GRISCHA

Dokumentation

Inhaltsverzeichnis

1. Einführung

GriScha ist eine Schachsoftware der HTW-Berlin, die ihre Leistungsstärke aus den Verteilungsaspekten des Grid-Computing bezieht. Dabei wird das Problem, möglichst gute Schachzüge zu finden, auf viele unabhängige Nodes im Grid verteilt. Jeder dieser unabhängigen Nodes hat eine einfache Schachlogik, die nur erlaubte Züge betrachtet und diese anhand ihrer Stellung bewertet. D.h. es wird nur die zu ziehende Figur betrachtet und wie ein möglicher Zug sich auswirken würde. Ob dadurch beispielsweise andere Figuren bedroht werden und wie weit die Figuren im gegnerische Feld stehen. Die Folgezüge die daraus berechnet werden erfolgen unabhängig von den anderen Worker Nodes, es findet innerhalb der Worker Nodes keine Kommunikation statt. Aus der Menge der hieraus entstehenden Zugmöglichkeiten wird der Beste Zug gewählt und als Antwort auf dem Schachbrett umgesetzt.
Aus den vorgegangen Schilderung folgt die These, dass viele schlechte Schachspieler eine gute Chance gegen einen guten Spieler haben. Darüber hinaus soll durch GriScha die Kommunikation von Echtzeitanwendungen im Grid analysiert werden können.

1.1. GriScha in a nutshell

Der grundlegende Algorithmus für die Schachengine ist der Alpha-Beta-Algorithmus, ein Algorithmus für Null-Summen-Spiele. Dieser Ansatz ermöglicht es eine Schachpartie als Baum aufzuspannen, sodass es ein großes Problem, gute Züge unter vieler möglichen Zügen zu finden, in kleinere Teilprobleme zerlegt werden kann.
Grundlegend kann GriScha in wie folgt aufgegliedert werden:

- Grid-Infrastruktur
  - Gatekeeper – alloziert Worker Nodes & legt die Pilot Jobs auf Worker Nodes fest
  - Workernodes – hier laufen die Pilot Jobs, nach außen gekapselt
  - Pilot Job – GriScha, Java-App die auf der WN ausgeführt wird, hier wird die Schachlogik aus dem Alpha-Beta-Algorithmus angewandt
- Master Node – Server außerhalb des Grid
  - evaluiert und steuert Engineverhalten
  - verteilt aktuelle Stellung des Schachbretts zu WN
  - Schnittstelle an User-Interface

Vereinfacht kann GriScha wie folgt beschrieben werden:
Der Master Node wird als Server-Instanz außerhalb des Grids gestartet und bekommt einen dedizierten Socket durch den er kommunizieren darf. Die Master Node fungiert als Ansprechpartner aller Worker Nodes aus dem Grid und entscheidet welche Züge gewählt werden. Um den Verteilungsaspekt anwenden zu können wird eine Grid-Infrastruktur benötigt, die es erlaubt Anwendungen/ Jobs in die Grid zu versenden. Die HTW-Berlin gehört zum DECH-Verband, d.h. stellt selber Ressourcen zur Verfügung und kann Resources aus dem DECH-Verbund nutzen. Mittels einer Middleware können Anwendungen (meist als Pilot Job, oder nur Job bezeichnet) in das Grid geschickt werden. Dies geschieht nicht direkt sondern durch die Middleware gLite (Lightweight Middleware for Grid Computing), dabei erhalten die Jobs beim Submit in die Grid die Adresse und den Port der Master Node, sodass diese Pilot Jobs von den Worker Nodes aus mit der Master Node kommunizieren können. Wie die einzelnen Pilot Jobs verteilt ist für den Anwender dabei vollständig transparent, die Middleware setzt, nach Möglichkeiten, das um, was der User in seiner Job Discription gefordert hat. Wenn die Jobs erfolgreich durch den Gatekeeper der Grid-Middleware auf die Worker Nodes verteilt worden sind wird auf der Worker Node die beschriebene Instanz, also der Pilot Job, gestartet. Anschließend melden sich die laufenden Instanzen bei der Master Node. Der Master Node verwaltet die ihn bekannten Worker Nodes, sodass er sowohl eingehende Ergebnisse als auch die aktuelle Stellung des Schachbretts in die Grid kommunizieren kann. Weiterhin sorgt der Master Node dafür, dass die eingehenden Ergebnisse evaluiert werden und das jeweils beste, vorliegende Ergebnis als Zug angenommen wird. Daher muss der Master Node auch mit dem User-Interface kommunizieren, sodass die evaluierten Züge für den Nutzer

sichtbar werden. Für das User-Interface können Xboard oder PyChess verwendet werden, da beide Anwendungen über das Winboard-Protokoll angesprochen werden können.

