höchst unterschiedlich aus, zum Beispiel können auf Seiten des Anbieters *Kosten* für die Benutzung anfallen (was kostet beispielsweise eine Minute Rechenzeit) — diese können sogar abhängig von der Tageszeit variieren. Tagsüber, wenn auch "interner" Bedarf besteht, ist die Benutzung der Ressource(n) kostbar, nachts hingegen eventuell günstiger. Ein Anbieter kann genausogut vorschreiben, dass die Ressourcen nur zu einem gewissen Zeitraum zur Verfügung stehen (z.B. nachts gibt es nicht genutzte Desktop-PCs).

Als Benutzer hingegen kann man zum Beispiel die Bedingung stellen, dass der Auftrag möglichst *schnell* ausgeführt wird (auch wenn dabei höhere Kosten entstehen). Diese Bedingungen werde ich in dieser Arbeit als *Präferenzen* bezeichnen und unterscheide zusätzlich zwischen Anbieter- und Benutzer-Präferenzen. Man kann sich noch wesentlich komplexere Bedingungen überlegen, die von beiden Seiten gestellt werden können (spezielle Software wird vorausgesetzt, nur zertifizierte Software wird zugelassen, etc.).

Um solche Präferenzen erfüllen zu können, ist es wichtig gewisse Voraussagen über einen Auftrag treffen zu können: der Benutzer muss bei Rechenaufträgen zum Beispiel wissen, wie lange er eine Ressource ungefähr reservieren

möchte. Anhand dieser Information kann ein Anbieter die eventuell anfallenden Kosten für die Reservierung berechnen und im Voraus planen, welche Ressource er zur Verfügung stellen kann (wie wir gleich sehen werden, ist das Wissen über die ungefähre Dauer einer Reservierung auch für das Scheduling außerordentlich hilfreich).

1.1. Überblick über die Arbeit

Diese Arbeit beschäftigt sich mit der Simulation eines Schedulers für Grid-Umgebungen unter zu Hilfenahme des Calana-Protokolls ([4]). Dieses Protokoll modelliert anhand von Agenten einen Marktplatz, auf dem mit Ressourcen nach dem Prinzip einer Auktion gehandelt wird. Jeder Anbieter wird durch einen Agenten (nähere Informationen zu Agenten findet man in [11, 21, 12]). Für jeden Auftrag wird durch einen *Broker* (wiederum ein Agent) eine Auktion eröffnet, bei der jeder Anbieter teilnehmen kann, indem er *Gebote* einreicht. Diese Gebote werden anschließend vom Broker bewertet und das *beste* Gebot bekommt den Zuschlag und somit den Auftrag. Wie genau diese Gebote aussehen und wie die Bewertungsfunktion darauf definiert wird, erkläre ich später.

Als Benutzer-Präferenz beschränke ich mich in dieser Arbeit vorerst auf eine Gewichtung von Preis und Zeitpunkt der Beendigung des Auftrags. Ein Benutzer kann demnach einen Auftrag zum Beispiel möglichst schnell beendet sehen wollen (er legt weniger auf den Preis wert). Was ich zeigen möchte ist, dass mit diesem Scheduler die Präferenzen erfüllt werden können. Genauer gesagt soll der Auftrag eines Benutzers, für den der Preis eine große Rolle spielt (hohe Preis-Präferenz), möglichst unterhalb des durchschnittlichen Preises ausgeführt werden.

Zuerst werde ich allerdings einige Begriffe erörtern. Danach wird auf *Calana* eingegangen und wie es implementiert wurde. Hierbei wird auch die Implementierung des Simulator-Kerns eine Rolle spielen. Anschließend wird Koallokation betrachtet und wie sie anhand des Protokolls implementiert werden kann (es zeigt sich unter anderem die Flexibilität des Protokolls und des gewählten Agent-basierten Ansatzes). Koallokation ist ein wichtiger Teilaspekt für einen Grid-Scheduler, da dem Benutzer neben dem Ausführen normaler Programme zusätzlich die Möglichkeit eröffnet wird, *gleichzeitig* verschiedene Ressourcen zu reservieren. Wenn ein Benutzer beispielsweise zwei Prozesse, die jeweils 4 CPUs benötigen, parallel ausführen möchte, kann er mittels Coallocation diese Reservierung (zweimal vier Prozessoren) vornehmen. Hierbei spielen “advanced reservations” ([16]) eine Rolle.

1.2. Scheduling

Unter Scheduling[1], auch Zeitablaufsteuerung genannt, versteht man die Erstellung eines Ablaufplanes (auch "schedule" genannt), der Prozessen zeitlich begrenzt Ressourcen zuweist. Dabei können die Ressourcen beispielsweise Prozessoren, Speicher oder Bandbreite in einem Netzwerk sein.

Verschiedene Verfahren werden in [19, Kapitel 2.5] genauer beschrieben, ich werde hier werde nur zwei grundlegend verschiedene Ansätze ansprechen: zum einen das *interaktive Scheduling* (im Desktop-Bereich anzutreffen) und zum anderen das *nicht-interaktive Scheduling* (im Server- bzw. Cluster-Bereich). Ein anderes Verfahren ist Echtzeit-Scheduling, welches aber für diese Arbeit keine Rolle spielt.

Die einzelnen Verfahren verfolgen spezielle Ziele, die teilweise in einem Konflikt miteinander stehen. Einige dieser Ziele sind zum Beispiel:

- Maximierung des Durchsatzes, d.h. die Anzahl der bearbeiteten Aufträge pro Zeiteinheit maximieren
- Minimierung der Antwortzeit, d.h. den Zeitabstand zwischen Eintreffen und Beendigung des Jobs minimieren
- Maximierung der Auslastung der Ressourcen
- Minimierung des zusätzlichen Overheads, der durch den Scheduler verursacht wird (insbesondere bei Kontextwechseln)
- faire Behandlung aller Prozesse, d.h. jeder Prozess darf irgendwann einmal rechnen

Beispielsweise stehen die Maximierung des Durchsatzes und die Minimierung der Antwortzeit in Konflikt miteinander.

Schnelle Antwortzeit bedeutet, dass ein neuer Prozess möglichst sofort die CPU erhält — bei *interaktiven* Programmen ist das von großer Bedeutung: wenn ich beispielsweise auf einem UNIX System `ls` eintippe, möchte ich die Auflistung des aktuellen Verzeichnisses ohne merkliche Verzögerung erhalten. Sollte ich der einzige Benutzer auf dem System sein, ist das nicht weiter schwierig, hat jedoch ein anderer Benutzer gerade ein sehr rechenintensives Programm gestartet, dann möchte ich mein Ergebnis trotzdem (vergleichsweise) schnell erhalten (es ist natürlich anzunehmen, dass es etwas länger dauern wird, da das System überlastet ist).

Dies erreicht man, indem man ein zeitliches Multiplexen der vorhandenen CPU(s) einführt. Bei einem Prozesswechsel wird allerdings kostbare Zeit verbraucht (man muss die Register der CPU sichern etc.), so dass sich der Durchsatz unweigerlich verringert — darin besteht der Konflikt.

[1] engl.: Zeitplanerstellung

Interaktives Scheduling

Auf einem Desktop-System wird eine verhältnismäßig kleine Anzahl Prozessoren und Benutzer verwaltet. Das Ziel des Schedulers ist es, die Antwortzeit der Prozesse zu minimieren und alle Prozesse des Systems fair zu behandeln. Fair bedeutet, dass jeder Prozess irgendwann einmal die CPU zugeteilt bekommt und nicht "ausgehungert wird". Hierbei wird den Prozessen für eine gewisse Zeitspanne (Zeitscheibe) der Zugriff auf einen Prozessor gewährt. Diese Strategie nennt man "Round - Robin Strategie" (allgemein: *preemptive scheduling*, Prozesse werden also von "außen" unterbrochen und es wird ihnen die CPU entzogen) und ist in Abbildung 1.2 dargestellt.

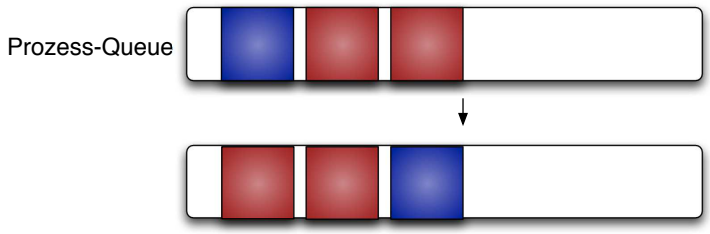


Abbildung 1.1.: Die Prozesse werden hier durch farbige Quadrate repräsentiert, der *blaue* Prozess ist aktuell im Besitz der CPU, bei Ablauf seiner Zeitscheibe wird ihm die CPU entzogen und es darf der nächste Prozess die CPU benutzen (der *rote* Prozess direkt dahinter). Der alte Prozess wird bei der Round-Robin Strategie an das Ende der Warteschlange gestellt (daher auch der Name dieses Verfahrens).

Aufgrund der ständigen Wechsel von lauffähigen (nicht blockierten) Prozessen (so genannte Kontextwechsel) entsteht ein nicht zu verachtender Overhead (der Scheduler muss interne Datenstrukturen aktualisieren, darunter fällt auch das Sichern der Prozessorregister), außerdem erkaltet der Cache, was zu einem kurzfristigen Performance-Einbruch führt. Die Zeitscheibe sollte demnach sehr viel größer gewählt werden, als ein Kontextwechsel an Zeit beansprucht, ansonsten wird die meiste Zeit damit verbracht, zwischen den Prozessen hin und her zu schalten, als tatsächliche Fortschritte (im Sinne von: Vorankommen der Prozesse) zu erzielen.

Man kann diese Strategie zusätzlich noch verbessern, indem man das tatsächliche Prozessverhalten mit einbezieht (Feedback) — beispielsweise kann man verfolgen, ob ein Prozess seine Zeitscheibe immer komplett aufbraucht (CPU intensiv) oder ob er sie häufig früher beendet (viel I/O, interaktive Prozesse), anhand dieser Information kann man die Prozesse dann verschiedenen Warteschlangen zuordnen, die verschieden lange Zeitscheiben benutzen, dadurch lässt sich der Overhead durch die Kontextwechsel reduzieren.

Nicht-interaktives Scheduling

Unter nicht-interaktivem Scheduling versteht man das Scheduling von Prozessen, die *nicht* von Benutzereingaben abhängig sind. Diese Prozesse bestehen aus sehr langen Berechnungs-Phasen, in denen sie ausschließlich auf internen Daten rechnen und aus I/O-Phasen, in denen zum Beispiel Daten nachgeladen oder gesichert werden. Wie bereits erwähnt führen Kontextwechsel zu Einbußen bei der Performance, da zum einen der Wechsel von einem Prozess zu einem anderen Zeit kostet und zum anderen der Cache erkaltet. Das Cacheprinzip geht von "Referenzlokalität" aus (wenn auf ein Datum an Adresse X zugegriffen wird, ist es sehr wahrscheinlich, dass in naher Zukunft auch auf ein Datum in der näheren Umgebung von X zugegriffen wird). Bei einem Prozesswechsel muss der gesamte Cache als ungültig markiert werden. Die Folge davon ist, dass jeder Speicherzugriff des neuen Prozesses direkt zu mindestens einem langsamen Hauptspeicherzugriff (im schlimmsten Fall sogar zu einem Festplattenzugriff) führt, anstatt zu einem schnellen Cachezugriff.

Abhilfe schaffen so genannte **Batch Scheduling Verfahren**. Bei diesen wird einem Prozess solange der Zugriff auf die CPU(s) gewährt, bis der Prozess entweder die CPU(s) selbst wieder freigibt oder eine blockierende I/O Operation anstößt. Ein Benutzer muss also die Möglichkeit haben, Aufträge in einer Warteschlange ablegen zu können (bei Clustersystemen wird dies oft durch einen Master-Knoten realisiert, auf dem sich die Benutzer interaktiv einloggen können). Der Scheduler arbeitet dann auf dieser Warteschlange und wählt nach einer Strategie den nächsten Prozess zur Bearbeitung aus. Eine mögliche Strategie ist zum Beispiel **F***irst* **C***ome* **F***irst* **S***erve* (FCFS), hierbei werden die Prozesse einfach der Reihe nach abgearbeitet (implizite Ordnung der Aufträge anhand ihrer Ankunftszeiten).

Dieses Verfahren soll den Durchsatz und die Auslastung des gesamten Systems maximieren. Optimal wäre es, wenn zusätzlich die Antwortzeit minimiert wird, aber dieses Ziel verträgt sich nicht mit den anderen beiden. Eine Verringerung der Antwortzeit ist jedoch möglich, dazu wird allerdings Hilfe durch den Benutzer nötig. Wenn man für einen Prozess nur einen Slot fester Größe reserviert (Anzahl CPUs $\times$ ungefähre Laufzeit des Prozesses), besteht die Möglichkeit, dass spätere Aufträge schon früher ausgeführt werden.

Das Vorziehen späterer Aufträge wird *Backfilling* ([8, 22, 14, 13, 7]) genannt. Es lässt sich dadurch einerseits die Auslastung erhöhen und andererseits lassen sich die Antwortzeiten verringern.

Das Verfahren beruht darauf, dass "breite" Prozesse warten müssen, bis genügend Ressourcen (CPUs) zur Ausführung stehen. Der Scheduler kann nun weitere Aufträge starten und somit die Lücke(n) füllen — einzige Bedingung dabei ist, dass der wartende Auftrag nicht verzögert wird. In Abbildung 1.2 ist ein kleines Beispiel dazu dargestellt.

Das Verfahren funktioniert nur, wenn die Reservierungen fest eingehalten werden. Ein Scheduler **muss** Aufträge abbrechen, wenn sie die für sie reservierte Zeit überschreiten. Eine Folge aus diesem Verhalten ist, dass Benutzer meist viel zu lange Blöcke reservieren, wie sich dies auf das Backfilling Verfahren auswirkt, kann man unter anderem in [22] und [17] nachlesen.

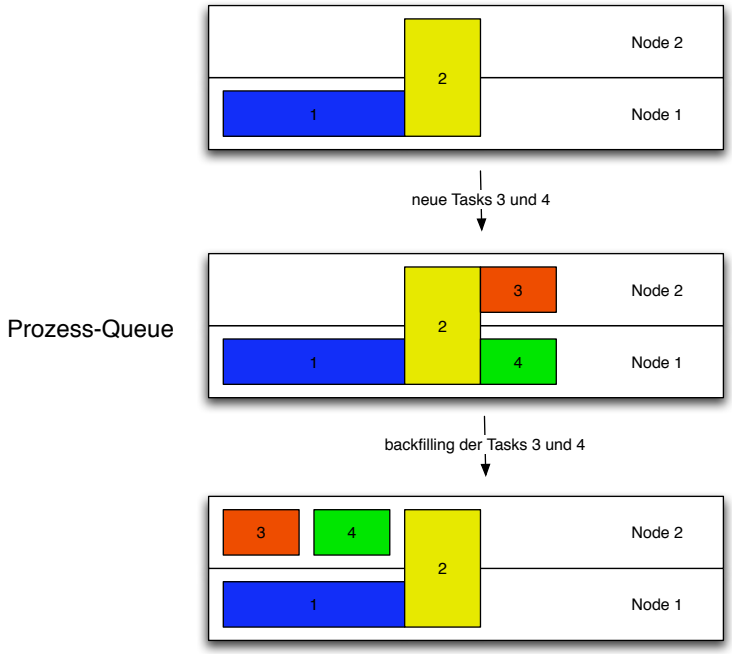


Abbildung 1.2.: Hier ist die Warteschlange eines Schedulers dargestellt, der zwei Nodes verwaltet. Task 2 muss warten bis Task 1 beendet ist, da er beide Nodes benötigt. Wenn die Tasks 3 und 4 gestartet werden sollen, stellt der Scheduler fest, dass sie in die Lücke zwischen Task 1 und Task 2 passen und kann "Backfilling" anwenden.

1.3. Ressourcen

Scheduler verwalten wie oben bereits angesprochen diverse Ressourcen (Prozessoren, Speicher, etc.). Ein Grid-Scheduler muss darüber hinaus sogar ganze Computer oder Cluster als Ressourcen betrachten.

Grid

Ein Grid kann man sich als ein stark verteiltes, lose gekoppeltes und dynamisches Netzwerk von Ressourcen vorstellen. Anfänglich entstanden ist der Begriff “Grid” Mitte der 90er Jahre. Damals war es ein Vorschlag für eine Infrastruktur zum Zusammenschluss verteilter Computersysteme zum Forschen und Entwickeln ([10]).

> Eine mögliche Anwendung besteht zum Beispiel darin, dass ein Forscher Nutzungszeit für ein Radio-Teleskop beantragt, zeitgleich eine Netzwerkverbindung mit genügend Bandbreite und Rechenzeit auf einem leistungsstarken Cluster. Die Daten, die das Teleskop sammelt werden über das Netzwerk auf den Cluster kopiert, um dort von einem Auswertungsprogramm bearbeitet zu werden. Abbildung 1.3 zeigt eine Auswahl möglicher Komponenten, die in einem Grid zur Verfügung stehen.