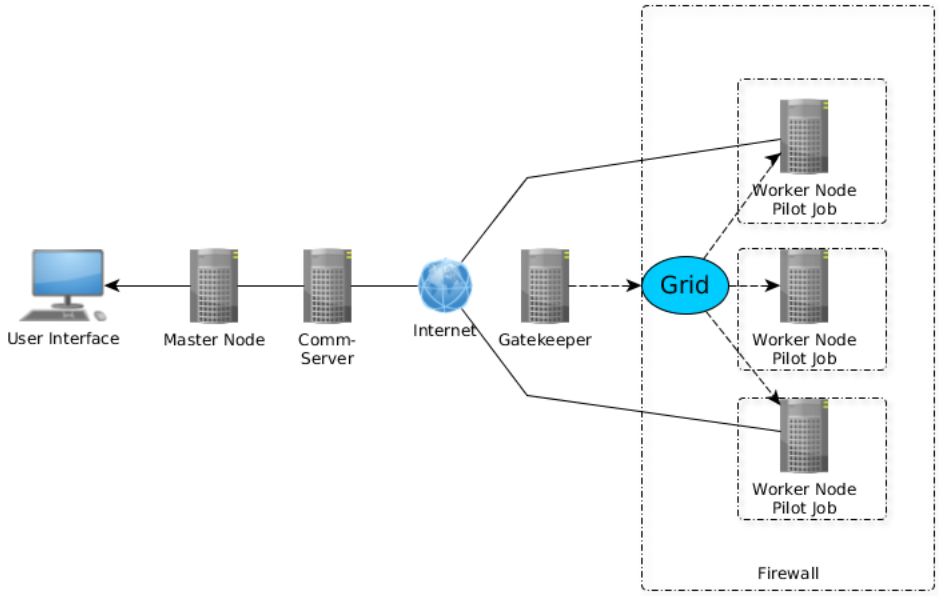


Abbildung 1: allgemeiner Aufbau GriSchas

1.2. Ziel

Das Ziel GriSchas ist es zu zeigen, ob eine einfache Schachengine, durch den Verteilungsaspekt auf viele Rechner, trotzdem gut spielen kann. Um dies zu zeigen wird GriSchas Kommunikation als Grid-Echtzeitanwendung untersucht und optimiert, sodass durch die verbesserte Kommunikation, mehr Ressource nutzbar sind und somit die Spielstärker erhöht wird.
Durch Erhöhung der genutzten Ressourcen aus dem Grid kann wiederum geschlossen werden, wie leistungsfähig die genutzten Kommunikationsprotokolle sind und ob die Nutzung eine Optimierungsmöglichkeit darstellt.

2. Quelltexte & Setup

GriScha kann als Git-Repository von Grischa-Server bezogen werden. Die aktuelle Version, sowie die Dokumentation ist unter folgender Adresse zu finden [1]:

```
1 git clone git@grischa.f4.htw-berlin.de:/grischa.git
```

Der Build-Process und wie die Redis-Version zu benutzen ist folgt im Abschnitt 2.1. Neben diesem aktuellen Branch dessen Kommunikation mittels Redis realisiert wurde gibt es noch den Legacy-Branch:

```
1 git clone git@grischa.f4.htw-berlin.de:/grischa_legacy.git
```

In diesem Repository sind folgende Branches gelistet, sowie die gesamte History der Entwicklung:

- origin/master
- origin/xmpp
- origin/redis
- origin/kk-database
- origin/socketio
- origin/monte-carlo

Weiterhin ist noch eine Repository verfügbar das die bisherigen Abschlussarbeiten und Paper zu Grischa enthält:

```
1 git clone git@grischa.f4.htw-berlin.de:/grischa_library.git
```

Alle hier gelisteten Branches benötigt grundlegend Java (ab JDK6) und ein Winboard fähiges User Interface wie Xboard oder PyChess. Darüber hinaus werden keine Abhängigkeiten in den Branches mitgeliefert. Die einzelnen Branches wurden mit unterschiedlichen IDE's und Build-Tools entwickelt. Hinweise, welche Abhängigkeiten und Build-Tools zu verwenden sind gibt es zumeist in den jeweiligen *README.md* Dateien.

2.1. Grischa

Details zu Architektur und Implementierung sind in den Kapiteln 5.4 und 6.1 zu finden. Für dieses Repository gibt es auch im Ordner `../handbook/uml/` die entsprechenden Klassendiagramme.
Folgendes sollte vorher auf dem System verfügbar sein:

- Ant
- Redis

[1] Das Passwort gibt es auf Anfrage

- folgende Abhängigkeiten müssen aufgelöst werden [2].
  - commons-cli-1.2.jar
  - commons-pool2-2.0.jar
  - jedis-2.1.0-sources.jar
  - junit-4.11.jar
  - log4j-1.2.15.jar
  - smack.jar
  - smackx.jar
  - smackx-debug.jar
  - smackx-jingle.jar
- die im Paket org.json [3] vorhandenen Klassen müssen ebenfalls kompiliert werden, dies sollte durch ant geschehen.

Die hier gelisteten Abhängigkeiten können so lange die Migration von Ant auf Maven nicht abgeschlossen ist auch hier bezogen werden:

Listing 1: Redis Abhängigkeiten

```
1 git clone git@grischa.f4.htw-berlin.de:grisha_dependencies
```

Die zusätzlichen Bibliotheken müssen im Verzeichnis `./libs` abgelegt werden. Danach kann mit auf den aktuellen Stand gebracht werden.