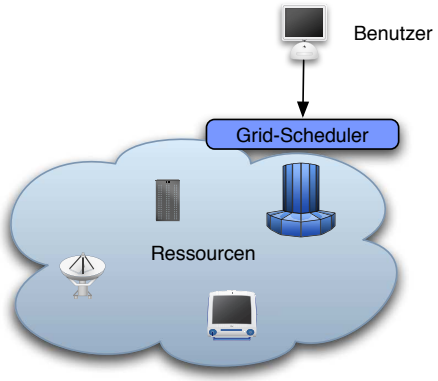


Abbildung 1.3.: Komponenten eines Grids

Mittlerweile bieten sich auch noch ganz andere Anwendungsfelder: dank der breiten Verfügbarkeit von Hochgeschwindigkeitszugängen in das Internet, kann ein Unternehmen beispielsweise nicht verwendete Rechenzeit (die Desktop-PCs der Mitarbeiter werden nachts oder während der Mittagspause nicht oder nur wenig benutzt) in einem Grid zur Berechnung von Simulationen bereitstellen. Die Benutzung der Kapazitäten muss nicht unentgeltlich erfolgen, das heißt, dass das Unternehmen eventuell sogar Gewinn erwirtschaften kann. Da ein Grid dynamisch strukturiert ist, kann ein PC, der doch wieder

von seinem Besitzer beansprucht wird, aus dem Pool der verfügbaren Ressourcen entfernt werden — für den Mitarbeiter ist das "Zweitleben" seines PCs völlig transparent.

In [20] werden verschiedene Ansätze für Grid-Scheduler vorgestellt und miteinander verglichen. Ich möchte hier nur einen kurzen Überblick über die möglichen Strukturen geben, die ein Grid annehmen kann.

Zentrale Scheduler besitzen genügend Informationen, um die Aufträge optimal auf die angeschlossenen Ressourcen zu verteilen. Diese zentral gespeicherten Informationen müssen allerdings immer aktuell gehalten werden, da sich die Struktur des Grids dynamisch ändern kann (es können Ressourcen hinzukommen oder wieder entfernt werden). Der Scheduler kann zu einem Engpass im System führen, da es bei einer Trennung zwischen ihm und den angeschlossenen Ressourcen zu Ausfällen kommen kann.

Um diese Probleme zu vermeiden, wird meist eine **hierarchische Struktur** eingesetzt, diese besteht zwar immer noch aus einem zentralen Scheduler, allerdings werden die Jobs zu *lokalen Schedulern* (zum Beispiel MAUI oder PBS) weitergeleitet. Der Vorteil dieses Aufbaus liegt darin, dass unterschiedliche Regeln für das globale und lokale Scheduling existieren können, der Nachteil von reinen zentralen Schedulern beleibt allerdings erhalten.

Dezentrale Scheduler besitzen mehrere Punkte, an denen Aufträge in das System eingebracht werden können, dadurch entfällt der "single point of failure" und der mögliche Flaschenhals bei der Kommunikation. Es wird immer zuerst versucht, den Auftrag lokal auszuführen (also dort, wo der Auftrag in das System gekommen ist). Ist dies nicht möglich, wird der Auftrag entweder direkt an einen anderen Scheduler weitergereicht, oder er wird in einer zentralen Warteschlange zwischengelagert. Aus diesem Pool können sich dann die lokalen Scheduler bedienen.

Der Hauptunterschied zwischen Gridcomputing und Metacomputing besteht darin, dass beim Gridcomputing zusätzlich zu reinen Recheneinheiten Ressourcen anderen Typs gemeinsam genutzt werden können: Sensoren, Datenbanken, Speicherplatz, Bandbreite in einem Netzwerk, Geräte zur Visualisierung von Simulationen, etc. ([9]).

Die von mir in dieser Arbeit verwendeten Ressourcen beschränken sich auf "computational resources" (*Node* und *Cluster*). Im folgenden Abschnitt, werde ich diese kurz erläutern.

Node

Eine Node ist ein einzelner Rechner, der eine überschaubare Anzahl an Prozessoren enthält — zum Beispiel mein Desktop-PC, bestehend aus einem Intel Pentium 4 Prozessor mit 2.4GHz und 1GB Hauptspeicher. Im folgenden werde ich nicht mehr zwischen CPU und Node unterscheiden, sondern nur noch von

Nodes sprechen (eine Node mit nur einer CPU entspricht dann einer einzelnen CPU). Das ist für diese Arbeit unproblematisch, da es für einen Benutzer in einem Grid nicht von Bedeutung ist, ob sein Auftrag, der nur eine physikalische CPU benötigt, von einem Desktop-PC mit einer CPU oder als einer von vielen auf einem Cluster ausgeführt wird.

Cluster

Cluster[2] bestehen aus zwei oder mehreren *Nodes*, die entweder zentral durch einen oder mehrere ausgezeichnete Nodes (so genannte Master-Nodes) verwaltet werden; Abbildung 1.4 zeigt einen möglichen Aufbau für einen Cluster. Weniger häufig anzutreffen ist eine dezentral organisierte Struktur, da man hierbei auf Konsistenz zwischen den einzelnen Nodes achten muss (die Reservierungs-Tabelle muss überall gleich aussehen).

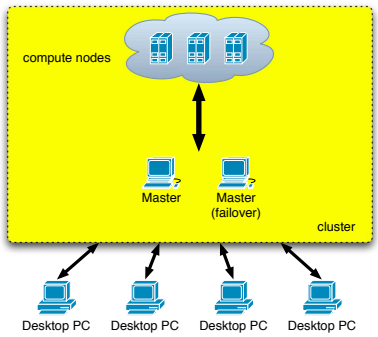


Abbildung 1.4.: Beispiel einer Cluster Architektur

Der Unterschied zum oben bereits erwähnten Metacomputing liegt in der Verbindungsstruktur. Cluster sind eng gekoppelt, entweder durch Standard Ethernet-Komponenten, oder hochspezialisierte Verbindungen wie zum Beispiel Myrinet oder InfiniBand®. Diese speziellen Verbindungen haben diverse Vorteile: höhere Übertragungsraten, geringere Latenz, Auslagerung von Netzwerkfunktionalität (Protokoll-Stacks zum Beispiel) in die Hardware und somit Entlastung des Betriebssystems; sie kosten aber auch entsprechend mehr.

Das Reservieren von Nodes für eine bestimmte Zeit muss durch den Benutzer selbst initiiert werden — als Schedulingstrategien bieten sich natürlich Batch-Verfahren an, da man Durchsatz und Auslastung maximieren möchte.

1.4. Präferenzen

Herkömmliche Grid-Scheduler versuchen nur, die Anfragen zu verteilen und optimieren bezüglich Auslastung und Durchsatz. Was aber ist mit speziellen Wünschen sowohl von den Anbietern, als auch von den Nutzern? Ein Grid-

[2] zum Beispiel das *Beowulf Project* ([1]).

Scheduler sollte also wenn möglich auch nach den Wünschen "seiner" Benutzer optimieren.

In dieser Arbeit habe ich einen solchen *Wunsch* als "Präferenz" modelliert. Wenn ein Benutzer den Wunsch hat, seinen Auftrag kostengünstig abarbeiten zu lassen, bedeutet das, dass er eine *hohe Präferenz auf den Preis* hat (der Preis ist ihm sehr wichtig). Wir betrachten hier nur zwei solcher Präferenzen: die *Preis-Präferenz* und die *Zeit-Präferenz*. Beide werden als prozentuale Gewichtung angegeben, die in der Summe 1 ergeben.

Jetzt haben wir die Wünsche der Benutzer modelliert, es fehlen allerdings noch die der Anbieter.

Hier kann man genauso argumentieren und sagen, dass ein Anbieter zum Beispiel lieber kurze als lange Aufträge annimmt. Dieser Fall tritt auf, wenn die angebotenen Ressourcen nur kurzfristig zur Verfügung stehen (beispielsweise die Desktop-PCs innerhalb einer Mittagspause).

Die Modellierung besteht nun darin, dass jedem Anbieter eine Kostenfunktion zugeordnet wird, die aus einem Grundpreis p_b und einem Preis pro Zeiteinheit p_t besteht. Der Preis für die Benutzung der Ressource errechnet sich dann als $p(\tau) = p_b + \tau * p_t$. Je nach Wahl der Parameter erhalten wir zum Beispiel für kurze Jobs einen hohen oder einen niedrigen Preis. Wenn der Anbieter kurze Jobs bevorzugt, definieren wir einen niedrigen Grundpreis, dafür aber einen hohen Preis pro Zeiteinheit (Minutenpreis).

Eine solche Modellierung deckt natürlich sehr viele Dinge nicht oder nur unzufriedenstellend ab, ein anderer Ansatz würde allerdings den Rahmen dieser Arbeit sprengen. Weitere mögliche Anforderungen sind zum Beispiel:

- der Anbieter soll möglichst zuverlässig sein (wenige durch ihn abgebrochene Aufträge)
- politische Fragestellungen bezüglich des Anbieters (der Auftrag darf nur bei einer Untermenge aller Anbieter ausgeführt werden)
- Art des Betriebssystems
- grundsätzliche Ausführbarkeit des Auftrags (werden zum Beispiel spezielle Softwareversionen vorausgesetzt?)

Die nächste Frage, die sich stellt ist: *Wie kann ein Scheduler diese Wünsche beachten?* Ein erster Ansatz beruht auf der zentralen Struktur herkömmlicher Grid-Scheduler. Um Präferenzen beachten zu können muss man die zentralen Daten, auf denen der Scheduler arbeitet um zusätzliche Informationen erweitern (die Kostenfunktion jedes Anbieters, Preis-Präferenz des Auftraggebers etc.). In unserem Fall ist das noch überschaubar, aber was passiert, wenn es sehr viele verschiedene Anbieter gibt, mehr als nur eine Kostenfunktion im Spiel ist oder sich das Verhalten des Anbieters verändert?

Ein möglicher Ansatz, der diese Probleme angeht, wird in [4] vorgestellt und liefert gleichzeitig die Grundlage für diese Arbeit. Es handelt sich dabei um einen hierarchischen, agentenbasierten Scheduler. Das gesamte System beruht

auf Auktionen, bei denen (aus Sicht des Benutzers) der Meistbietende den Zuschlag für den Auftrag erhält. Die Auktionen werden von einem *Broker* geleitet und jeder Anbieter hat die Möglichkeit auf die Aufträge zu bieten.

Die zentrale Instanz benötigt keine näheren Informationen über die einzelnen Anbieter (er muss sie nur kennen und mit ihnen kommunizieren können). Der Broker muss in der Lage sein, die Gebote anhand der Benutzer-Präferenz zu bewerten und somit in eine Ordnung zu bringen. Ein Anbieter (repräsentiert durch einen Agenten) vermerkt in seinem Gebot alle relevanten Informationen (hier: Preis und Zeit). Ein Anbieter ist nicht dazu gezwungen, ein Gebot abzugeben.

Diese Herangehensweise beinhaltet eine große Flexibilität: die Kostenfunktion muss zum Beispiel über die Zeit nicht konstant bleiben, sondern kann von der Tageszeit, der aktuellen Last oder sogar von der Marktsituation abhängen. Der Broker ist von diesen Veränderungen unabhängig. Dem Anbieter steht es vollkommen frei, wie sich sein Repräsentant (der Agent) verhält, so lange er sich an ein definiertes Protokoll (siehe Kapitel 2) hält. In Abschnitt 1.5 wird das Protokoll benutzt, ohne es zu verändern, um dem gesamten System mehr Funktionalität zu verleihen, nämlich Koallokation — dabei war es nicht nötig, den Broker oder andere Agenten zu verändern.

Diese hohe Flexibilität erkauft man sich allerdings durch einen erhöhten Kommunikationsaufwand, der zwischen den einzelnen Komponenten (Scheduler und Anbietern) stattfinden muss.

1.5. Koallokation

Koallokation bedeutet in unserem Fall, dass man für einen Auftrag mehrere unabhängige Gruppen von Ressourcen gleichzeitig benötigt — die Anfrage besteht demnach aus mehreren Teil-Aufträgen, die gleichzeitig gestartet werden müssen. Eine Voraussetzung hierfür sind “advance reservations” ([16]). In Kapitel 3 gehe ich genauer darauf ein, inwiefern Koallokation in dieser Arbeit berücksichtigt wurde.

2. Calana

Das folgende Kapitel beschäftigt sich mit "Calana", einem hierarchischen und auf Agenten basierenden Ansatz für einen Grid-Scheduler [4].

2.1. Motivation

Halten wir uns nocheinmal das Beispiel mit dem Forscher aus Abschnitt 1.3 vor Augen und betrachten die Benutzung des Clusters etwas näher.

> Nehmen wir an, der Forscher möchte die Ergebnisse seiner Beobachtungen in einer Publikation veröffentlichen und *benötigt die Ergebnisse rechtzeitig*. Nehmen wir weiterhin an, er arbeite an einem renommierten Institut und es stünde ihm genügend Geld für seine Forschungszwecke zur Verfügung.
>
> Genausogut kann man den Forscher durch einen Studenten ersetzen. Diesem steht viel weniger Geld zur Verfügung und er muss sich daher mit einem günstigeren (wahrscheinlich langsameren) Anbieter zufrieden geben.
>
> Bei der Reservierung der Rechenzeit verfolgen beide unterschiedliche Ziele, wodurch sich die in Frage kommenden Anbieter eingrenzen lassen. Ein Grid-Scheduler sollte diese Präferenzen beachten und den Auftrag dem aus Sicht des Benutzers am besten geeigneten Anbieter zuteilen.

Die meisten Ansätze verteilen die Aufträge nur nach Kriterien wie zum Beispiel: grundsätzliche Ausführbarkeit, Maximierung der Auslastung und des Durchsatzes. Dalheimer et al. beschreiben in [4] mit *"Calana"* einen agentenbasierten Ansatz, der versucht, die Präferenzen zu beachten.

Um die Benutzer-Präferenzen beachten zu können, wird die Verteilung eines Auftrages anhand einer Auktion entschieden. Teilnehmer der Auktionen sind die Anbieter, die durch jeweils einen Agenten ([11, 21]) repräsentiert werden.

Warum Agenten? Ein agentenbasierter Ansatz ist sehr **flexibel**, da jeder Anbieter das Verhalten "seines" Agenten verändern und an die lokalen Bedürfnisse anpassen kann. Wie wir später sehen werden, kann man das System mit "virtuellen" Agenten erweitern, um weitere Funktionalität zu implementieren.

Diese Flexibilität erkauft man sich durch ein mitunter sehr komplexes Protokoll, um die Kommunikation mit den Agenten zu regeln. Damit folgt auch ein

eventuell hohes Nachrichtenaufkommen, welches die Auslastung des Netzwerks erhöht.

Warum Auktionen? Durch die Auktionen bildet das System einen Marktplatz nach, auf dem mit Ressourcen und Aufträgen gehandelt wird. Damit der Scheduler die Benutzer-Präferenzen erfüllen kann, muss er aus allen Anbietern "den besten" auswählen und dies geht am einfachsten über eine Auktion.

Anhand der Auktionen lässt sich direkt das folgende Problem umgehen: Viele Grid-Scheduler erwarten, dass die angeschlossenen Ressourcen *exklusiv* dem Grid zur Verfügung stehen. Das ist aber nicht unbedingt im Sinne des Besitzers der Ressource. Bei Calana kann der Anbieter lokale Aufträge direkt der Ressource über den lokalen Scheduler zuweisen. Diesen Scheduler benutzt auch der Agent, um Gebote zu erstellen, so dass hier kein Interessenkonflikt auftritt.

Ich vermute allerdings, dass sich durch die Verwendung der Auktionen die Auslastung des gesamten Systems nicht optimieren lässt (im Gegensatz zu anderen Grid-Schedulern). Wenn man sich dazu folgendes Szenario vorstellt: Alle Benutzer möchten ihre Aufträge günstig berechnen, die Zeit spielt keine Rolle; unter den Anbietern gibt es einen, der seine Ressourcen für unter geringen Kosten zur Verfügung stellt. Dieser Anbieter würde jede Auktion gewinnen und somit die anderen Anbieter ausstechen. Der Nebeneffekt ist, dass seine Ressourcen ausgelastet werden, die der anderen aber gar nicht. Die erzielte Auslastung hängt somit entscheidend von den Präferenzen ab. Es stellt sich mir die Frage:

> *"Was ist wichtiger — die Zufriedenheit der Kunden oder die Auslastung der Ressourcen?"*

Ich würde sagen, es kommt ganz darauf an und man sollte ein sinnvolles Mittelmaß finden.

2.2. Architektur

In diesem Abschnitt werde ich den in [4] beschriebenen Ansatz näher erläutern und auf das verwendete Kommunikationsprotokoll eingehen.

Das System besteht aus drei unterschiedlichen Komponenten, die zusammen das Bild in Abbildung 2.1 ergeben.

Workload Manager Über diese Schnittstelle hat ein Benutzer die Möglichkeit, seinen Auftrag in das System zu bringen. Es kann mehr als nur eine solche Instanz geben, da das spätere Verteilen der Aufträge durch einen oder mehrere Broker erfolgt.

Broker Ein Broker leitet für jeden Auftrag eine Auktion. Dabei werden von jedem Agent Gebote eingeholt und bewertet.

Agent Jeder Anbieter besitzt einen eigenen Agenten, der für ihn Gebote abgeben kann. Der Agent ist auch für die Kommunikation mit dem lokalen

Scheduler verantwortlich. Es obliegt dem Anbieter, wie sich sein Agent verhält.

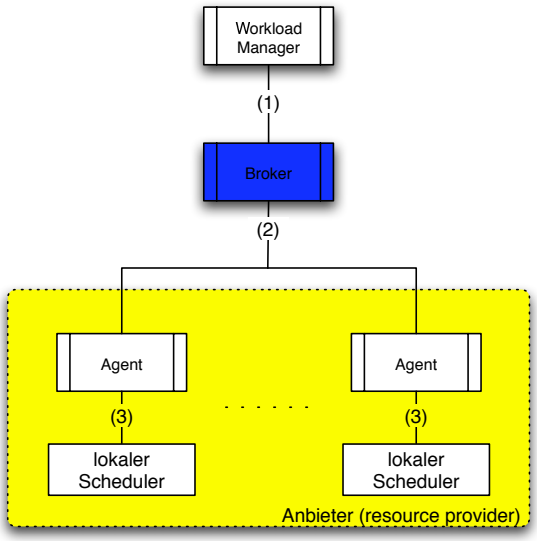


Abbildung 2.1.: Calana System-Architektur

Überblick über den Ablauf

Das System funktioniert grob wie folgt: Ein Benutzer spezifiziert seinen Auftrag, dabei gibt er neben dem auszuführenden Programm auch noch die erwartete Laufzeit ("walltime"), die Anzahl der zu reservierenden Nodes und seine Präferenz an — eine so genannte Jobbeschreibung. Die Präferenz ist in unserem Fall eine Gewichtung von Preis und Zeitpunkt der Beendigung des Auftrages (in Zukunft "finish time" genannt). Die Dauer und die Anzahl der Nodes werden für die lokalen Scheduler benötigt, damit sie die Reservierung möglichst effizient platzieren und eventuell Backfilling (siehe Abschnitt 1.2 auf Seite 11) anwenden können. Diese Beschreibung sendet der Benutzer an den Workload Manager (zur Vereinfachung gehe ich in Zukunft von nur einer Instanz aus).