Listing 2: Laden der Submodule

```
1 git submodule update --init
```

Und anschließen kann mithilfe von Ant GriScha gebaut werden:

Listing 3: Ant Build

```
1 ant gnode gclient
2 # bzw in einem Schritt
3 ant grischa
```

Das erste baut nur die gnode und den gclient, das zweitere baut alle Targets.
Es werden zwei bzw drei JAR-Archive erstellt. Dabei ist *gnode.jar* für die Pilot Jobs in der Grid erstellt worden und ein Client *gclient.jar* für das User-Interface zu Xboard.
Der Deploy erfoglt für einen lokalen Aufbau wird folgendermaßen ausgeführt:

Listing 4: lokaler Deploy

```
1 java -jar gnode.jar
2 xboard -fcp "java -jar gclient.jar"
```

[2] wird demnächst auf Maven geändert, sodass dies nicht mehr händisch erledigt werden muss, hat aber noch Migrationsschwierigkeiten

[3] `../src/org/json`

2.1.1. Redis im Mobile Computing Labor

Bevor Testläufe im Labor gestartet werden können, muss der f4-Account von den Laboringenieuren zur Moco-Gruppe hinzugefügt werden. Die Testläufe sollten außerhalb der Vorlesungs-/ Übungszeiten umgesetzt werden, d.h. etwa $8^{\underline{30}} - 08^{\underline{00}}$ Uhr.
GriScha kann für Testläufe im Labor WHC-625 ausgeführt werden, dabei kann wie folgt vorgegangen werden: Die Anmeldung erfolgt via SSH. Dabei wird durch das startLab-Skript per Wake On LAN alle im Labor befindlichen Rechner hochgefahren. Achtung: Die Rechner werden dabei mit Ubuntu gebooted bzw. laufende Rechner werden rebooted! Beim Stoppen des Labors sollte darauf geachtet werden, dass keine anderen User bzw. deren Prozessen noch laufen, da stopLab ein poweroff an die Laborrechner sendet.

Listing 5: Starten der MoCo-Labors

```
1 ssh sxxxxx@gridgateway.f4.htw-berlin.de
2 startLab #fuer das Starten
3 sudo stopLab #fuer das Herunterfahren
```

Nachdem die Rechner im Labor Online sind, sollten 21 Rechner zur Verfügung stehe, Siehe Tabelle 1, die nun wiederum auch per SSH erreichbar sind. Nun kann ein Rechner

Tabelle 1: MoCo-Rechner-Pool

IP-Extern	141.45.154.136 - 141.45.154.156
IP-Intern	10.10.10.1 - 10.10.10.21
Rechnername	pluto1 - pluto21
HTW-DNS-Suffix	.f4.htw-berlin.de
Labor-DNS-Suffix	.mocolabor.f4.htw-berlin.de
Offene Ports	Ports 4710-4720

als Master Node fungieren und die restlichen Rechner als Worker Nodes.

2.1.2. Redis im Grid

TODO

2.2. Grischa Legacy

Der Master Branch des Legacy Repositories nutzt das Protokoll Simon, um zwischen Worker Nodes und Master Node zu kommunizieren. Es wird ein Build-Tool wie Gnu Make oder CMAKE benötigt. Der Build erfoglt wie folgt:

Listing 6: Build mit Make

```
1 cd grischa_legacy
2 make clean
3 make jar
```

Es wird ein JAR-Archiv erstellt, dies kann lokal wie folgt deployed wird:

Listing 7: Deploy SIMON Teil 1

```
mkdir -p bench/version_1 bench/version_2 \
&& echo bench/version_1 bench/version_2 | xargs -n 1 cp GriScha.jar
```

Listing 8: Deploy Server

```
java -jar bench/version_1/GriScha.jar server -p 4711
```

Listing 9: Deploy Worker Node

```
java -jar bench/version_2/GriScha.jar wn -s 127.0.0.1 -p 4711
```

Listing 10: Deploy Xboard User Interface

```
xboard -tc 15 -fcp "java -jar \
>>absolute_pathToGriScha<</bench/version_1/GriScha.jar \
 xboard -s 127.0.0.1 -p 4711" -fd \
>>pathToGriScha<</bench/version_1
```

Im zweiten Teil wird GriScha als Server, also als Master Node und einmal als Worker Node aufgesetzt. Wobei die Worker Node sich bei der Master Node registriert. Im letzten Schritt wird schließlich Xboard gestartet. Die angegebenen Parameter sorgen für die Kommunikation zwischen Master oder User-Interface.

2.2.1. SIMON im Mobile Computing Labor

2.2.2. SIMON im Grid

Für die Ausführung im Grid: – Änderung der Grid-Middleware – muss noch angepasst werden!

2.3. XMPP Branch

TODO

2.4. ZMQ

TODO

3. Grid-Computing

> Per Definition ist das Grid-Computing eine Technik zur Integration und gemeinsamen, institutionsübergreifenden, ortsunabhängigen Nutzung verteilter Ressourcen auf Basis bestehender Kommunikationsinfrastrukturen wie z.B. dem Internet.[BBKS15][S. 447]

Die nutzbaren Ressourcen werden durch VO (Virtuelle Organisationen) bereitgestellt, sodass diese den Nutzern dynamisch zur Verfügung stehen. Die Verfügbaren Ressourcen sind im Grid verschiedenen Einrichtungen zuzuordnen, die unabhängig voneinander administriert werden.
Nach Foster muss ein Grid folgende drei Kriterien erfüllen:

1. Ein Grid koordiniert unterschiedlichste Arten von dezentralen Ressourcen. Dazu gehören Standard-PCs, Workstations, Großrechner, Cluster, usw. Benutzergruppen sind in sog. Virtuellen Organisationen (Virtual Organisations) zusammengefasst. Es kann kooperativ auf die Ressourcen des Pools zugegriffen werden.

2. Grids verwenden offene, standardisierte Protokolle und Schnittstellen. Da in einem Grid wichtige Punkte wie Authentifikation , Autorisierung und das Auffinden und Anfordern von Diensten eine fundamentale Rolle spielen, müssen die verwendeten Protokolle und Schnittstellen offen und standardisiert sein. Ansonsten handelt es sich um ein applikationsspezifisches System und nicht um ein Grid.

3. Grids bieten unterschiedliche, nicht-triviale Dienstqualitäten an. Die verschiedenen Ressourcen eines Grids offerieren zusammen genommen eine Vielzahl von Möglichkeiten im Bezug auf Durchsatz, Sicherheit, Verfügbarkeit und Rechenleistung. Der Nutzen der zu einem Grid zusammengeschlossenen Systeme ist größer als die Summe der einzelnen Teile. ([Fos02])

Zwei zentrale Konzepte des Grid Computings sind die Virtual Envoirement und Virtual Organization. Virtual Envoirement meint, dass alle Systeme in einer Grid-Infrastruktur zu einem „Pool“von Ressourcen gehören. Dieser Pool ist für den Nutzer transparent nutzbar, der Nutzer muss kein Wissen über die Lokation, Allokation etc. besitzen, um Ressourcen nutzen zu können. Virtual Organization bezeichnet die Nutzung der Verfügbaren Ressourcen, zumeist wird ein kooperatives Modell der Ressourcennutzung damit beschrieben.
Computing-Grid kann zusammenfassend als ein verteiltes System beschrieben werden, dass Rechenleistung den Transparenzkriterien verteilter Systeme entsprechend [BBKS15][S. 27f], [TS13][S. 4f] [4] anbietet. Dabei ist wohl der wichtigste Aspekt, dass es im Grid keine einheitliche Vorgaben in Bezug auf Hardware, Software und Infrastruktur gibt. Ergo ist das Grid eine sehr heterogene Umgebung, die nur durch eine vermittelnde Schicht, der Grid-Middleware, ansprechbar ist.

[4] Orts-,Zugriffs-, Persistenz-, Nebenläufigkeits-, Skalierungs-, Migrations-, Fehler- und Ausfall-, Verteilungs- und Leistungstransparenz

3.1. Grid Architektur

3.2. Grid Computing

In Abbildung 2 kann der Ablauf einer Grid-Session nachvollzogen werden. Folgender Ablauf lässt sich daraus herleiten:
Der User schickt durch die Gri-Middleware Aufgaben ins Grid, diese werden von einem Broker verwaltet. Der Broker sucht aus einem Katalog – dem Resourcen Pool, entsprechend den Anforderungen des Jobs, Worker Nodes die im Grid verfügbar sind. Da passende Worker Nodes in unterschiedlichen „Regionen“ des Grids liegen können muss der Broker entsprechend die Jobs an die jeweiligen Grid Gateways weiterreichen. Die Gateways wiederum senden nun den Job an die jeweiligen Worker Nodes, wo der Job schließlich ausgeführt wird.

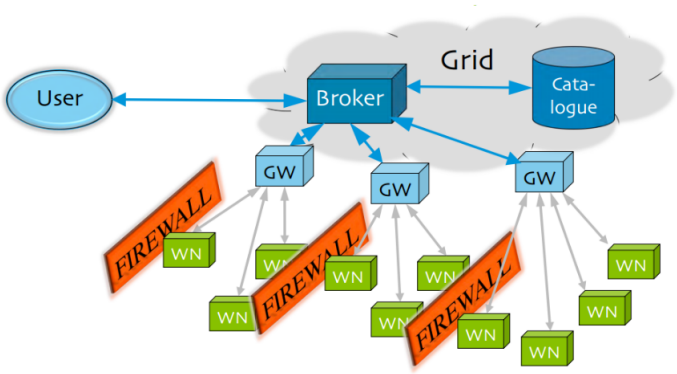


Abbildung 2: Aufbau eines Grid [Heß13]

3.3. Grid-Middleware – EMI

European Middleware Initiative – TODO

4. Schachlogik

Grundsätzlich braucht ein Schachprogramm zwei wichtige Komponenten: einen Zuggenerator und eine Bewertungsfunktion. Wobei der Zuggenerator den Spielbaum aufspannt und die Bewertungsfunktion die jeweilige Spielstellungen analysiert.