In **Schritt 1** der Abbildung 2.1 sendet der Workload Manager seinerseits die Beschreibung des Auftrages an den Broker. Der Broker ist nun dafür zuständig einen Anbieter auszuwählen, dem der Auftrag zugeteilt werden soll.

Dazu eröffnet der Broker in **Schritt 2** eine Auktion für den Auftrag und sendet im Zuge dessen eine Anfrage an alle registrierten Agenten (Anbieter). Zu dieser Anfrage gehören die Anzahl und die Art der zu reservierenden Ressourcen und die Dauer, für die sie reserviert werden sollen, nicht aber das auszuführende Programm.

Ein Agent kann auf diese Anfragen mit beliebig vielen Geboten antworten, er kann auch gar kein Gebot abgeben, wenn er den Auftrag nicht annehmen

möchte oder ihn nicht annehmen kann (wenn beispielsweise zu wenig Ressourcen vorhanden sind).

Um ein Gebot erstellen zu können, muss der Agent zuerst in Erfahrung bringen, wann der Auftrag frühestens gestartet werden kann. Dazu beauftragt er in **Schritt 3** seinen lokalen Scheduler, eine *vorläufige Reservierung* vorzunehmen. Auf Grund der Daten, die der Agent von seinem Scheduler erhalten hat, kann er den Preis und die voraussichtliche finish time berechnen.

Da ein Agent die Möglichkeit hat, mehrere Gebote abzugeben, kann er beispielsweise ein günstiges Gebot und ein teures Gebot abgeben; ersteres wird dafür allerdings eine viel spätere finish time beinhalten als das zweite Gebot. Worin genau sich die Gebote unterscheiden und ob überhaupt mehr als ein Gebot abgegeben wird, liegt vollkommen in den Händen des Agenten und hängt von den jeweiligen (Anbieter-)Präferenzen ab.

Der Broker sammelt alle Gebote, die zu einer Auktion gehören und bewertet sie. Zur Bewertung wertet er die Funktion 2.1 aus. Diese verwendet Preis und Zeit jedes Gebots und gewichtet sie anhand der Benutzer-Präferenzen. Das Ergebnis ist eine positive reelle Zahl, die anschließend zum Ordnen der Gebote benutzt wird.

$$v(b) = w_p * \frac{b_p}{max_{\tilde{b} \in B}\{\tilde{b}_p\}} + w_t * \frac{b_t}{max_{\tilde{b} \in B}\{\tilde{b}_t\}} \tag{2.1}$$

Mit den folgenden Variablenbelegungen:

- B ist die Menge aller Gebote zu einer Anfrage
- b ist das zu bewertende Gebot
- w_p ist die Preis-Präferenz des Benutzers
- w_t ist die Zeit-Präferenz des Benutzers (es gilt: $w_t = 1 - w_p$)
- b_p ist der Preis des Gebots b
- b_t ist die finish time des Gebots b

Das aus Sicht des Benutzers beste Gebot[1] gewinnt die Auktion. In diesem Fall das Gebot, welches v maximiert. Der zugehörige Agent erhält eine Bestätigung und zusätzliche Informationen, insbesondere das auszuführende Programm.

Der soeben beschriebene Ablauf soll nun genauer spezifiziert werden. Allerdings müssen wir uns noch Gedanken über ein paar wichtige Aspekte machen. Zum Beispiel sollte es sowohl dem Benutzer, als auch dem Anbieter zu jeder Zeit möglich sein abzubrechen, das heißt, entweder noch bevor ein Vertrag zustande gekommen ist — also der Auftrag tatsächlich einem Anbieter zugewiesen wurde — oder danach. Bei einem Abbruch nachdem der Vertrag zustande gekommen war, fallen natürlich Kosten an, die von der abbrechenden Seite übernommen werden müssen. Es ist wichtig, dass diese Informationen an einer zentralen Stelle (beim Broker zum Beispiel) gesammelt werden, so dass ein Accounting stattfinden kann.

[1] bei mehreren gleich bewerteten Geboten, kann man beliebige Heuristiken einsetzen, um ein Gebot auszuwählen

2.3. Protokoll

Der folgende Abschnitt beschreibt das Calana-Protokoll. Dazu beschreibe ich zuerst die Modularisierung des Protokolls und anschließend werde ich ausgewählte Szenarien anhand von *Message Sequence Charts* verdeutlichen und auf den genauen Ablauf eingehen.

2.3.1. Modularisierung

Das Protokoll ist in insgesamt zwei kleinere Protokolle unterteilt. Die eine Hälfte beschreibt die Kommunikation zwischen Workload Manager und Broker. Der Benutzer steht dabei in direktem Kontakt mit dem Workload Manager (beispielsweise durch ein Web-Portal). Die andere Hälfte regelt die Kommunikation zwischen Broker und Agenten.

Der Broker nimmt demnach eine "Doppelrolle" ein, indem er das Bindeglied zwischen den zwei Welten darstellt. Der Vorteil davon ist, dass man den Teil "hinter" dem Broker (aus Sicht des Benutzers) beliebig ersetzen kann, ohne dass der Benutzer oder der Workload Manager etwas davon mitbekommt.

Als abkürzende Schreibweisen führe ich folgende Bezeichnungen ein:

Kürzel	Bedeutung
C2B	Abkürzung für "Consumer-to-Broker", also die Kommunikation zwischen "Nutzer" (Workload Manager) und Broker "Anbieter"
C2BConsumer	Nutzerseite der C2B Kommunikation
C2BProvider	Anbieterseite der C2B Kommunikation
B2A	Abkürzung für Broker-to-Agent, also die Kommunikation zwischen Broker und Agent (allgemein: Agent)
B2AConsumer	Nutzerseite der B2A Kommunikation
B2AProvider	Anbieterseite der B2A Kommunikation

Tabelle 2.1.: Nomenklatur

2.3.2. Message Sequence Charts

Ich werde nun Schritt für Schritt wichtige Teile das Protokolls beschreiben und die Abläufe dabei mittels *Message Sequence Charts* (MSCs) verdeutlichen. Eine genaue Beschreibung findet man in [3] beziehungsweise im Anhang ab Seite 49. Die Doppelrolle des Brokers habe ich in den nachfolgenden MSCs weggelassen und nur die B2AConsumer-Seite dargestellt. In der Implementierung besteht die Doppelrolle fast ausschließlich darin, intern die entsprechenden Zustandsübergänge zu veranlassen.

Die dargestellten Zustandsübergänge beziehen sich immer auf **einen Auftrag**, wenn also in einem MSC eine Instanz (Agent) "terminiert", dann bedeu-

tet dies lediglich, dass der endliche Automat für diesen einen Auftrag terminiert, nicht der Agent selbst. Bei einer Implementierung ist also stets darauf zu achten, dass ein neuer Automat initialisiert werden muss, wenn ein Auftrag eintrifft.

Die in diesem Abschnitt relevanten Nachrichtentypen sind in Tabelle 2.2 mit einer kurzen Erläuterung aufgelistet.

Nachricht	Bedeutung
BookingReq	vom Consumer zum Provider, beinhaltet alle Informationen, um einen Job zu beschreiben.
AuctionBid	wird von den Agenten zum Broker gesendet und enthält alle für ein Gebot relevanten Informationen (Preis, finish time, etc.)
AuctionAccept	dient der Mitteilung an den Agent, dass er die Auktion gewonnen hat
AuctionDeny	dient der Mitteilung an den Agent, dass er die Auktion verloren hat und alle vorläufig reservierten Ressourcen freigeben kann.
Booked	wird verwendet, um anzuzeigen, dass eine Reservierung vorgenommen werden konnte.
Confirm	Bestätigung durch den Consumer (C2B, B2A), so dass die Reservierung aktiv wird.
Close	dient der Mitteilung, dass ein Auftrag abgeschlossen wurde.
CloseAck	Bestätigung zu einer Close Nachricht
ConsumerCancel	wird versendet, wenn ein Benutzer abbrechen möchte.
ProviderCancel	wird versendet, wenn ein Anbieter abbrechen möchte.

Tabelle 2.2.: Verwendete Nachrichten

Beginn einer Auktion

In Abbildung 2.2 wird der Ablauf des Protokolls dargestellt, der entsteht, sobald ein neuer Auftrag eintrifft.

Der Benutzer sendet seine Jobbeschreibung an den Workload Manager, der daraufhin eine *BookingReq*-Nachricht an den angeschlossenen Broker schickt. Der Broker broadcastet[2] alle nötigen Informationen an alle ihm bekannten Agenten, dazu wird wieder eine BookingReq-Nachricht verwendet.

Am Ende dieses Schrittes befindet sich die B2A Seite (also alle Agenten und der Broker) im Zustand *StAuctionRunning*. Eine Auktion läuft eine gewisse Zeitspanne, so dass jeder Anbieter die Möglichkeit hat, mehrere Gebote abzugeben. Es kann aber genauso gut passieren, dass keiner der Anbieter den

[2] besser: multicastet

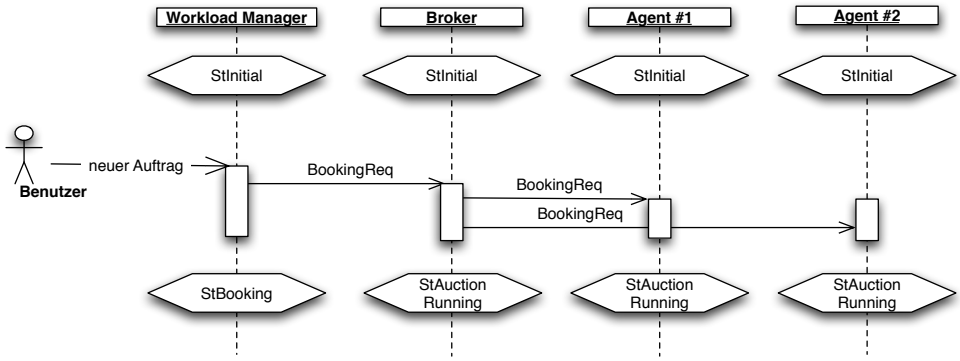


Abbildung 2.2.: Beginn einer Auktion

Auftrag annehmen möchte oder annehmen kann. In einem solchen Fall ist die Auktion fehlgeschlagen und der Auftrag kann nicht ausgeführt werden. Für den Broker ist es in diesem Fall unentscheidbar, ob die Anbieter nicht wollten oder nicht konnten. Eine typische Ursache für diesen Fall ist allerdings, dass ein Benutzer einen Job ausführen möchte, der mehr Nodes benötigt als maximal vorhanden sind.

Auktion war erfolgreich

In den Abbildungen 2.3 und 2.4 wird das erfolgreiche Abschließen einer Auktion dargestellt. Man kann erkennen, dass nur genau ein Agent gewinnt und demzufolge eine *AuctionAccept*-Nachricht erhält. Allen anderen Agenten wird mittels einer *AuctionDeny*-Nachricht mitgeteilt, dass sie die Auktion verloren haben.

Bei Erhalt einer *AuctionDeny*-Nachricht kann ein Agent die vorläufige Reservierung, die er mittels seines lokalen Schedulers vorgenommen hat, abbrechen. Außerdem kann er sein Verhalten der aktuellen Marktsituation anpassen, da in der *AuctionDeny*-Nachricht auch Informationen über den Gewinner und vor allem über sein Gebot stehen. Dieser Aspekt fällt allerdings unter die vertraglichen Regelungen, die eventuell zwischen Broker und Agent existieren können.

Man kann in den Abbildungen erkennen, dass der Broker, nachdem er die Agenten über ihr Abschneiden bei der Auktion informiert hat, eine *Booked*-Nachricht an den Workload Manager sendet. In dieser Nachricht sind alle relevanten Informationen über das von ihm ausgewählte Gebot enthalten, dazu gehören Preis, finish time und Informationen über die reservierte Hardware.

Der Benutzer hat an dieser Stelle die Möglichkeit, dem Gebot zuzustimmen oder es abzulehnen. Die Bestätigung (*Confirm*-Nachricht) wird einfach von “oben” nach “unten” (vergleiche auch Abbildung 2.1) weitergereicht.

Alle noch aktiven Instanzen[3] befinden sich am Ende dieses Schrittes im Zustand *StConfirmed* — der Vertrag ist demnach zustande gekommen.

[3]Workload Manager, Broker und der Agent, der gewonnen hat

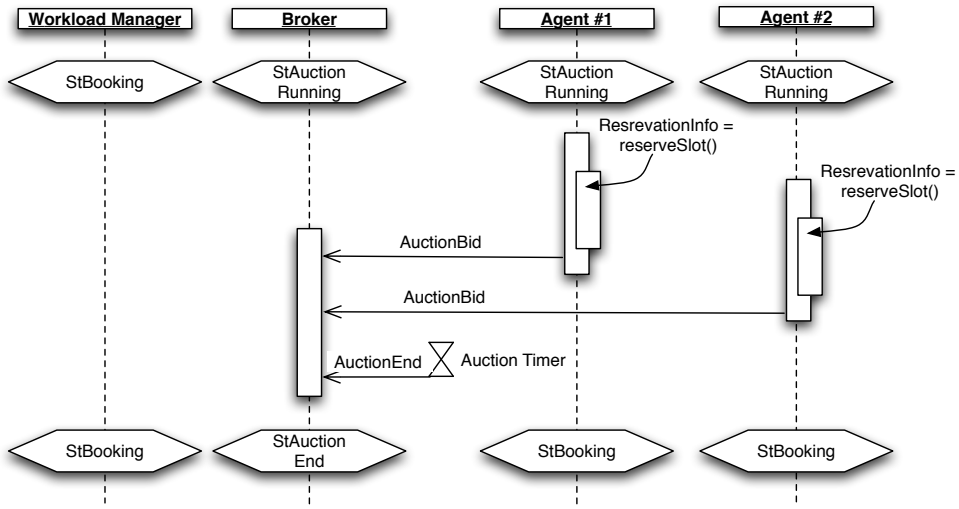


Abbildung 2.3.: Erfolgreiche Auktion (Teil 1/2)

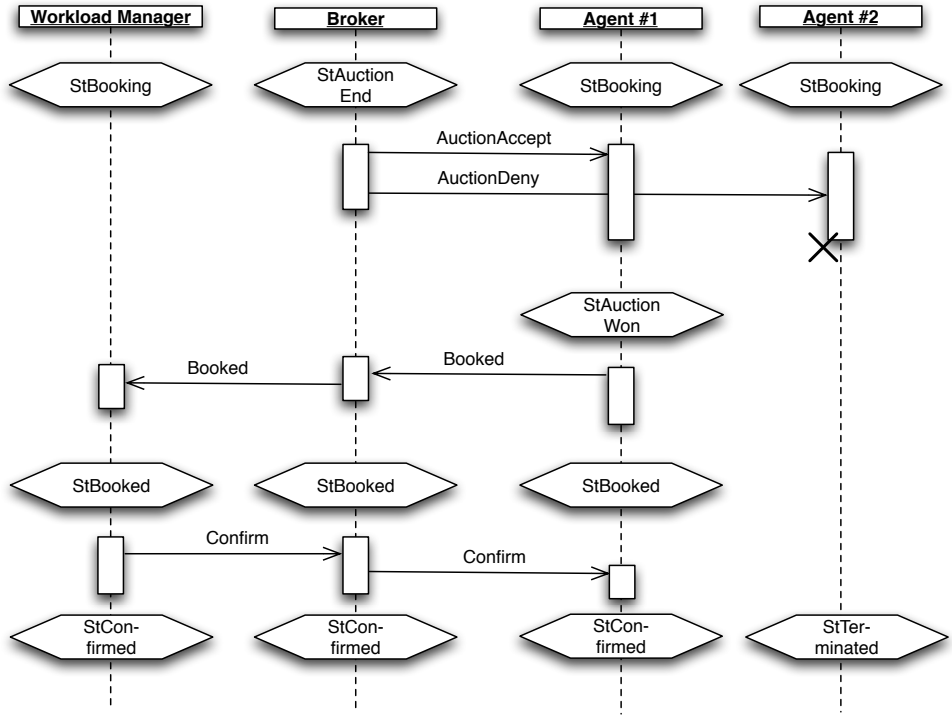


Abbildung 2.4.: Erfolgreiche Auktion (Teil 2/2)

Auktion ist fehlgeschlagen

Wie bereits beschrieben, kann es passieren, dass eine Auktion nicht erfolgreich durchgeführt werden kann. In diesem Fall (Abbildung 2.5) hat keiner der Agenten ein Gebot abgegeben, so dass dem Broker nichts anderes übrigbleibt, als dem Benutzer mitzuteilen, dass sein Auftrag nicht zugewiesen werden konnte; dazu sendet der Broker eine *BookingRejected*-Nachricht an den Workload Manager. Der Workload Manager bestätigt dies daraufhin mit einer entsprechenden *BookingRejectedAck*-Nachricht und terminiert.

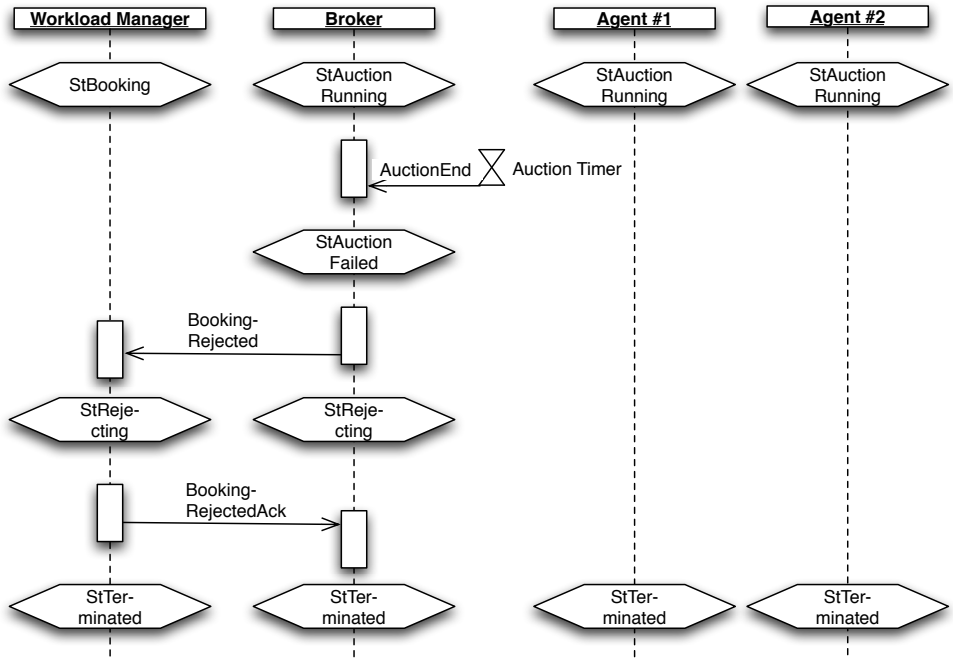


Abbildung 2.5.: Fehlgeschlagene Auktion

Erfolgreiche Beendigung eines Auftrages

Wenn eine Auktion erfolgreich durchgeführt und der Auftrag an einen der Anbieter übermittelt wurde, warten die einzelnen Komponenten nur noch darauf, dass der Auftrag entweder abgebrochen oder ordentlich beendet wird. In Abbildung 2.6 kann man sehen, was passiert, wenn ein Job erfolgreich bearbeitet wurde.

Der ausführende Agent bekommt von seinem lokalen Scheduler mitgeteilt, dass der Auftrag ordnungsgemäß ausgeführt werden konnte. Daraufhin versendet der Agent eine *Close*-Nachricht — bestehend aus Exitcode und diversen statistischen Daten — an den Broker.

Dieser sendet die Nachricht an den Workload Manager und bestätigt dem Agenten mittels einer *CloseAck*-Nachricht, dass er die Nachricht erhalten hat. Der Workload Manager geht ebenso vor.

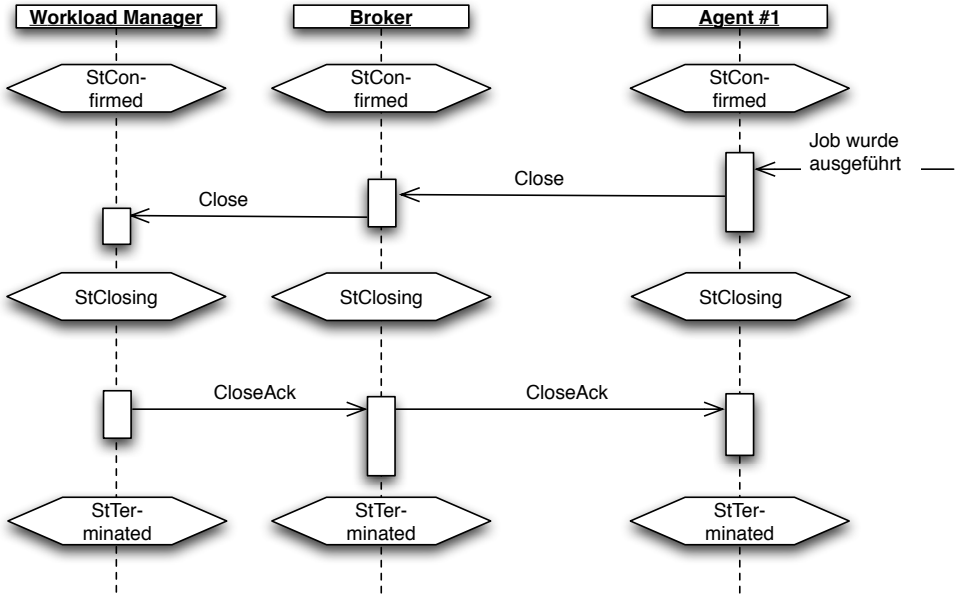


Abbildung 2.6.: Beendigung eines Jobs

Abbruch durch den Benutzer

Es muss für den Benutzer immer die Möglichkeit bestehen, einen Auftrag wieder abbrechen zu können. Im Calana-Protokoll übernimmt diese Rolle die so genannte *ConsumerCancel*-Nachricht. Kosten für den Benutzer entstehen dabei nur, wenn bereits ein Vertrag zustande gekommen ist (das Protokoll befindet sich im Zustand *StConfirmed*). Abbildung 2.7 stellt diesen Fall dar.

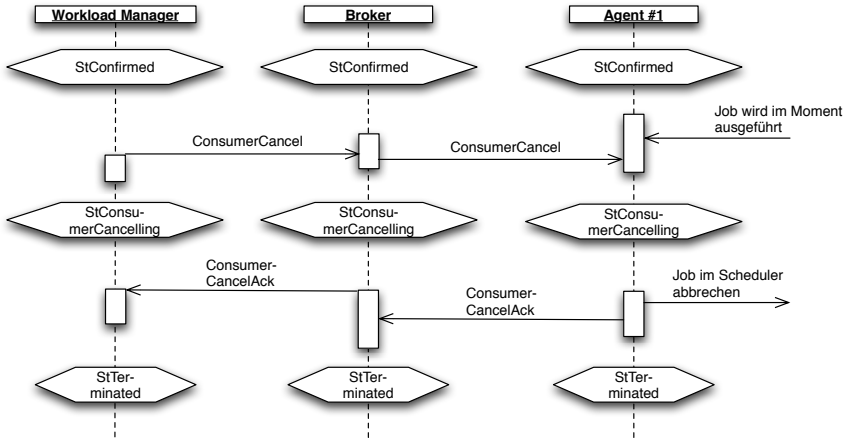


Abbildung 2.7.: Abbruch eines Jobs durch den Benutzer

Abbruch durch den Anbieter

Für einen Anbieter kommen mehrere Gründe in Frage, weshalb er einen Auftrag abbrechen muss.

- Beispielsweise kann ein Hardwarefehler auftreten, so dass der Auftrag nicht weiter bearbeitet werden kann — in diesem Fall trägt der Anbieter die Kosten, es sei denn es wurde mit dem Benutzer eine andere Übereinkunft getroffen[4]
- Ebenfalls möglich und wahrscheinlich häufiger anzutreffen ist der Fall, dass der Benutzer eine zu kurze walltime angegeben hat — den Anbieter trifft hier keine Schuld und der Benutzer muss für alle entstandenen Kosten aufkommen.
- Eine weitere Möglichkeit ist eher strategischer Natur: ein Anbieter kann einen bereits angenommenen Auftrag zu Gunsten eines wesentlich lukrativeren Auftrags abbrechen.

  Dazu erstellt er ein Gebot so, als wären die Ressourcen noch frei. Falls er die Auktion tatsächlich gewinnen sollte, bricht er den Auftrag, der die Ressourcen momentan benutzt ab. Er muss sich aber über eventuelle Strafen[5], die ihm zuvor auferlegt worden sind, im Klaren sein.

Abbildung 2.8 stellt den Fall einer zu kurzen walltime dar, so dass der lokale Scheduler gezwungen ist, den Auftrag abzubrechen.

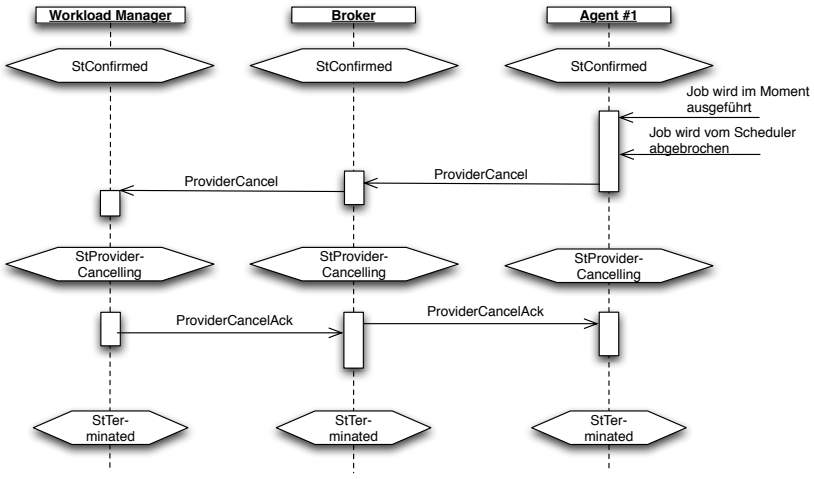


Abbildung 2.8.: Abbruch eines Auftrages durch den Anbieter

[4] Beispielsweise kann in einem Gebot die Klausel enthalten sein, dass der Anbieter für keinerlei Systemfehler die Verantwortung übernimmt.

[5] zusätzliche Kosten

2.3.3. Zusammenfassung

Wir wissen nun, wie ein Auftrag zu einem Anbieter gelangt, was passiert, wenn er von einer der beiden Seiten abgebrochen wird oder erst gar nicht ausgeführt werden kann, und wir wissen, wie eine erfolgreiche Bearbeitung aussieht. Es wird Zeit, dass wir uns mit der Implementierung und der Simulation des Protokolls auseinandersetzen.

2.4. Implementierung

Die gesamte Simulationsumgebung ist von mir in Java[6] implementiert worden. Es ist eine eventbasierte Simulation, die komplett in einer Hauptschleife (einem Kontrollfluss) abläuft. Die Simulationsumgebung stellt eine simulierte Zeit bereit, die ausschließlich mit den Events voranschreitet.

> *Warum sollte man ein solches System simulieren, wenn man es doch auch gleich an realen Systemen testen könnte?*

Wir haben es hier mit einem hochgradig verteilten System zu tun. Die Probleme, die bei einem solchen System auftreten können (unterschiedliche Zeiten auf den Systemen, Synchronisationsprobleme, Nachrichtenverluste, etc.), erschweren es, ein neues Protokoll zu testen und vorallem Fehler schnell zu finden.

Das Testen ist wesentlich einfacher, wenn man das Protokoll (oder auch jede andere beliebige Software) erst unter kontrollierten und vereinfachten Bedingungen untersuchen kann, bevor man es an realen Systemen testet:

- Die Simulation beruht auf einer synthetischen Zeit, die schrittweise voranschreitet, nämlich immer zum Zeitpunkt des nächsten Events. Jede simulierte Komponente hat also dieselbe Zeit, das ist in einem realen verteilten System nicht einfach zu lösen.
- Man kann bei einer Simulation von allen Unannehmlichkeiten, die einem beispielsweise eine Netzwerkverbindung bereitet, absehen. Darunter fallen unter anderem Verlust, Verfälschung, Verzögerung und damit eine mögliche Umordnung von Nachrichten.
- Man kann wesentlich schneller Testläufe durchführen, da vom aktuellen Zeitschritt zum nächsten beliebig viel simulierte Zeit vergehen kann.

  > Nehmen wir zum Beispiel das Verschicken eines Briefes. In Realzeit würde das ein bis zwei Tage dauern. In einer Simulierten Umgebung, kann man dem Simulator einen speziellen *BriefAngekommen*-Event übergeben und ihn in 86400 Zeiteinheiten zustellen lassen.

[6] Java 1.5, java.sun.com

Beim nächsten Durchlauf durch die Simulationsschleife wird ein ganzer Tag an Zeiteinheiten[7] übersprungen (vorausgesetzt, wir verschicken nur diesen einen Brief) und der Brief erreicht seinen Empfänger.

- Nach und nach können die gemachten Einschränkungen jedoch entfernt werden. Dazu muss nur die Simulationsumgebung dahingehend erweitert werden. Zum Beispiel kann man den Verlust oder eine zufällige Verfälschung von Nachrichten implementieren. Daraufhin kann man dann schrittweise das Protokoll erweitern, um mit den neuen Anforderungen zurecht zu kommen.

Die implementierte Simulationsumgebung besteht aus insgesamt drei Java Paketen, die aufeinander aufbauen. Der Grund für die Unterteilung ist die Modularisierung in (teilweise) unabhängige Komponenten (siehe Abbildung 2.9 für die genauen Abhängigkeiten), so dass sie auch in einem anderen Umfeld benutzt werden können.

Die Grundlage bildet das `simulator` Paket, in dem alle nötigen Klassen für eine eventbasierte Simulation enthalten sind. Das `grid` Paket stellt allgemeine Klassen zur Verfügung, die in einem Gridumfeld auftreten und für diese Arbeit wichtig sind.

Darunter fallen zum Beispiel: die Beschreibung eines Jobs, diverse Ressourcen (Node, Cluster) und lokale Scheduler, die auf diesen Ressourcen arbeiten.

Alles, was mit der Implementierung von *Calana* im Speziellen zu tun hat, ist im Paket `calana` enthalten. Dort sind beispielsweise die Agenten, die Präferenzen und die benutzten Nachrichten definiert.

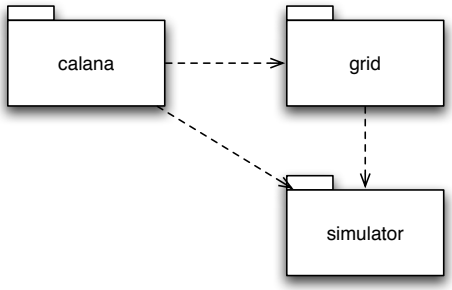


Abbildung 2.9.: Paketübersicht der Implementierung

2.4.1. Das `simulator` Paket

Da meine Hauptaufgabe darin bestand, eine eventbasierte Simulation zu erstellen, benötigte ich zu allererst einen Simulator beziehungsweise einen "Simulationskern", der in der Lage ist, Events von einem *Objekt* zu einem anderen zu "verschicken".

[7] die Zeiteinheit sei in diesem Fall *Sekunde*

Ich habe mich bei der Implementierung dafür entschieden, nur einen Thread für das gesamte System zu benutzen. Die Vorteile davon sind unter anderem:

- keine Synchronisationsprobleme (Deadlocks, Inkonsistenzen)
- keine Kontextwechsel und somit kein zusätzlicher Overhead
- leichte Verfolgbarkeit der durchgeführten Simulation
- leichte Reproduzierbarkeit der Ergebnisse einer Simulation

Nachteilig bei dem von mir gewählten Ansatz ist, dass man die Simulation nicht parallelisieren kann. Die meisten Anweisungen, die innerhalb eines Zeitschrittes durchgeführt werden, können unabhängig von einander ausgeführt werden.

Ein anderer Ansatz, der beispielsweise von *GridSim* ([18]) verfolgt wird, benutzt für jedes aktive Objekt[8] einen eigenen Thread. Dieser Ansatz ist wesentlich näher an der Realität, hat aber im Gegenzug dazu die oben genannten Punkte als Nachteile: es können Synchronisationsprobleme auftreten, es finden viele Kontextwechsel statt und auf Grund der nebenläufigen Kontrollflüsse ist die Simulation nicht mehr so einfach zu verfolgen (nach zu vollziehen).

Das `simulator` Paket besteht aus einigen wichtigen Klassen, stellt allerdings auch ein paar nützliche Hilfsklassen zur Verfügung. Die wichtigen Klassen sind:

`simulator.core.event.Event` Ein Event besitzt eine Quell- sowie eine Zieladresse (in Form von EventHandler Objekten) und der Angabe eines Zeitpunktes, zu dem dieser Event eintreten soll ("Eintrittszeitpunkt"). Man kann niemals einen Event generieren, der sofort (im aktuellen Zeitschritt) oder in der Vergangenheit eintreten soll, versucht man es trotzdem, so beendet sich die Simulation sofort mit einer Fehlermeldung.

`simulator.core.event.EventHandler` Ein Objekt, das Events empfangen möchte, muss dieses Interface implementieren. Zum Senden von Events ist es allerdings nicht zwingend erforderlich, ein `EventHandler` zu sein — der Empfänger hat dann aber keine Möglichkeit, herauszufinden, wo genau der Event herkam.

`simulator.core.SimpleSimulator` Diese Klasse implementiert das `Simulator` Interface und stellt somit den Simulator für das implementierte System dar. Der Simulator stellt Funktionen bereit, die zum Verschicken und Empfangen von Events benötigt werden, dazu verwaltet er intern eine Liste von Events, die nach ihrem Eintrittszeitpunkt sortiert sind. Die *simulierte Zeit* beginnt immer bei 0 und schreitet mit den Events fort, es werden also keine festen Zeitschritte simuliert, sondern es wird immer von Event zu Event weitergeschaltet. Alle Events, die zu ein und demselben Zeitpunkt gehören, werden nacheinander abgearbeitet

[8] im Sinne von *Sender*

(dabei bleibt die Uhr quasi stehen). Einem EventHandler steht zur Bearbeitung eines Events beliebig viel Zeit zur Verfügung, es wird also von der tatsächlichen Rechenzeit abstrahiert und alles geht in "konstanter" Zeit, nämlich einem Zeitschritt. Ein Auftrag beispielsweise besitzt eine Ankunftszeit und eine Dauer; er kann direkt einen Zeitschritt nachdem er gestartet worden ist beendet werden (sofern keine weiteren Events dazwischen liegen).