4.1. Nullsummenspiele

Holler und Illing nach ist ein Nullsummenspiel wie folgt definiert: „Nullsummenspiele beschreiben in der Spieltheorie Situationen, also Spiele im verallgemeinerten Sinne, bei denen die Summe der Gewinne und Verluste aller Spieler zusammengenommen gleich null ist.“[HI13][S. 55]

4.2. Zuggeneratoren & Spielbaum

Ein Zuggenerator ist eine Funktion, die ein existierendes Spielbrett entgegennimmt und daraus alle erlaubten Züge einer Partei als mögliche Folgezüge zurückgibt. Dieses Unterfangen geschieht rekursiv, sodass ein Spielbaum einer bestimmten Tiefe generiert werden kann. Die Rekursion findet je Rekursionsschritt abwechselnd für beide Parteien statt. Die hierbei entstehende Baum hat eine bestimmte Tiefe, jede Tiefe repräsentiert einen Halbzug. Daher besteht ein durchgerechneter Zug aus zwei Halbzügen, jeweils für beide Parteien einen Halbzug. Ein einfaches Beispiel ist in Abbildung 3 zu sehen, da je Partei nur 2 Mögliche Züge vorhanden sind. Bei einer „echten“ Schachpartie gibt es in der Ausgangstellung 20 Zugmöglichkeiten und nach zwei Halbzügen gibt es bereits 400 mögliche Züge. Dabei wächst die Zahl der möglichen Halbzüge exponentiell, wenn auch viele Züge als nicht sinnvoll erachtet werden können. Die Blätter des Spielbaums erlauben es der Bewertungsfunktion zu errechnen, ob eine Stellung sinnvoll ist, oder nicht. Beim Minimax-Algorithmus geschieht dies durch Maximum Zahlen für gute Züge (Weiß ist im Vorteil) und Minimum für schlechte Züge (Schwarz ist im Vorteil).

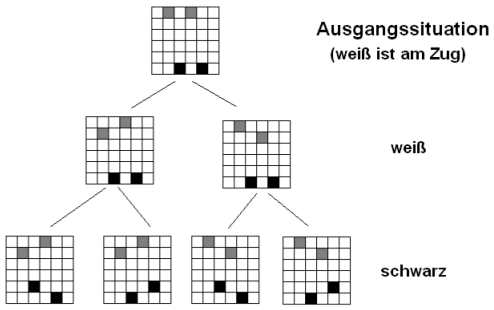


Abbildung 3: Einfacher rekursver Aufruf für zwei Halbzüge

4.3. Minimax-Algorithmus

Der Minimax-Algorithmus ist ein Algorithmus für die Suche nach optimale Zugmöglichkeiten. Es wird dabei davon ausgegangen, dass der Spieler jeweils immer seinen stärksten möglichen Zug als Folgezug wählen würde. Weiß würde also immer den maximal positiven und Schwarz maximal negativen Zug wählen.
Der Algorithmus kann wie folgt beschrieben werden: Der Zuggenerator erhält das aktuelle Spielbrett und prüft alle legalen Züge. Dementsprechend wird der Spielbaum aufgespannt, das jeder mögliche Folgezug ein Knoten unterhalb der Ausgangsstellung erhält.

Mit diesen Ausgangszügen wird dann ein rekursiver Abstieg bis zu einer bestimmten Tiefe vorgenommen und an den werdeh Blättern jeweils die Stellungen bewertet. Im rekursiven Aufstieg wird anschließend je Tiefe abwechselnd immer Minimiert oder Maximiert. So erhalten die Elternknoten, je nach Spielpartei, das Maximum bzw. Minimum der Kindknoten.

4.3.1. Bewertungsfunktionen

Die Bewertungsfunktion hat die Aufgabe ein gegebenes Spielbrett zu analysieren und entsprechend der Spielstellung einen Wert zurückzugeben.
Als Grundidee kann die Bewertungsfunktion nur drei verschieden Werte zurückgeben, entweder 1 für weiß gewinnt, -1 für schwarz gewinnt und 0 für „unentschieden“[5]. Um so bewerten zu können wird vorausgesetzt, dass die Blätter die Endstellung (das Spiel ist zu Ende) repräsentieren. Da dies zwar möglich wäre, aber ein Zeit- und Speicherproblem nach sich zieht, muss die Bewertungsfunktion auch Stellungen bewerten, bei denen das Spiel noch nicht zu Ende ist.
Um nun auch Stellung bewerten zu können, bei denen das Spiel nicht zu Ende ist, werden Heuristiken eingesetzt. In Grischa geschieht dies durch eine Mischung aus Materialwerten der auf dem Spielfeld befindlichen Figuren und der Stellung der Figuren zueinander. Es wird dabei den Figuren ein Wert zugewiesen, Figuren mit höherem strategischem Wert erhalten einen größeren Materialwert (eine Dame ist mehr Wert als ein Turm, der wiederum ist mehr Wert als ein Läufer etc.). Weiterhin wird neben diesen Materialwerten auch die Positionierung der Figuren zueinander berücksichtigt. Also wo steht eine Figur und wie ist deren Stellung in Bezug auf die eigenen Figuren, welche Figuren werden durch sie gedeckt, als auch gegenüber dem Gegner, welche Figuren werden angegriffen, gefesselt, liegt eine Gabel vor etc. Auch der Angriff auf eine Figur und wie diese gedeckt wird wird mit einbezogen. So erhält das jeweilige Spielfeld im Blatt entsprechend seiner Situation ein Bewertung. Diese errechnete Bewertung wird dann gemäß dem Minimax im rekursiven Aufstieg nach oben gegeben.

4.4. Alpha-Beta-Algorithmus

4.5. Alpha-Beta-Algorithmus Implementation

[5] gibt es eigentlich nicht, denn entweder gibt es ein Remis, ein Patt, kein Matt, die dreifache Stellungswiederholung oder die 50-Züge-Regel

5. Architektur

Abbildung 4 zeigt die generelle Architektur GriSchas, wobei die als Comm bezeichnete Komponente durch die jeweils eingesetzte Lösung für den Nachrichtenaustausch auszutauschen ist.
GriSchas Architektur lässt sich auf folgende Komponenten runterbrechen:

- **GriScha**
  - **master** Master Node – Hauptprogramm, interagiert mit dem User Interface
  - **worker** Worker Node – Programm das in der Grid auf den Worker Nodes, als Pilot Job ausgeführt wird
- **Kommunikation**
  - **server** verteilt Nachrichten an Clients & verarbeitet Antworten
  - **client** erhält Nachrichten des Servers & verarbeitet Aufgaben aus Nachrichten

  [Ste14][S. 20]

Die Master Node (GMaster) ist die zentrale Ansprechstelle zwischen User Interface und der Kommunikation. Hauptaufgabe des Masters ist es, die aktuelle Stellung der Schachpartie festzustellen und mittels der Comm Servers die daraus abgeleitet Aufgaben an die Worker Nodes zu senden. Die Worker Nodes evaluieren ankommende Anfrage der Master Node und berechnen jeweils mögliche Folgezüge. Diese werden wiederum via der Comm Servers an die Master Node geschickt. Nun hat die Master Node die Aufgabe alle ankommenden Nachrichten zu evaluieren und das beste Ergebnis an das User Interface zu geben.

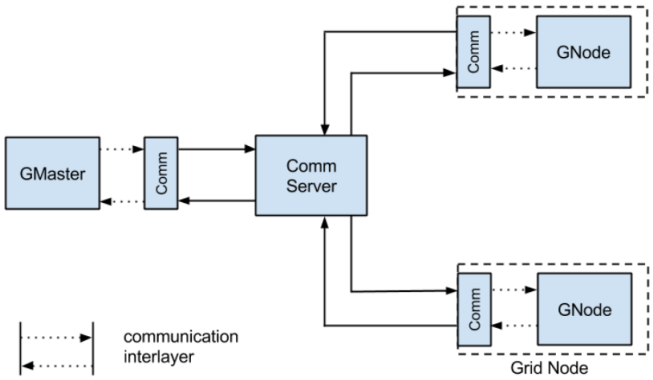


Abbildung 4: allgemeine Architektur [Ste14][S. 21]

5.1. Kommunikationsablauf

Abbildung 5 legt den Ablauf einer Kommunikation zwischen den beteiligten Komponenten dar. Es kann wie folgt beschrieben werden:

1. Node1 registriet sich beim Communication Server und ist anschließend verfügbar
2. Node2 registriet sich beim Communication Server und ist anschließend verfügbar
3. der Master Node fragt nach allen verfügbaren Nodes
4. der Cmmunication server sendet alle verfügbaren Nodes
5. der Master Node sendet eine anfrage an mit dem Payload 1 für Node1 and den Communication Server
6. der Master Node sendet eine anfrage an mit dem Payload 2 für Node2 and den communication server
7. der Communication server sendet die Anfrage mit Payload 1 an Node 1
8. der Communication server sendet die Anfrage mit Payload 2 an Node 2
9. Node2 sendet Ergebnis 2 an den Communication Server
10. der Communication Server sendet das Ergebnis 2 zum Master Node.
11. Node1 sendet Ergebnis 1 an den Communication Server

5.2. Simon

Die erste Version GriSchas nutzte für die Kommunikation SIMON [6]. In HIER VERWEIS AUF QUELLE FÜR SIMON ABSCHLUSSARBEIT EINFÜGEN gibt es einen detaillierteren Überblick zur Architektur. Abbildung 6 zeigt die Architektur mit Simon als Kommunikationsschicht.

5.3. XMPP

–TODO

5.4. Redis

Redis ist eine Open Source Key-Value-Datenbank, die „in-memory“ betrieben wird. D.h. alle Daten sind im RAM persistiert und haben dadurch eine sehr hohe Performance. Dabei ist Redis keine reine Key-Value-Speicher

[6] Simple Invocation of Methods Over Network

5.4.1. Jedis

Jedis ist eine Open Source Bibliothek, die es erlaubt sich mit Redis zu verbinden und kann mit dem Publish-Subscribe Nachrichten Muster umgehen.

5.5. Publish-Subscribe via JedisPool

5.6. ZMQ

–TODO

6. UML

Alle Klassen können als UML-Diagramme im Git-Repository im Ordner `grischa/handbook/uml` gefunden werden. Die wichtigsten Komponenten werden im Folgenden hier beschrieben, sodass die Architektur etwas konkreter nachvollzogen werden kann.