Starten der Simulation Die Simulation wird gestartet, indem man die `simulate` Methode auf der Simulator Instanz aufruft. Man betritt dadurch die "Eventschleife" (eventloop) des Simulators, die erst wieder verlassen wird, wenn die Simulation endet. Zuvor werden jedoch die `start` Methoden der EventHandler, die sie anbieten, aufgerufen. Innerhalb dieser `start` Methode können bereits Events versendet werden.

Bearbeiten eines Events Der Simulator besitzt eine Liste aller Events, die nach dem Zeitpunkt des Eintretens der Events sortiert ist. In jedem Zeitschritt entfernt der Simulator alle Events, die zu genau diesem Zeitschritt gehören und bearbeitet sie. Dazu ruft der Simulator eine erst zur Laufzeit bestimmbare Methode des Empfängers auf, der Name der Methode wird aus dem Wort `visit` und dem Klassennamen des Events zusammengesetzt. Um zum Beispiel den `Timer` Event t an den EventHandler H zuzustellen, ruft der Simulator `H.visitTimer(t)` auf. Dabei wird auch die Vererbungshierarchie beachtet; das heißt, es werden zusätzlich zum Namen "Timer" auch noch die Namen aller implementierten Interfaces und die Namen der Oberklassen probiert bis eine passende Funktion gefunden wird (bei `visitObject` endet die Suche).

Das ist ein leicht abgewandeltes *Visitor* Pattern. Es ist sehr einfach, das System um neue Events zu erweitern, ohne dass man bestehenden Code abändern muss (potentielle Fehler werden vermieden). Die meisten anderen Systeme reservieren für jeden Event eine eindeutige ID, die man in einer `process` Methode des Handlers überprüfen muss (zum Beispiel in einer großen `switch` Anweisung). Das hat den Nachteil, dass man jede dieser Anweisungen ändern muss, wenn man den Event bearbeiten möchte.

Beenden der Simulation Die Simulation endet, wenn entweder keine weiteren Events zur Bearbeitung vorliegen, eine Exception bis in die Simulationsschleife durchdringt oder ein `Stop` Event auftritt.

Der Weg eines Events beginnt damit, dass ein zugehöriges Objekt instanziiert und die `fire` Methode auf dem Simulator aufgerufen wird. Der Event besitzt eine positive Verzögerung (delay), die dazu benutzt wird, den Zeitpunkt des Eintretens zu bestimmen: Wenn τ der aktuelle Zeitschritt und δ die Verzögerung ist, dann wird der Event im Zeitschritt τ' mit $\tau' := \tau + \delta$ bearbeitet. Wenn τ' erreicht ist, wird die passende `visit` Methode des Empfängers aufgerufen und der Event aus dem System entfernt.

Zu den zusätzlichen Klassen, die dieses Paket bereitstellt, gehört zum Beispiel ein `Alarm` Objekt, welches die Funktionalität eines "Weckers" kapselt. Ein Benutzer dieses Objekts registriert sich an ihm mit einer Callback-Funktion und kann von nun an "geweckt" werden (es wird die Callback-Funktion aufgerufen). Der lokale Scheduler, den ich in Kürze beschreiben werde, benutzt ein solches Alarm Objekt, um seine Prozess-Queue zu verwalten.

2.4.2. Das `grid` Paket

Dieses Paket enthält alle Klassen, die ich in dieser Arbeit benötigt habe, um ein einfaches Grid zu repräsentieren. Es sind nur die wichtigsten Komponenten enthalten, wie zum Beispiel:

`Job` Hierin ist die Beschreibung eines Jobs enthalten. Ein Job besitzt die folgenden Eigenschaften (auszugsweise):

- eindeutige ID
- Benutzer ID, damit später unter anderem die Präferenzen zugeordnet werden können
- "submit time", die der Simulation sagt, wann der Job das System betritt
- walltime, welche die geschätzte Laufzeit des Jobs angibt
- runtime — die tatsächliche Laufzeit
- Anzahl der benötigten Nodes

`SimpleCluster` In dieser Klasse wird ein einfacher Cluster definiert, der aus n 1-CPU-Nodes besteht.

`Scheduler` Dieses Interface definiert einen lokalen Scheduler, der beispielsweise einen `SimpleCluster` verwalten kann. Eine Implementierung ist mit dem `FCFSScheduler` gegeben. Dieser implementiert die FCFS Strategie mit Backfilling (siehe dazu den Abschnitt über "Nicht-interaktives Scheduling" in Kapitel 1.2 oder in [8, 6, 5]).

Der Scheduler bietet die Möglichkeit, Reservierungen vorzunehmen. Eine Reservierung befindet sich anfänglich im Zustand *"pending"*, der besagt, dass die Reservierung noch keinen zugewiesen Auftrag enthält und jederzeit vom Benutzer abgebrochen werden kann. Einer solchen Reservierung können keine Nodes zugewiesen werden. Erst wenn sich der Benutzer dazu entschließt, der Reservierung einen Auftrag zuzuweisen und sie damit in den Zustand *"active"* versetzt, darf die Reservierung starten. Das Starten der Reservierung erfolgt mittels des Alarm Objektes, indem zum geplanten Startzeitpunkt eine Callback-Funktion aufgerufen wird.

Der Benutzer, von dem hier die Rede ist, kann jedes beliebige Objekt sein, welches das `SchedulerUser` Interface implementiert.

`Workload` Eine Workload beschreibt entweder einen Mitschnitt der ausgeführten Jobs eines Systems (Cluster, etc.) oder sie beschreibt ein generiertes Auftragsaufkommen, welches gewissen statistischen Verteilungen genügt.

Beide Arten rufen auf dem simulierten System eine Auslastung ("load") gewisser Höhe hervor. Die in dieser Arbeit verwendeten Workloads werden von einem separaten Tool generiert ([2]). Dieses Tool erzeugt eine XML Beschreibung einer Workload, die der in Kapitel B.2 beschriebenen Definition entspricht.

Aus diesen XML Dateien wird mittels des `XMLWorkloadBuilder` ein Workload Objekt erstellt. Die Datei enthält die Beschreibung der Benutzer inklusive ihrer Präferenzen und die Beschreibung der einzelnen Aufträge.

2.4.3. Das `calana` Paket

In diesem Paket findet man alle Klassen, die für die Implementierung von *Calana* notwendig waren. Das Paket stellt die verwendeten Agenten zur Verfügung (Broker, BidAgent), einen WorkloadManager, die Präferenzmodelle und die Protokollimplementierung. In Abbildung 2.10 ist das Zusammenspiel der Agenten mit ihrem lokalen Scheduler dargestellt.

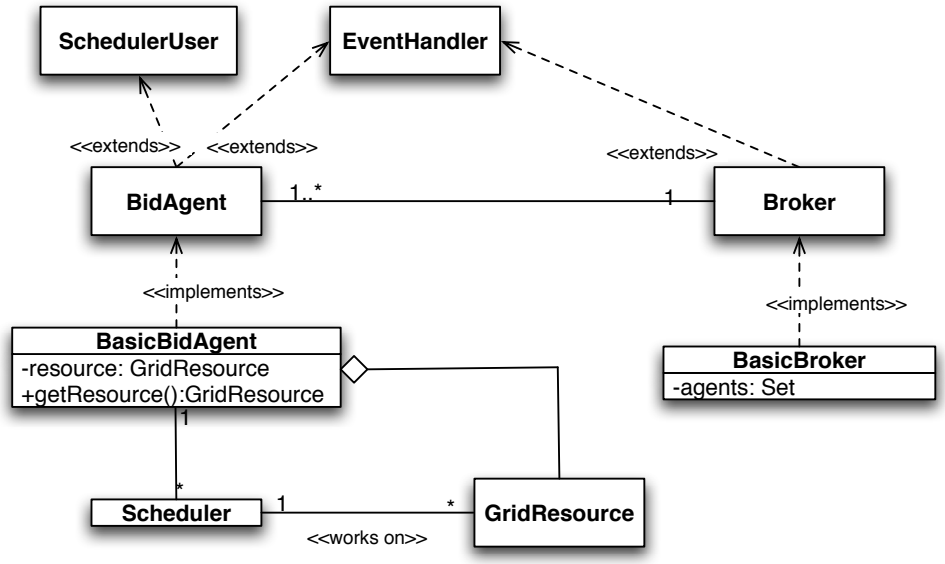


Abbildung 2.10.: Zusammenspiel von Agent, Broker und Scheduler

Alle Nachrichten, die das Protokoll verwendet, werden direkt auf Events mit dem gleichen Namen abgebildet — eine `Close` Nachricht wird durch einen `Close`-Event implementiert. Im Paket `calana.protocol` sind neben den Nachrichten auch die Zustandsautomaten definiert, die für die Implemen-

tierung des Protokolls verwendet wurden. Um die Zustandsautomaten zu beschreiben, habe ich den "State Machine Compiler" ([15]) verwendet; die zugehörigen Beschreibungen der einzelnen Automaten befinden sich im Anhang.

`BasicBidAgent` Diese Klasse implementiert einen sehr einfachen Agenten. Um auf einen `BookingReq` zu reagieren, lässt er seinen lokalen Scheduler eine vorläufige Reservierung vornehmen; diese beinhaltet unter anderem die voraussichtliche "finish time" des Auftrages.

Seine Gebote bestehen nun aus einem Preis für die Benutzung seiner Ressourcen und der "finish time". Den Preis berechnet dieser Agent aus einem konfigurierbaren Basispreis und einem Preis pro Zeiteinheit: $p(\tau) = p_b + p_t * \tau$.

Der Agent ist auch für das Ausführen der Jobs auf seiner Ressource zuständig. Dazu weist er bei Erhalt einer *Confirm* Nachricht der Reservierung den Job zu und teilt seinem Scheduler mit, dass die Reservierung nun gültig ist. Es erfolgt in meiner Implementierung keine direkte Kommunikation zwischen WorkloadManager und Ressourcen.

`DynamicBidAgent` Mit dieser Klasse demonstriere ich, wie wenig aufwendig es ist, ein anderes Verhalten des Agenten zu implementieren. Dieser Agent geht genauso vor wie der `BasicBidAgent`, nur berechnet er den Preis anhand der aktuellen Marktsituation. Dazu beginnt er ebenfalls mit einem Basis- und einem Zeitpreis. Allerdings ist der Zeitpreis nicht mehr konstant, sondern wird ständig angepasst. Dazu werden die bisher beobachteten Preise exponentiell geglättet, siehe dazu Algorithmus 1.

Algorithm 1 Calculate the smoothed market price

let P be a list of prices, that recent "winning bids" had
let s be a "smoothing factor"
$i \leftarrow |P|$ {the number of prices}
$p \leftarrow 0.0$ {the smoothed market price}
while $i > 0$ **do**
  $\tilde{p} \leftarrow P_{i-|P|+1}$
  $p \leftarrow p + (s * (1-s)^i) * \tilde{p}$
  $i \leftarrow i - 1$
end while
return p

Der Agent passt sich dem durchschnittlichen Marktpreis an und versucht somit möglichst viele Aufträge zu erhalten. Dadurch steigt die Auslastung seiner Ressourcen und seine Gewinnspanne.

`BasicBroker` Der Broker verbindet den WorkloadManager mit den einzelnen Agenten, indem er Nachrichten von der einen Seite zur anderen Seite weiterleitet.

Für die Verbindung mit dem WorkloadManager implementiert er das `c2b.ProviderProtocolInstance` Interface und für die Verbindung

mit den Agenten das `b2a.ConsumerProtocolInstance` Interface.

`WorkloadManager` Der WorkloadManager bezieht seine Jobbeschreibungen aus einer Workload. Für jeden Auftrag, der in der Workload enthalten ist, generiert er eine `BookingReq` Nachricht und sendet sie an den Broker.

Er repräsentiert innerhalb der Simulation den Benutzer, indem er *Confirm* oder *ConsumerCancel* Nachrichten verschickt.

2.5. Durchgeführte Experimente

Alle Experimente wurden wie folgt durchgeführt: Als erstes musste eine adäquate Workload generiert werden, die auf der zu testenden Systemkonfiguration eine gewisse Load verursacht. Dazu wurde der Calana Grid Workload Generator ([2]) benutzt.

Eine Systemkonfiguration besteht aus der Anzahl der Anbieter und wieviele Nodes jeder Anbieter zur Verfügung stellt. Der CGWG generiert für eine gegebene Konfiguration ein "Workload Set". Dieses Set besteht aus einzelnen Workloads, welche die Loadlevel 0.025 bis 1.0 (in $\frac{1}{40}$-tel Schritten) hervorrufen.

Anhand dieser Sets kann man beobachten, wie sich das System bei steigender Load verhält. Leider gab es kleinere Probleme mit dem Generator, so dass die erzeugte Workload nicht immer die erforderliche Load hervorgerufen hat.

2.5.1. 2 Agenten mit jeweils einer Node

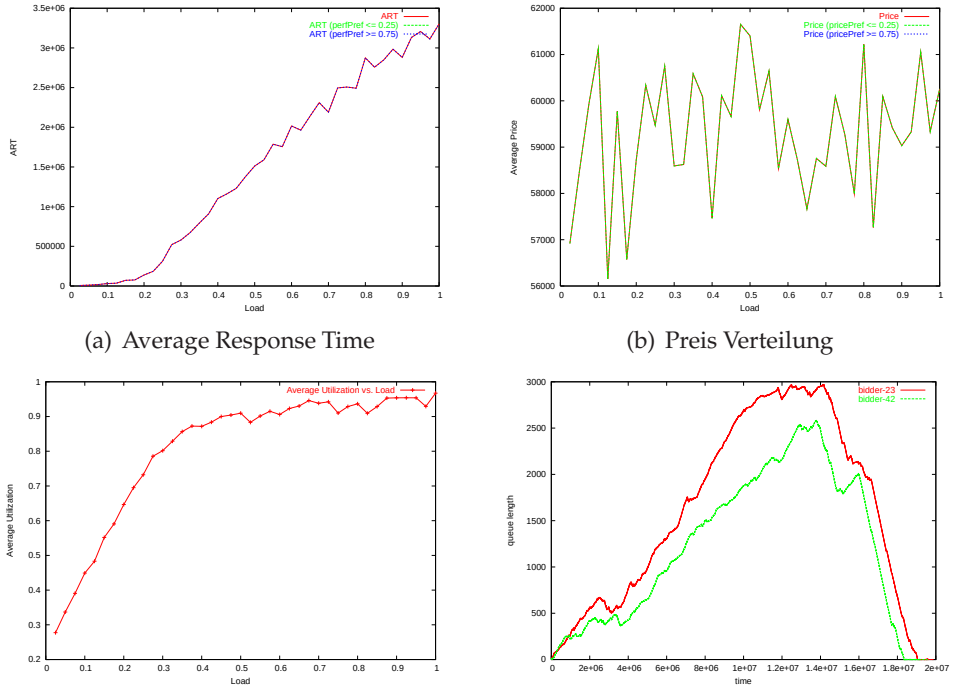
Das erste Experiment, das ich durchgeführt habe, besteht aus nur zwei Agenten, die jeweils eine 1-CPU-Node verwalten. Bearbeitet wurden circa 10000 Aufträge, die von 1000 unterschiedlichen Benutzern aufgegeben worden sind. Alle Benutzer hatten das Ziel, ihre Aufträge schnell bearbeiten zu lassen; der Preis spielte dabei keine Rolle. Die Benutzer-Präferenz bestand also einfach aus $w_t = 1.0$ und somit $w_p = 1 - w_t = 0$.

Agent	bidder-23	bidder-42
Nodes	1	1
p_b	14.0	19.67
p_t	4.09	1.25

Tabelle 2.3.: Experiment 1 - Aufbau

Wie man in Abbildung 2.11(a) erkennen kann, steigt mit zunehmender Load wie erwartet auch die ART[9]. Der durchschnittliche Preis für die Ausführung eines Jobs schwankt sehr stark und man kann keine eindeutige Tendenz erkennen (2.11(b). Das liegt daran, dass der Preis bei der Auswahl eines Gebots durch den Broker ignoriert wird (der erste Summand in der Gleichung 2.1 verschwindet).

[9] die *Average Response Time* eines Auftrages ist die Zeit, die von Betreten bis Verlassen des Systems vergeht



(a) Average Response Time

(b) Preis Verteilung

(c) Auslastung

(d) Länge der Warteschlangen

Abbildung 2.11.: Ein einfaches Experiment mit zwei Agenten

Die anderen beiden Graphen (2.11(c) und 2.11(d) zeigen zum einen die Auslastung aller Systeme und zum anderen die Länge der Warteschlange bei einer Load von 0.95. Die beiden Agenten teilen die Aufträge nahezu gleichermaßen unter sich auf und die gesamte Load der Testumgebung wächst langsam bis zu einem Sättigungspunkt.

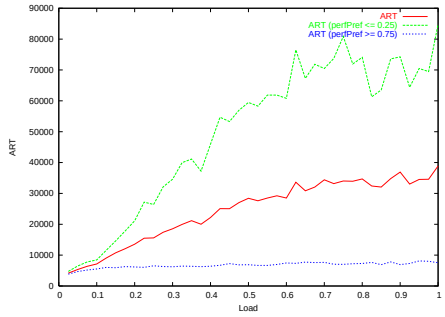
Das grundlegende System scheint demnach zu funktionieren und es liefert plausible Ergebnisse.

2.5.2. Überprüfung der Benutzer-Präferenzen

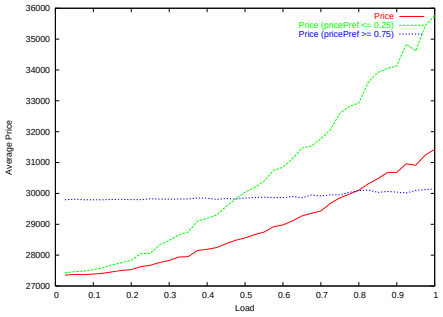
In diesem Experiment wurde überprüft, ob die Benutzer-Präferenzen eingehalten werden können. Dazu wurden als Testumgebung 50 Anbieter verwendet, die jeweils eine Node bereitstellen.

Die Kosten für die Benutzung der einzelnen Nodes wurde statistisch so verteilt, dass ein Anbieter der einen hohen Basispreis hat, einen vergleichsweise niedrigen Minutenpreis und andersherum.

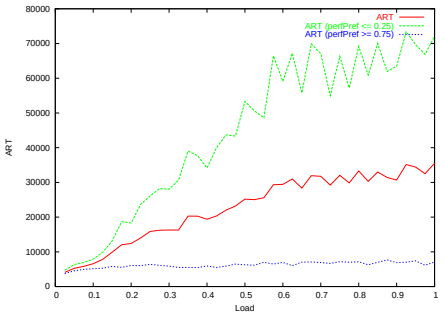
Bearbeitet wurden wieder circa 10000 Aufträge, die von 1000 Benutzern stammen. Diesmal wurden jedoch die Benutzer-Präferenzen verändert und uniform auf den Bereich $[0, 1.0]$ verteilt.
Die Average Response Time verhält sich so wie erwartet. Im zugehörigen Graphen (Abbildung 2.12(a)) kann man erkennen, dass die Aufträge der Benutzer, die eine hohe Zeit-Präferenz ($w_t > 75\%$) aufwiesen, eine kürzere Response Ti-



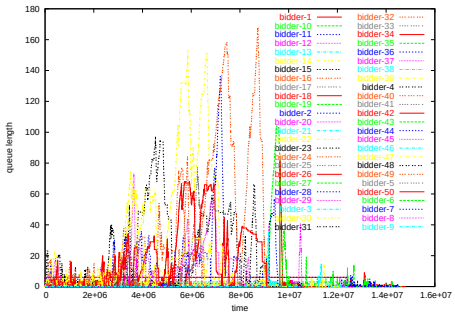
(a) Average Response Time



(b) Preisverteilung



(c) ART mit $< 100\%$ Wahrscheinlichkeit beim Bieten



(d) Länge der Warteschlangen

Abbildung 2.12.: 50 Agenten mit jeweils einer 1-CPU-Node

me hatten (die *blaue* Linie im unteren Teil des Graphen). Zusätzlich lässt sich feststellen, dass die ART für die Aufträge dieser Benutzergruppe lastunabhängig war; die ermittelten Werte beschreiben nahezu eine konstante Funktion.

Auch ein *Ändern der Wahrscheinlichkeit*, mit der ein Agent auf eine Anfrage reagiert, ändert an der Tendenz der ART nichts, sie wird nur an einigen Stellen "gestört". Diese Stellen treten immer dann auf, wenn ein "besserer" Agent kein Gebot abgibt.

Der durchschnittliche Preis ist viel zu hoch für Aufträge von Benutzern, die eine hohe Präferenz ($\geq 75\%$) auf den Preis hatten. Die Kurve dazu ist in Abbildung 2.12(b) dargestellt und beschreibt eine nahezu konstante Funktion. Erwartet hatte ich allerdings ein mit der ART vergleichbares Verhalten.

Es lässt sich jedoch erkennen, dass sich die anderen Kurven wie erwartet verhalten. Der durchschnittliche Preis steigt mit der Load, da die günstigen Anbieter nach und nach überlastet werden und somit immer häufiger teurere Anbieter gewählt werden.

Man erkennt ebenfalls, dass die Benutzer, denen die Zeit sehr wichtig war (niedrige Preis-Präferenz, $\leq 25\%$), immer mehr zahlen müssen je höher das System belastet wird.

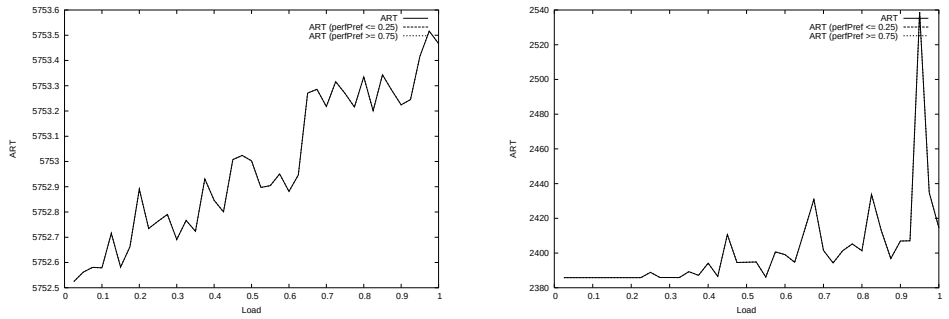
Das konstante Verhalten dieser Kurve im Vergleich zu den beiden anderen, lässt allerdings auf einen Fehler in der Auswertung der Simulationsergebnisse schließen.

2.5.3. *Scoring* der Agenten

Dieses Experiment ist durch Zufall entstanden. Bei der Analyse der Ergebnisse des ersten Experimentes ist mir aufgefallen, dass der Broker dazu tendierte einige Agenten zu bevorzugen. Da dieses Verhalten nicht mit Absicht implementiert wurde, musste es eine Erklärung dafür geben.

> Das Problem liegt daran, dass jeder Agent vorläufige Reservierungen macht, die manchmal nicht rechtzeitig entfernt werden können. Wenn mehrere Aufträge zeitnah, das heißt *während einer anderen Auktion*, das System betreten, dann besteht eine hohe Wahrscheinlichkeit, dass alle Agenten (oder einige davon) mit derselben finish time bieten: Die Scheduler haben eine vorläufige Reservierung und müssen davon ausgehen, dass sie aktiviert werden kann, demnach wird die nächste Reservierung "hinten angehängt".
>
> Da alle Agenten nur eine Node verwalten ergibt sich überall dasselbe Bild und somit betrachtet der Broker die Gebote als gleichwertig. Die Folge davon ist, dass er auf Grund der internen Datenstrukturen immer denselben Agenten auswählt.

Um das Problem zu beheben, habe ich einen "Scoring" Mechanismus im Broker implementiert, der die Agenten danach bewertet, wie oft ihnen ein Auftrag zugewiesen wurde. Sollte der Fall auftreten, dass mehrere Agenten gleichwertige Gebote abgeben, wird derjenige Agent mit dem geringsten Scoring Wert (Zählerstand) ausgewählt.



(a) Average Response Time (ohne Scoring) (b) Average Response Time (mit Scoring)

Abbildung 2.13.: Average Response Time mit und ohne Scoring

Die Ergebnisse in den Abbildungen 2.13(a) und 2.13(b) zeigen, dass sich die ART allein mit Hilfe des Scoring-Mechanismus um ungefähr die Hälfte hat verkürzen lassen. Aus diesem Grund ist der Broker standardmäßig darauf konfiguriert, Scoring zu verwenden.

Dieser Abschnitt schließt den ersten Teil der Implementierung von *Calana* ab. Es fehlen noch ausgereifte Mechanismen, um viele verschiedene Präferenzen, sowohl der Benutzer als auch der Anbieter, zu repräsentieren. Dazu zählt zum

Beispiel die Frage nach der Zuverlässigkeit der anderen Seite (für einen Benutzer kann es von großer Bedeutung sein, dass der Anbieter eine geringe Ausfallquote oder Abbruchquote hat).

Aber obwohl wir uns nur auf eine sehr kleine Auswahl an Präferenzen beschränkt haben, zeigen die Ergebnisse, dass das Verfahren funktioniert und dazu verwendet werden kann, diese unterschiedlichen Präferenzen bei verschiedenen Lastsituationen zu beachten.

Im nächsten Abschnitt befasse ich mich mit der Erweiterung des Schedulers um Koallokation.

3. Koallokation

Das Thema *Koallokation* hatte ich in der Einführung bereits kurz angerissen. In diesem Kapitel möchte ich erklären, worum genau es dabei geht und wie ich Koallokation implementiert habe.

Koallokation dient, wie der Name bereits vermuten lässt, der gleichzeitigen Reservierung (Allokation) verschiedener Ressourcen. Diese Ressourcen müssen nicht von ein und demselben Anbieter bereitgestellt werden, sondern können quer über das Grid verteilt sein:

> Der Forscher aus Abschnitt 1.3 möchte *Radio-Teleskop*, *Netzwerk* und den *Cluster* zeitgleich benutzen.
>
> Dazu wendet er sich an den Workload Manager und beauftragt ihn mit der Reservierung eben dieser Ressourcen. Es kommt dabei nicht darauf an, **wo** sich die Ressourcen befinden, sondern **ob überhaupt** eine gleichzeitige Reservierung möglich ist und wenn ja, ob diese die Präferenzen des Forschers erfüllt.
>
> Das Problem, eine solche Reservierung zu finden, bezeichnet man als *Koallokation*.

Es ist sehr wahrscheinlich, dass in diesem Fall die drei Ressourcen nicht von einem Anbieter bereitgestellt werden, sondern von drei verschiedenen. Das hat zur Folge, dass auch *drei Verträge* abgeschlossen werden müssen. Sollte einer der Vertragspartner den Vertrag nicht einhalten können, müssen alle Kosten ermittelt werden, die dadurch entstehen!

> Sollte zum Beispiel der Anbieter, der das Radio-Teleskop zur Verfügung stellt, auf Grund eines Stromausfalls die bereits bestätigte Reservierung nicht einhalten können, so muss er nicht nur für seine eigenen Kosten aufkommen, sondern muss auch alle abhängigen Kosten tragen (Nutzung des Netzwerks, Rechenzeit auf dem Cluster).

Es gibt verschiedene Möglichkeiten die Koallokation in *Calana* zu integrieren, einige davon werde ich im nächsten Abschnitt kurz vergleichen und im Anschluss die von mir gewählte vorstellen.

3.1. Implementierung

Die Koallokation beruht darauf, Reservierungen zu den Teilaufträgen so zu überlappen, dass alle Teile darin enthalten sind. In Algorithmus 2 ist ein Verfahren angegeben, welches dazu benutzt werden kann.

Eine Möglichkeit, Koallokation zu implementieren, besteht darin, die Implementierung im Grid-Scheduler — also sehr weit oben — vorzunehmen. Dazu erweitert man den Broker um das Verständnis von Koallokations-Aufträgen:

- wenn der Broker einen Koallokations-Auftrag vom Workload Manager bekommt, startet er für jeden Teil des Auftrags eine separate Auktion.
- Die Ergebnisse der Auktionen kann er anschließend dazu verwenden, mit Hilfe des Algorithmus 2 eine passende Kombination zu finden.

Dieser Ansatz hat den Vorteil, dass der Broker alle Gebote in Betracht ziehen kann, da der erst später entscheiden muss, welcher Agent gewinnen soll.

Ein weiterer Ansatz könnte zum Beispiel Broker und Agenten um Koallokation erweitern. Dazu würde jeder Agent mit mehreren Geboten auf einen Koallokations-Auftrag antworten. Der Broker bestimmt anhand dieser Gebote wiederum eine passende Allokation.

Beide Ansätze haben den Nachteil, dass eine oder mehrere zentrale Komponenten verändert werden müssen. In der Realität sollte man eine solche nachträgliche Veränderung nur mit Bedacht vornehmen, da sehr schnell ein unbrauchbares System entstehen kann.

Der von mir gewählte Ansatz fügt einen weiteren, speziellen Agenten in das System ein, den *Koallokations-Agent*[1]. Diese Erweiterung des Systems kann man als eine Art *Decorator* Pattern betrachten, da es möglich war, Koallokation ohne Veränderung der bereits bestehenden Infrastruktur zu implementieren.

3.1.1. Der *Koallokations-Agent*

Der Koallokations-Agent reagiert auf einen Koallokations-Auftrag, indem er selbst als Benutzer auftritt. Er spielt also genauso wie der Broker eine Doppelrolle: zum einen als Agent, der Aufträge annehmen und bearbeiten kann und zum anderen als ein *virtueller Benutzer*. Der prinzipielle Ablauf sieht wie folgt aus:

- Bei Erhalt eines Auftrags überprüft der Koallokations-Agent, ob es sich um einen Koallokations-Auftrag handelt. Wenn dies der Fall sein sollte, erstellt er für jeden Teilauftrag eine *BookingReq* Nachricht und sendet sie direkt an den Broker.

  Die Anfragen bestehen dabei aus *primitiven* Aufträgen, die von den anderen Agenten gehandhabt werden können.

  Erhält ein normaler Agent einen Koallokations-Auftrag, verwirft er die Anfrage, da er nichts damit anfangen kann.
- Der Broker eröffnet für jeden Teilauftrag nun eine Auktion und lässt die Agenten bieten. Da der Koallokations-Agent keine reale Ressource verwaltet, verwirft er alle Aufträge, die keine Koallokation benötigen.

[1] implementiert in der *CoallocationAgent* Klasse

- Nachdem die Auktion beendet ist, gibt es für jeden Teilauftrag ein Gebot, welches gewonnen hat. Diese werden dem Koallokations-Agent zur Begutachtung vorgelegt (der Broker warten auf die *Confirm* Nachrichten).
- Der Koallokations-Agent betrachtet nun alle Gebote und die darin enthaltenen Reservierungen und versucht mit Algorithmus 2 eine Überlappung zu finden.

  Kann eine solche gefunden werden, bietet der Koallokations-Agent auf den Koallokations-Auftrag. Der Preis bestimmt sich hierbei aus der Summe der einzelnen Beträge plus einer Gebühr.

  Gewinnt er diese Auktion, so bestätigt er alle vorgenommenen Teil-Reservierungen; die Koallokation konnte erfolgreich durchgeführt werden.

  Wenn keine Reservierung gefunden werden kann oder die Auktion verloren wird, sendet er für jeden Teilauftrag eine *ConsumerCancel* Nachricht an den Broker. Dabei entstehen keine Kosten, da noch kein Vertrag zustande gekommen ist.

Algorithm 2 Find coallocation slots

let T be the coallocation task (with $|T|$ subtasks)
let m be the maximum duration of a subtask in T
let B be the set of all bids to the subtasks of T
sort B according to the starttime
$A \leftarrow \emptyset$ {availability set}
$C \leftarrow \emptyset$ {candidate set}
for all $b \in B$ **do**
    $s \leftarrow starttime(b)$
    $e \leftarrow endtime(b)$
    for all $\tilde{b} \in A$ **do**
        if $endtime(\tilde{b}) \leq s$ **then**
            remove $\tilde{b}$ from A {$\tilde{b}$ cannot be used anymore, since it ends too early}
        else if $endtime(\tilde{b}) < e$ **then**
            $e \leftarrow endtime(\tilde{b})$
        end if
    end for
    $A \leftarrow A \cup b$
    if $|A| \geq |T|$ and $e - s \geq m$ **then** {enough slots in availability set and length of window is largeenough}
        $A \leftarrow \emptyset$
        $C \leftarrow C \cup (s, e)$ {a new slot from $s \rightarrow e$}
    end if
end for

Ein Nachteil bei diesem Ansatz ist, dass dem Koallokations-Agent für jeden Teilauftrag nur ein möglicher Slot zur Verfügung steht. Aus diesem Grund versucht er sehr viel längere Reservierungen vorzunehmen, als tatsächlich benötigt werden. Wenn die Reservierungen bestätigt werden, werden ihre tatsächlichen Längen bekanntgegeben.

Ein schöner Nebeneffekt dieser Implementierung ist die Möglichkeit, verschiedene Koallokations-Agenten im System zu haben, die unterschiedliches Verhalten aufweisen können. Diese Agenten beeinträchtigen das reguläre Verhalten des Systems nicht und werden nur aktiv, wenn ihre Dienste auch tatsächlich benötigt werden.

Man muss sich nun noch Gedanken darüber machen, was passiert, wenn einer der Komponenten einen Abbruch signalisiert.

3.1.2. Kaskadierte Verträge

Durch Verwendung des Koallokations-Agenten bestehen keine direkten Verträge zwischen Benutzer und Anbieter. Dafür gibt es einen Vertrag zwischen Benutzer und Koallokations-Agent und zwischen Koallokations-Agent und den einzelnen Anbietern.

Sollte nun eine der Parteien sich dazu entschließen, von ihrem Vertrag zurücktreten, tritt der Koallokations-Agent wieder als Mittelsmann auf.

Tritt ein Anbieter zurück, werden ihm die Kosten für die Benutzung der anderen Ressourcen auferlegt, zusätzlich werden den anderen Anbietern *ConsumerCancel* Nachrichten geschickt.

Dem Benutzer wird daraufhin eine *ProviderCancel* Nachricht geschickt und der Koallokations-Auftrag ist damit ordnungsgemäß abgebrochen. Der Benutzer fordert seine Entschädigung vom Koallokations-Agent, welcher seinerseits die Forderungen an den schuldigen Anbieter abtritt.

Bei einem Abbruch durch den Benutzer wird die *ConsumerCancel* Nachricht direkt an die betroffenen Anbieter weitergeleitet.

Es folgen die Ergebnisse einiger durchgeführter Simulationen, um zu zeigen, dass die Koallokation von Aufträgen durchgeführt werden kann.

3.2. Durchgeführte Experimente

Das prinzipielle Vorgehen ist dasselbe wie im Abschnitt 2.5. Allerdings beinhalten die Workloads diesmal einen gewissen Anteil Koallokations-Aufträge.