6.1. Redis UML

Der GClient ist eine der Klassen, die am wichtigsten ist. In dessen Konstruktor folgendes initialisiert wird:

```
1  public class GClient implements Runnable {
2      public GClient() {
3      }
4      @Override
5      public void run() {
6          // Boot registry and client
7          GWorkerNodeRegistry.getInstance();
8          GClientConnection.getInstance();
9
10         WinboardCommunication cli = new
               WinboardCommunication();
11         cli.run();
12     }
13
14     /**
15      * @param args
16      */
17     public static void main(String[] args) {
18         GClient client = new GClient();
19         new Thread(client).start();
20     }
21 }
```

A. UML

B. Ablaufdiagramme

C. Literatur

Literatur

[BBKS15] Bengel Bengel, Baun Baun, Kunze Kunze, and Stucky Stucky. *Masterkurs Parallele und Verteilte Systeme - Grundlagen und Programmierung von Multicore-Prozessoren, Multiprozessoren, Cluster, Grid und Cloud*. Springer-Verlag, Berlin Heidelberg New York, 2. aufl. edition, 2015.

[Fos02] Ian Foster. What is the grid? a three point checklist. *Argonne National Laboratory & University of Chicag*, 2002.

[Heß13] Hermann Heßling. Big data and real-time computing (keynote). In *IEEE 15. International Conference on Modelling and Simulation*, 2013.

[HI13] Manfred J. Holler and Gerhard Illing. *Einf"hrung in die Spieltheorie -*. Springer-Verlag, Berlin Heidelberg New York, 2013.

[Ste14] Philip Stewart. Message-based communication in real-time grid computing. Master's thesis, Hochschule für Technik und Wirtschaft Berlin, Dezember 2014.

[TS13] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2013.

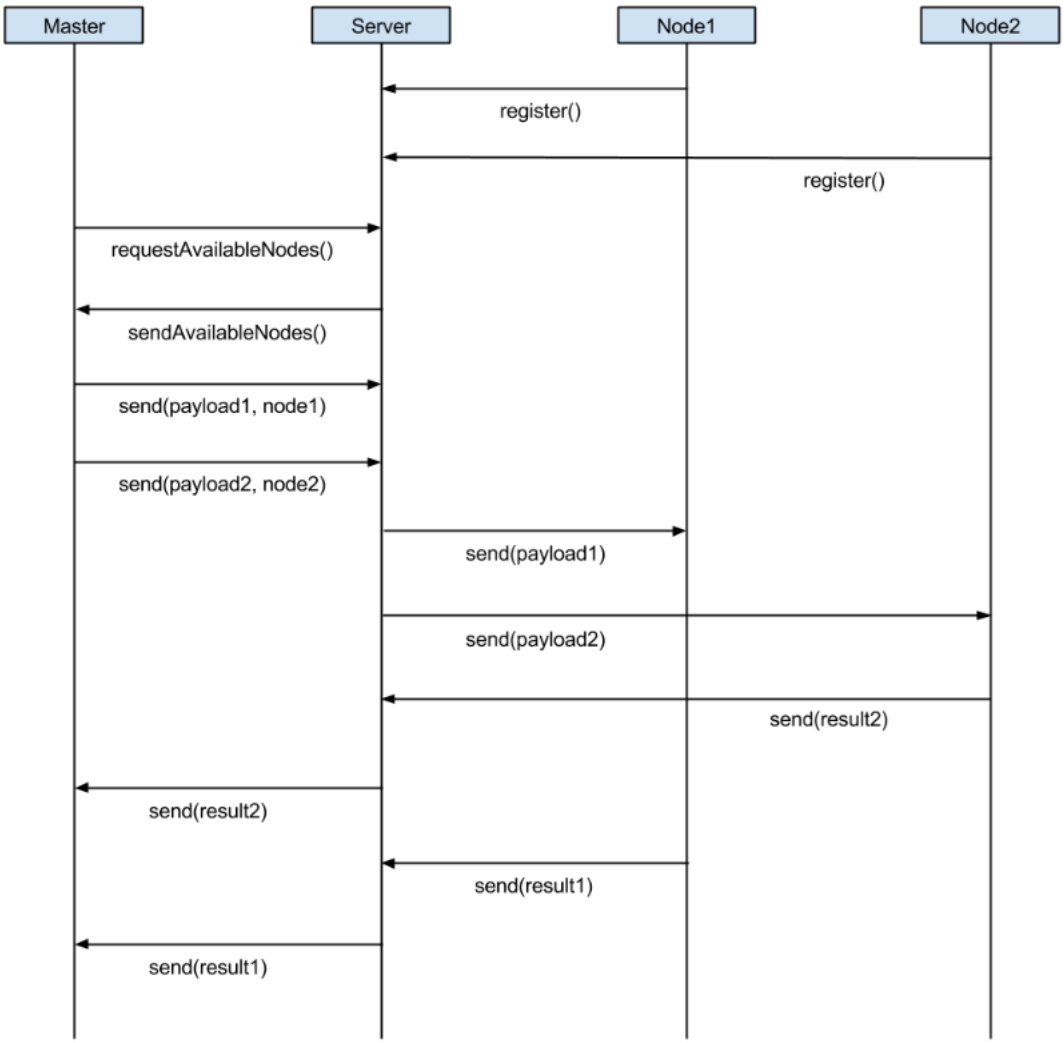


Abbildung 5: Ablaufdiagramm der Kommunikation nach [Ste14]