In den folgenden Experimenten wird eine kleine Grid-Umgebung simuliert, die aus 28 Agenten und einem Koallokations-Agenten besteht. Die Ressourcen verteilen sich wie folgt auf die Agenten:

20 Agenten bedienen nur eine Node

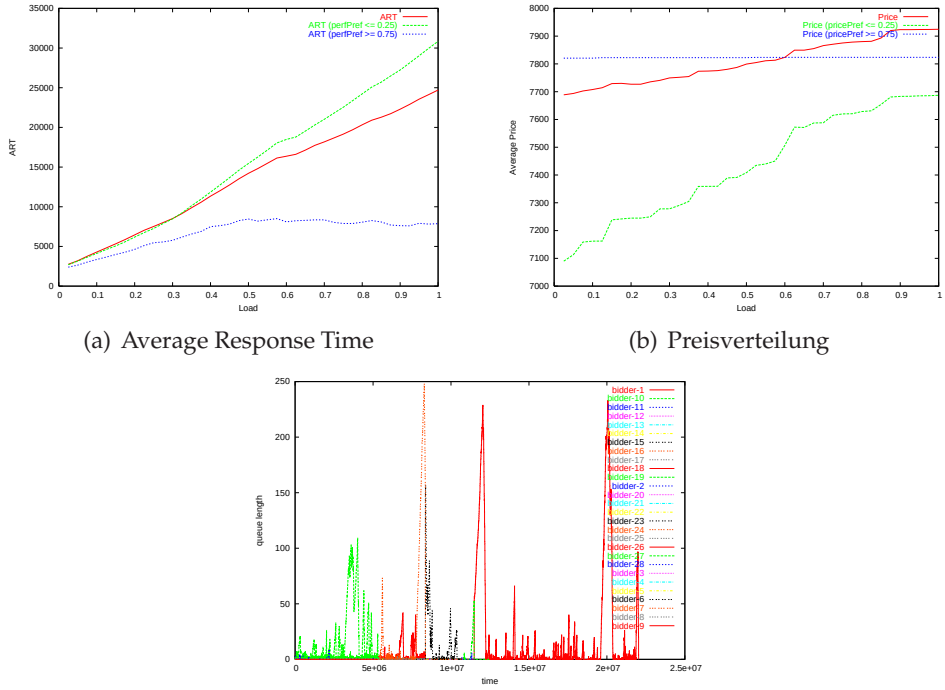
5 Agenten einen 16-Node Cluster und

3 Agenten einen 32-Node Cluster

3.2.1. Ein kleines Grid (0% Koallokation)

Dieses Experiment dient allein dem Zweck zu analysieren was passiert, wenn Koallokationen vorkommen. Dazu wird zunächst das Verhalten des Systems untersucht, wenn **keine** Koallokationen auftreten.

Die Preise der Ressourcen und die Benutzer-Präferenzen sind wie gehabt uniform verteilt.



(a) Average Response Time

(b) Preisverteilung

(c) Länge der Warteschlangen

Abbildung 3.1.: Ergebnisse eines Vergleich Experiments

Man kann in Abbildung 3.1(a) erkennen, dass sich die ART wie in den vorhergehenden Experimenten verhält. Für die Benutzer mit einer hohen Zeit-Präferenz gibt es bis zu einer Systemauslastung von circa 40%–45% einen leichten Anstieg der ART. Ab diesem Punkt bleibt die ART aber nahezu konstant.

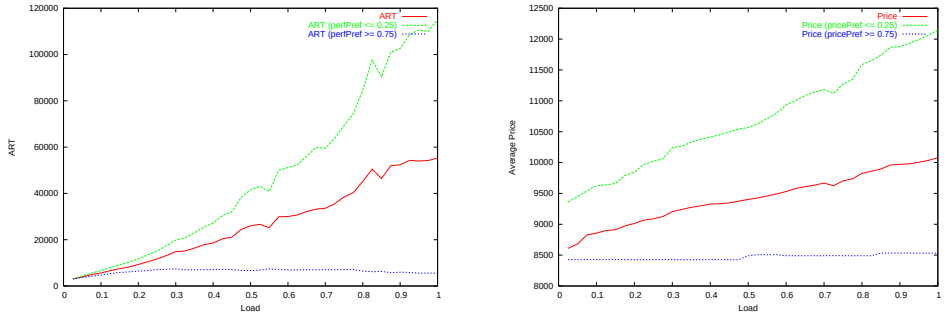
Eine mögliche Erklärung dafür ist, dass bei einer höheren Load häufiger Backfilling dafür sorgt, dass ein Auftrag schneller ausgeführt werden kann.

In den folgenden Experimenten wird der Anteil der Koallokation-Aufträge langsam erhöht.

3.2.2. Ein kleines Grid (10% Koallokation)

In diesem Experiment betrachten wir denselben Versuchsaufbau wie im Experiment zuvor, allerdings wurde die Anzahl an Koallokations-Aufträgen erhöht (10 % aller Aufträge beinhalten eine Koallokation).

Wenn man die ART aus Abbildung 3.2(a) mit der aus Abbildung 3.1(a) vergleicht, erkennt man, dass sich die unteren Kurven kaum unterscheiden. Im



(a) Average Response Time (b) Preisverteilung

Abbildung 3.2.: Ergebnisse des Simulationslaufes mit Workloads, die jeweils 10% Koallokations-Aufträge enthielten

ersten Graphen genauso wie im zweiten Graphen geht ihr Maximum bis circa 7500 Zeiteinheiten und die Kurve flacht im Laufe der Zeit ab. In 3.2(a) lässt sich sogar ein leichter Rückgang der ART erkennen, je höher die Auslastung des Systems. Man kann wieder schlussfolgern, dass die ART für die Benutzer mit einer hohen Präferenz auf die Zeit gering, ja sogar lastunabhängig ist.

Der durchschnittliche Preis für die einzelnen Präferenzen verhält sich so wie erwartet.

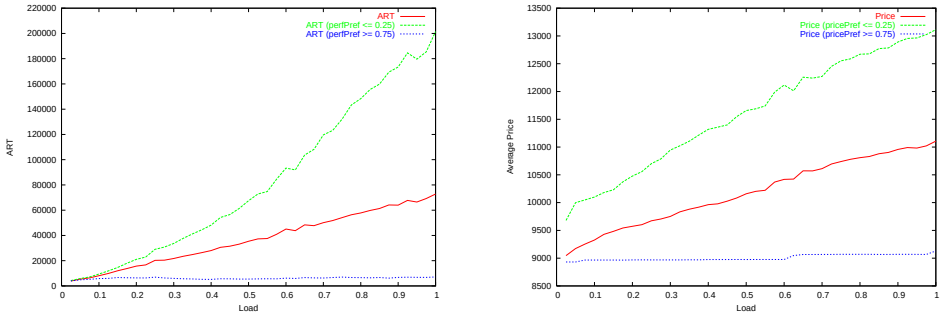
Die untere Kurve (Preis-Präferenz $\geq 75\%$) bleibt über die verschiedenen Loadlevels ungefähr konstant (vergleichbar mit der ART), wohingegen der Preis für schnelle Aufträge (Preis-Präferenz $\leq 25\%$) sowie der Durchschnittspreis über alle Aufträge stets ansteigt.

3.2.3. Ein kleines Grid (20% Koallokation)

In diesem Experiment wurde der Anteil an Koallokations-Aufträgen ein weiteres Mal erhöht, diesmal auf 20%. Die Tendenz der einzelnen Graphen in Abbildung 3.2.3 ist dieselbe wie in den Abbildungen 3.2.1 und 3.2.2.

Die maximale ART hat sich allerdings um circa 100.000 Zeiteinheiten vergrößert und der Preis stieg durchschnittlich um ungefähr 700 – 1.000 Geldeinheiten.

Es lässt sich insgesamt aus den letzten beiden Abbildungen (3.2(a) und 3.3(a)) ablesen, dass die ART für Aufträge von Benutzern, die eine hohe Präferenz auf die Bearbeitungszeit angegeben haben, konstant niedrig ist im Vergleich zum Gesamtdurchschnitt. Dies gilt sowohl unabhängig von der Load als auch für den Anteil an Koallokations-Aufträgen. Für den Preis lässt sich prinzipiell die gleiche Aussage treffen.



(a) Average Response Time (b) Preisverteilung

Abbildung 3.3.: Ergebnisse des Simulationslaufes mit Workloads, die jeweils 20% Koallokations-Aufträge enthielten

4. Zusammenfassung

Mit dieser Arbeit ist eine grundlegende, eventbasierte Simulationsumgebung entstanden, mit deren Hilfe Grid-Umgebungen simuliert werden können. In diesem Rahmen ist auch eine Implementierung von *Calana*, einem agentenbasierten Grid-Scheduler, entstanden.

Calana benutzt Auktionen, um aus den diversen Anbietern den besten auszuwählen. Die Entscheidung kann vom Benutzer beeinflusst werden, indem er so genannte Präferenzen definiert. Diese Präferenzen sollen die Wünsche des Benutzers widerspiegeln. In der aktuellen Implementierung werden als Präferenzen nur der Preis für die Benutzung einer Ressource und die Bearbeitungszeit des Auftrags betrachtet.

Nachdem die Protokoll-Implementierung und die Implementierung aller erforderlichen Grundlagen (Simulator, einfacher Scheduler für parallele Systeme, etc.) abgeschlossen war, konnte das System um Koallokation erweitert werden. Dazu wurde kein zentraler Bestandteil des Systems verändert (Protokoll, etc.); es musste nur ein neuer *Agent* eingebunden werden.

Mit den durchgeführten Experimenten hat sich gezeigt, dass der implementierte Scheduler die angegebenen Benutzer-Präferenzen unter verschiedenen Lastsituationen einhalten kann, dazu wurden spezielle Workloads generiert, die eine bestimmte Last auf dem Gesamtsystem hervorrufen.

Um eine vollständige Simulation eines realen Grids durchführen zu können, benötigt man allerdings noch ein paar Grundbausteine, wie zum Beispiel die Modellierung eines *Netzwerkes*, so dass Verbindungswege zwischen den Anbietern mit in das Scheduling einbezogen werden können. Desweiteren fehlen Mechanismen, um einen *Workflow* zu beschreiben. Ein solcher Workflow könnte beispielsweise aus den folgenden elementaren Schritten bestehen: Datentransfer, Reservierung von Rechenzeit, Datentransfer, Reservierung einer Ausgabeeinheit (Visualisierungs-Pipeline).

Da der implementierte Scheduler agentenbasiert ist, kann man noch sehr viele Tests mit unterschiedlichem Agentenverhalten durchführen. Ich habe nur sehr grundlegende Agenten implementiert, die kein außergewöhnliches Verhalten besitzen. An dieser Stelle lässt sich noch sehr viel experimentieren. Besonders interessant sind sicherlich Agenten die nach wirtschaftlichen Kriterien ihre eigenen Parameter anpassen (Preis) oder absichtlich Aufträge abbrechen, um einen besseren Auftrag annehmen zu können. Ebenfalls denkbar sind “Geheimabkommen” unter einigen Agenten, um anderen Agenten gegenüber eventuell einen Vorteil zu haben.

A. Das Calana Protokoll

Es folgen die Beschreibungen für die Zustandsübergänge der einzelnen Komponenten des Protokolls.

A.1. Client $\longleftrightarrow$ Broker

A.1.1. Consumer

```
%{
/**
 * Copyright (c) Alexander Petry 2006 <petry@itwm.fhg.de>
 *
 */
%}

%class C2BConsumer
%package calana.protocol.c2b
%import calana.protocol.messages.*
%import calana.core.Bid
%access public
%start Consumer::StInitial

%map Consumer
%%
StInitial
Entry { register(); }
{
        // in this state we are only able to make booking requests
        BookingReq(msg: BookingReq)
                StBooking
                {
                        bookingReq(msg);
                }
}

StBooking {
        Booked(b: Bid)
                StBooked
                {
                        booked(b);
                }

        BookingRejected(reason: BookingRejected.Reason)
                StRejecting
```

```
37                  {
38                          bookingRejected(reason);
39                  }
40
41          ConsumerCancel
42              StConsumerCancelling
43              {
44                  consumerCancel();
45              }
46  }
47
48  StBooked {
49      ProviderCancel
50          StProviderCancelling
51          {
52              providerCancel();
53          }
54
55      ConsumerCancel
56          StConsumerCancelling
57          {
58              consumerCancel();
59          }
60
61          Confirm
62                  StConfirmed
63                  {
64                          confirm();
65                  }
66  }
67
68  StConfirmed {
69          Close
70                  StClosing
71                  {
72                          close();
73                  }
74
75      ProviderCancel
76          StProviderCancelling
77          {
78              providerCancel();
79          }
80
81      ConsumerCancel
82          StConsumerCancelling
83          {
84              consumerCancel();
85          }
86  }
87
88  StClosing {
89          Terminated
```

```
90                  StTerminated
91                  {
92                          closeAck();
93                  }
94  }
95
96  StConsumerCancelling {
97          ConsumerCancelAck
98                  StTerminated
99                  {
100                         consumerCancelAck();
101                 }
102
103         Booked(b: Bid)
104                 nil
105                 {}
106
107         ProviderCancel
108                 StProviderCancelling
109                 {
110                         providerCancel();
111                 }
112
113         Close
114                 StClosing
115                 {
116                         close();
117                 }
118
119         BookingRejected(reason: BookingRejected.Reason)
120                 StRejecting
121                 {
122                         bookingRejected(reason);
123                 }
124 }
125
126 StProviderCancelling {
127         Terminated
128                 StTerminated
129                 {
130                         providerCancelAck();
131                 }
132 }
133
134 StRejecting {
135         Terminated
136                 StTerminated
137                 {
138                         bookingRejectedAck();
139                 }
140 }
141
142 StTerminated
```

```
143 Entry { unregister(); }
144 {
145 }
146
147 %%
```

A.1.2. Provider

```
1  %{
2  /**
3   * Copyright (c) Alexander Petry 2006 <petry@itwm.fhg.de>
4   *
5   */
6  %}
7
8  %class C2BProvider
9  %package calana.protocol.c2b
10 %import calana.protocol.messages.*
11 %import calana.core.Bid
12 %access public
13 %start Provider::StInitial
14
15 %map Provider
16 %%
17 StInitial
18 Entry { register(); }
19 {
20         // in this state we are only able to handle booking requests
21         BookingReq(msg: BookingReq)
22                 StBooking
23                 {
24                         bookingReq(msg);
25                 }
26 }
27
28 StBooking {
29         Booked(b: Bid)
30                 StBooked
31                 {
32                         booked(b);
33                 }
34
35         BookingRejected(reason: BookingRejected.Reason)
36                 StRejecting
37                 {
38                         bookingRejected(reason);
39                 }
40
41     ConsumerCancel
42         StConsumerCancelling
43         {
```

```
44              consumerCancel();
45          }
46  }
47
48  StBooked {
49      ProviderCancel
50          StProviderCancelling
51          {
52              providerCancel();
53          }
54
55      ConsumerCancel
56          StConsumerCancelling
57          {
58              consumerCancel();
59          }
60
61          Confirm(msg: Confirm)
62                  StConfirmed
63                  {
64                          confirm(msg);
65                  }
66  }
67
68  StConfirmed {
69          Close
70                  StClosing
71                  {
72                          close();
73                  }
74
75      ProviderCancel
76          StProviderCancelling
77          {
78              providerCancel();
79          }
80
81      ConsumerCancel
82          StConsumerCancelling
83          {
84              consumerCancel();
85          }
86  }
87
88  StClosing {
89          CloseAck
90                  StTerminated
91                  {
92                          closeAck();
93                  }
94
95      ConsumerCancel
96          nil
```

```
97          {
98              consumerCancel();
99          }
100 }
101
102 StConsumerCancelling {
103         ConsumerCancelAck
104                 StTerminated
105                 {
106                         consumerCancelAck();
107                 }
108 }
109
110 StProviderCancelling {
111     ProviderCancelAck
112         StTerminated
113         {
114             providerCancelAck();
115         }
116
117     Confirm
118         nil
119         {}
120
121         ConsumerCancel
122                 nil
123                 {}
124 }
125
126 StRejecting {
127         BookingRejectedAck
128             StTerminated
129             {
130                 bookingRejectedAck();
131             }
132
133         ConsumerCancel
134         nil
135                 {}
136 }
137
138 StTerminated
139 Entry { unregister(); }
140 {
141
142 }
143
144 %%
```