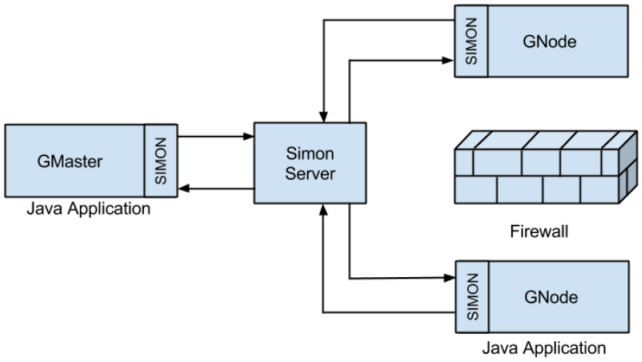
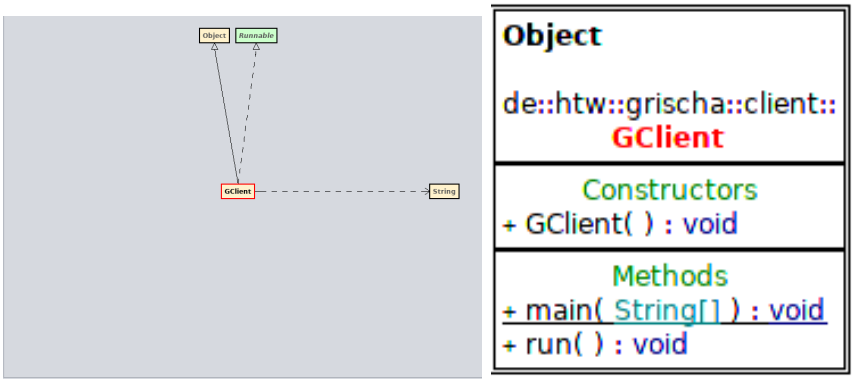
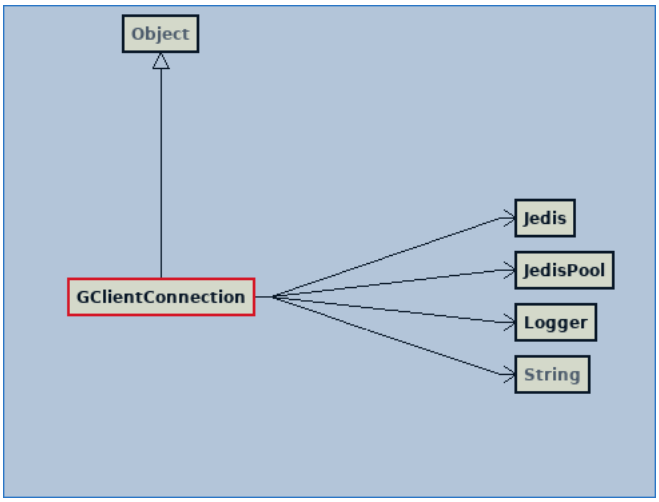


Abbildung 6: allgemeine Architektur mit SIMON [Ste14][S. 19]

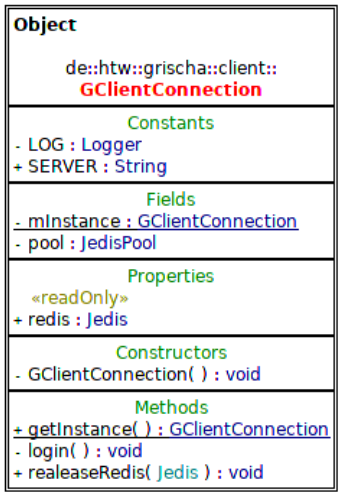


(a) UML (b) Klasse GClient

Abbildung 7: GCLient

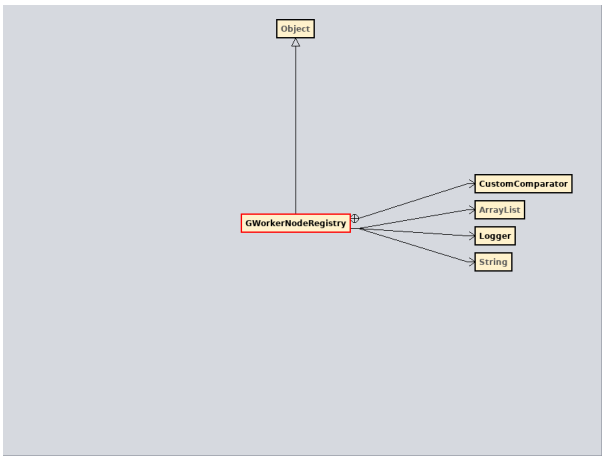


(a) UML

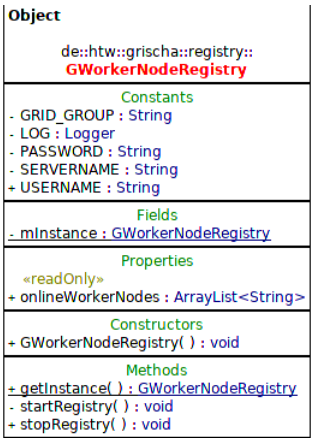


(b) Klasse GClientConnection

Abbildung 8: Singleton GclientConnection die den Pool für Redis-Verbindugen bereitstellt



(a) UML



(b) Klasse GWorkerNodeRegistery

Abbildung 9: Singleton GclientConnection die den Pool für Redis-Verbindugen bereitstellt