A.2. Broker ⟷ Agent

A.2.1. Consumer

```
%{
/**
 * Copyright (c) Alexander Petry 2006 <petry@itwm.fhg.de>
 *
 */
%}

%class B2AConsumer
%package calana.protocol.b2a
%import calana.core.Bid
%import calana.protocol.messages.*
%access public
%start Consumer::StInitial

%map Consumer
%%
StInitial
Entry { register(); }
{
        // in this state we are only able to make booking requests
        BookingReq(msg: BookingReq)
                StAuctionRunning
                {
                        bookingReq(msg);
                }
}

StAuctionRunning {
    AuctionBid(b: Bid)
        nil
        {
            auctionBid(b);
        }

    AuctionEnd
        // the auction has failed, if no bids have arrived by now
        [ctxt.getAuction().getBids().isEmpty()]
        StAuctionFailed
        {
            auctionFailed();
        }

    AuctionEnd
        // the auction has completed, if at least one bid arrived
        [!ctxt.getAuction().getBids().isEmpty()]
        StAuctionEnd
        {
            auctionEnd();
```

```
49          }
50  }
51
52  StAuctionFailed {
53      // ignore any new bid
54      AuctionBid(b: Bid)
55          nil
56          {}
57
58      BookingRejected(reason: BookingRejected.Reason)
59          StRejecting
60          {
61              bookingRejected(reason);
62          }
63
64      ConsumerCancel
65          StConsumerCancelling
66          {
67              consumerCancel();
68          }
69  }
70
71  StAuctionEnd {
72      // ignore any new bid
73      AuctionBid(b: Bid)
74          nil
75          {}
76
77      Booked(b: Bid)
78          StBooked
79          {
80              booked(b);
81          }
82
83      BookingRejected(reason: BookingRejected.Reason)
84          StRejecting
85          {
86              bookingRejected(reason);
87          }
88
89      ConsumerCancel
90          StConsumerCancelling
91          {
92              consumerCancel();
93          }
94  }
95
96  StBooked {
97      ProviderCancel
98          StProviderCancelling
99          {
100             providerCancel();
101         }
```

```
102
103     Confirm(msg: Confirm)
104         StConfirmed
105         {
106             confirm(msg);
107         }
108
109      ConsumerCancel
110         StConsumerCancelling
111         {
112             consumerCancel();
113         }
114 }
115
116 StConfirmed {
117      Close
118         StClosing
119         {
120             close();
121         }
122
123      ProviderCancel
124         StProviderCancelling
125         {
126             providerCancel();
127         }
128
129      ConsumerCancel
130         StConsumerCancelling
131         {
132             consumerCancel();
133         }
134 }
135
136 StClosing {
137      Terminated
138         StTerminated
139         {
140             closeAck();
141         }
142 }
143
144 StConsumerCancelling {
145      ConsumerCancelAck
146         StTerminated
147         {
148             consumerCancelAck();
149         }
150
151       Booked(b: Bid)
152         nil
153         {}
154
```

```
155         AuctionBid(b: Bid)
156           nil
157           {}
158
159         ProviderCancel
160           StProviderCancelling
161           {
162               providerCancel();
163           }
164
165         Close
166           StClosing
167           {
168               close();
169           }
170
171         BookingRejected(reason: BookingRejected.Reason)
172           StRejecting
173           {
174               bookingRejected(reason);
175           }
176 }
177
178 StProviderCancelling {
179         Terminated
180           StTerminated
181           {
182               providerCancelAck();
183           }
184 }
185
186 StRejecting {
187     Terminated
188           StTerminated
189           {
190               bookingRejectedAck();
191           }
192 }
193
194 StTerminated
195 Entry { unregister(); }
196 {
197 }
198
199 %%
```

A.2.2. Provider

```
1  %{
2  /**
3   * Copyright (c) Alexander Petry 2006 <petry@itwm.fhg.de>
```

```
 *
 */
%}

%class B2AProvider
%package calana.protocol.b2a
%import calana.core.AuctionInfo
%import calana.protocol.messages.*
%access public
%start Provider::StInitial

%map Provider
%%
StInitial
Entry { register(); }
{
        BookingReq(msg: BookingReq)
                StAuctionRunning
                {
                        bookingReq(msg);
                }
}

StAuctionRunning {
    AuctionBid(msg: BookingReq)
             [ctxt.wantsToBid(msg)]
        StBooking
        {
            auctionBid(msg);
        }

    AuctionBid(msg: BookingReq)
        StTerminated
        {
            auctionNotBid(msg);
        }

    AuctionCancel()
        StTerminated
        {
            auctionCancel();
        }
}

StBooking {
    // there may be different bids, that we have given, so we
    // need the actual winning bid at this point
    AuctionAccept(a: AuctionInfo)
        StAuctionWon
        {
            auctionWon(a);
        }

```

```
57      AuctionDeny(a: AuctionInfo)
58          StTerminated
59          {
60              auctionDeny(a);
61          }
62
63      ConsumerCancel
64          StConsumerCancelling
65          {
66              consumerCancel();
67          }
68  }
69
70  StAuctionWon {
71      // due to system failures and so on, we may have
72      // to cancel the reservation
73      BookingRejected(reason: BookingRejected.Reason)
74          StRejecting
75          {
76              bookingRejected(reason);
77          }
78
79      Booked
80          StBooked
81          {
82              booked();
83          }
84
85      ConsumerCancel
86          StConsumerCancelling
87          {
88              consumerCancel();
89          }
90  }
91
92  StBooked {
93      ProviderCancel
94          StProviderCancelling
95          {
96              providerCancel();
97          }
98
99      Confirm(msg: Confirm)
100         StConfirmed
101         {
102             confirm(msg);
103         }
104
105      ConsumerCancel
106         StConsumerCancelling
107         {
108             consumerCancel();
109         }
```

```
110 }
111
112 StConfirmed {
113      Close
114         StClosing
115         {
116             close();
117         }
118
119      ProviderCancel
120         StProviderCancelling
121         {
122             providerCancel();
123         }
124
125      ConsumerCancel
126         StConsumerCancelling
127         {
128             consumerCancel();
129         }
130 }
131
132 StClosing {
133     CloseAck
134         StTerminated
135         {}
136
137     ConsumerCancel
138         nil
139         {}
140 }
141
142 StConsumerCancelling {
143      ConsumerCancelAck
144         StTerminated
145         {
146             consumerCancelAck();
147         }
148 }
149
150 StRejecting {
151     BookingRejectedAck
152         StTerminated
153         {
154             bookingRejectedAck();
155         }
156
157     ConsumerCancel
158         nil
159         {}
160 }
161
162 StProviderCancelling {
```

```
163     ProviderCancelAck
164         StTerminated
165         {
166             providerCancelAck();
167         }
168
169     Confirm
170         nil
171         {}
172
173     ConsumerCancel
174         nil
175         {}
176 }
177
178 StTerminated
179 Entry { unregister(); }
180 {
181 }
182
183 %%
```

B. XML Beschreibungen

Das gesamte Programmpaket lässt sich über verschiedene XML-Dateien konfigurieren.

B.1. Experiment

Es folgt die DTD (Document Type Definition) für ein Experiment. Ein Experiment beschreibt den Aufbau des Grids. Dabei werden die einzelnen Agenten mit ihren Präferenzen definiert und ihnen Ressourcen zugewiesen (in Form eines Clusters). Man kann für jede Komponente (WorkloadManager, Broker, Bidder) die verwendete Java®Klasse angeben, so dass man sehr flexibel verschiedene Agenten ausprobieren kann, ohne den Code ändern zu müssen.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2
3  <!-- Authors: Alexander Petry. -->
4  <!-- Version: 0.1 -->
5
6  <!--
7  This document describes, how an experiment xml
8  file should be specified
9  -->
10
11 <!ELEMENT calana:experiment (description?,workload,cluster+,agents)>
12 <!ATTLIST calana:experiment
13   name                CDATA #IMPLIED
14   xmlns:calana        CDATA #FIXED "http://www.itwm.fhg.de/calana/experiment/"
15 >
16
17 <!ELEMENT description (#PCDATA)>
18
19 <!ELEMENT workload (input?,output?)>
20
21
22 <!ELEMENT input (file)>
23 <!ATTLIST input
24   format         (swf|xml) "xml"
25 >
26 <!ELEMENT output (file)>
27 <!ATTLIST output
28   format         (xml|txt) "xml"
29 >
30 <!ELEMENT file (#PCDATA)>
31
```

```
<!ELEMENT cluster (cpus,scheduler)>
<!ATTLIST cluster
  name                ID        #REQUIRED
  class                CDATA        #IMPLIED
>
<!ELEMENT cluster-ref EMPTY>
<!ATTLIST cluster-ref
  name                IDREF        #REQUIRED
>

<!ELEMENT cpus (#PCDATA)>
<!ATTLIST cpus
  mips                CDATA        "1"
>
<!ELEMENT scheduler EMPTY>
<!ATTLIST scheduler
  type                (easy|fcfs) "easy"
  class                CDATA        #IMPLIED
>

<!ELEMENT agents (manager,broker+,agent*)>
<!ATTLIST agents
  topbroker        IDREF        #REQUIRED
>

<!ELEMENT manager (prop*)>
<!-- the default class is: calana.agents.impl.BasicWorkloadManager -->
<!ATTLIST manager
  name                ID        #REQUIRED
  class                CDATA        #IMPLIED
>

<!ELEMENT broker (prop*, cluster-ref?)>
<!ATTLIST broker
  name                ID        #REQUIRED
  class                CDATA        #IMPLIED
>
<!ELEMENT broker-ref EMPTY>
<!ATTLIST broker-ref
  name                IDREF        #REQUIRED
>

<!ELEMENT agent (cluster-ref?, broker-ref?, prop*, ruleconfig?)>
<!ATTLIST agent
  name                ID        #REQUIRED
  class                CDATA        #IMPLIED
>

<!ELEMENT ruleconfig (prop*)>

<!ELEMENT prop EMPTY>
<!ATTLIST prop
  name                CDATA        #REQUIRED
```

```
  value                CDATA        #REQUIRED
>
```

B.1.1. Beispiel Experiment

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE calana:experiment SYSTEM "experiment.dtd">
<calana:experiment name="foo-bar"
    xmlns:calana="http://www.itwm.fhg.de/calana/experiment/">
    <description>
        some experiment
    </description>
    <workload>
        <input format="xml">
           <!-- we use the file from the commandline -->
           <file>some-non-existent-file.xml</file>
        </input>
        <output format="txt">
           <file>-</file>
        </output>
    </workload>
    <cluster name="cluster-1">
        <cpus>1</cpus>
        <scheduler type="fcfs"/>
    </cluster>
    <agents topbroker="broker">
        <manager name="workload-manager"/>
        <broker name="broker">
           <prop name="auctionDuration" value="100"/>
        </broker>
        <agent name="bidder-23"
               class="calana.agents.impl.BasicBidAgent">
           <cluster-ref name="cluster-1"/>
           <prop name="timePrice" value="4.09"/>
           <prop name="bidProbability" value="1.00"/>
           <prop name="rejectProbability" value="0.00"/>
           <prop name="basePrice" value="14.0"/>
        </agent>
        <agent name="bidder-42"
               class="calana.agents.impl.BasicBidAgent">
           <cluster-ref name="cluster-1"/>
           <prop name="timePrice" value="1.25"/>
           <prop name="bidProbability" value="1.00"/>
           <prop name="rejectProbability" value="0.00"/>
           <prop name="basePrice" value="19.67"/>
        </agent>
    </agents>
</calana:experiment>
```

B.2. Workload

Es folgt die DTD zu einer Workload XML-Datei. In der Workload Beschreibung findet man die Benutzer inklusive ihrer Präferenzen, die einzelnen Tasks (seien es sequentielle Tasks oder Koallokations-Tasks) und die zugehörigen Job Beschreibungen.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2
3  <!-- Authors: Mathias Dalheimer, Alexander Petry. -->
4
5  <!-- Version: 0.2 -->
6
7  <!-- A workload consists of users and jobs -->
8  <!ELEMENT gridworkload (users,tasks,jobs)>
9  <!ATTLIST gridworkload
10   load                  CDATA #IMPLIED
11   timecorrection        CDATA #IMPLIED
12   meanRuntime           CDATA #IMPLIED
13   xmlns:gridworkload    CDATA #FIXED "http://calana.net/gridworkload">
14
15 <!-- the users element holds all defined user definitions -->
16 <!ELEMENT users (user+)>
17 <!ATTLIST users >
18
19 <!-- the user element requires an  id and may specify a set of preferences
20 -->
21
22 <!ELEMENT user (pref+)>
23 <!ATTLIST user
24   id            ID      #REQUIRED>
25
26 <!-- a preference  consists of a key which describes  the preference and a
27      weight
28  -->
29
30 <!ELEMENT pref EMPTY>
31 <!ATTLIST pref
32   key           (finishtime|price)      #REQUIRED
33   weight        CDATA   #REQUIRED>
34
35 <!ELEMENT tasks (task+)>
36 <!ATTLIST tasks >
37
38 <!ELEMENT task (part+)>
39 <!ATTLIST task
40   id            ID      #REQUIRED
41   type          (sequence|coallocation) #REQUIRED>
42
43 <!ELEMENT part EMPTY>
44 <!ATTLIST part
45   job-ref       IDREF   #IMPLIED>
```

```
46
47 <!-- this element holds all defined jobs -->
48 <!ELEMENT jobs (job+)>
49 <!ATTLIST jobs >
50
51 <!-- a  job consists of timing information,  such as the point  in time at
52      which the job enters the system and so on.
53
54      a job  can be exluded from  the experiment by  specifying the exclude
55      attribute, this means the job is handled as normal but does not count
56      to statistics and so on.  -->
57
58 <!ELEMENT job (timing,size,meta,cluster,dependency)>
59 <!ATTLIST job
60   id             ID      #REQUIRED
61   exclude        (true|false) "false">
62
63 <!ELEMENT timing EMPTY>
64 <!ATTLIST timing
65   submittime    CDATA   #REQUIRED
66   waittime      CDATA   "0">
67
68 <!-- Here are the jobs dimensions  going to be specified, a job has actual
69 size parameters and requested parameters.  Requested time is also known as
70 'walltime', here it is the attribute 'time' in the requested child.  -->
71
72 <!ELEMENT size (actual,requested)>
73 <!ATTLIST size>
74
75 <!ELEMENT actual EMPTY>
76 <!ATTLIST actual
77   cpus          CDATA   #REQUIRED
78   runtime       CDATA   #REQUIRED
79   memory        CDATA   "-1"
80   avgcputime    CDATA   "-1">
81
82 <!ELEMENT requested EMPTY>
83 <!ATTLIST requested
84   cpus          CDATA   "-1"
85   walltime      CDATA   #REQUIRED
86   memory        CDATA   "-1">
87
88 <!-- The  meta information  hold things such  as the executing  user (uid)
89 with which  group (gid)  and an application  id. It also  contains penalty
90 information, that are those costs a provider has to pay, if he cancels the
91 job due to some failure.  -->
92
93 <!ELEMENT meta (status,userID,groupID?,appID?,penalty?)>
94 <!ATTLIST meta>
95
96 <!ELEMENT status EMPTY>
97 <!ATTLIST status
98   value         CDATA   "0">
```

```
99  <!ELEMENT userID EMPTY>
100 <!ATTLIST userID
101   value         IDREF   #IMPLIED>
102 <!ELEMENT groupID EMPTY>
103 <!ATTLIST groupID
104   value         CDATA   "0">
105 <!ELEMENT appID EMPTY>
106 <!ATTLIST appID
107   value         CDATA   "0">
108 <!ELEMENT penalty EMPTY>
109 <!ATTLIST penalty
110   value         CDATA   "0">
111
112 <!-- The cluster  child holds information about the  cluster at which this
113 job has been scheduled.  -->
114
115 <!ELEMENT cluster EMPTY>
116 <!ATTLIST cluster
117   queueID       CDATA   "-1"
118   partitionID   CDATA   "-1">
119
120 <!-- A job may depend on some other jobs. These information is given here.
121 One  can give  several 'depends'  to specify  how a  job depends  on other
122 jobs. The id of the precursor-job  and an optional amount of time this job
123 has to wait before it can execute.
124
125         The job cannot be started as long  as not all of the jobs this one
126         is depending on have finished.  -->
127
128 <!ELEMENT dependency EMPTY>
129 <!ATTLIST dependency
130   preceedingJobID               CDATA   #REQUIRED
131   timeAfterPreceedingJob        CDATA   "0">
132
133 <!ELEMENT multijob (job+)>
134 <!ATTLIST multijob
135   id    CDATA   #REQUIRED>
```

Literaturverzeichnis

[1] Beowulf Project: `http://www.beowulf.org`.

[2] Calana Grid Worload Generator: `http://developer.berlios.de/projects/cgwg`.

[3] M. Dalheimer. Calana Protocol Definition, Working Paper. 2006.

[4] M. Dalheimer, F. Pfreund, and P. Merz. Agent-based Grid Scheduling with Calana, 2005.

[5] Dror G. Feitelson, Larry Rudolph, and et al. Parallel Job Scheduling - A Status Report.

[6] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and Practice in Parallel Job Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer Verlag, 1997.

[7] Dror G. Feitelson and Edi Shmueli. Backfilling with Lookahead to Optimize the Packing.

[8] Dror G. Feitelson and Ahuva Mu'alem Weil. Utilization and Predictability in Scheduling the IBM SP2 with Backfilling. In *12th Intl. Parallel Processing Symp.*, pages 542–546, 1998.

[9] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[10] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In *CCGRID*, pages 6–7. IEEE Computer Society, 2001.

[11] S. Franklin and A. Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *Intelligent Agents III. Agent Theories, Architectures and Languages (ATAL'96)*, volume 1193, Berlin, Germany, 1996. Springer-Verlag.

[12] Nicholas R. Jennings and Michael J. Wooldridge. Applications of Intelligent Agents. In Nicholas R. Jennings and Michael J. Wooldridge, editors, *Agent Technology: Foundations, Applications, and Markets*, pages 3–28. Springer-Verlag: Heidelberg, Germany, 1998.

[13] William A. Ward Jr., Carrie L. Mahood, and John E. West. Scheduling Jobs on Parallel Systems Using a Relaxed Backfill Strategy. *Lecture Notes in Computer Science*, 2537:88–102, 2002.

[14] A. Mu'alem and D. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling, 2001.

[15] The State Machine Compiler: `http://smc.sourceforge.net`.

[16] Warren Smith, Ian Foster, and Valerie Taylor. Scheduling with Advanced Reservations. *ipdps*, 00:127, 2000.

[17] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Characterization of Backfilling Strategies for Parallel Job Scheduling, 2002.

[18] Anthony Sulistio, Gokul Poduval, Rajkumar Buyya, and Chen-Khong Tham. Constructing a grid simulation with differentiated network service using gridsim. In *International Conference on Internet Computing*, pages 437–444, 2005.

[19] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[20] Achim Streit Voker Hamscher, Uwe Schwiegelshohn and Ramin Yahyapour. Evaluation of Job-Scheduling Strategies for Grid Computing. In R. Buyya and M. Baker, editors, *Grid 2000*, pages 191–202. Springer-Verlag: Heidelberg, Germany, 2000.

[21] M. Wooldridge. Agent-based Computing. *Interoperable Communication Networks*, 1(1):71–97, 1998.

[22] Dmitry Zotkin and Peter J. Keleher. Job-Length Estimation and Performance in Backfilling Schedulers. In *HPDC*, 1999.