GriScha

Dokumentation

BT

7. November 2016

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 guter Schachzüge zu finden, auf viele unabhängige Nodes im Grid verteilt. Jeder dieser unabhängigen Nodes hat eine einfache Schachlogik, die Folgendezüge berechnet, ohne dabei mit anderen Worker Nodes zu kommunizieren. Aus der Menge an berechneter Zugmöglichkeiten wird der Beste Zug gewählt. 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.

1.1 GriScha in a nutshell

GriScha ist eine Schachengine die ihre Leistungsstärke aus dem Verteilungsaspekt eines verteilten Systems bezieht, dem Grid. Der grundlegende Algorithmus für die Schachengine ist der Alpha-Beta-Algorithmus, ein Algorithmus für Null-Summen-Spiele, der es ermöglicht eine Schachpartie als Baum aufzuspannen, sodass es möglich ist ein großes Problem in kleinere Teilprobleme zu zerlegen.
Grundlegend kann GriScha in drei Teile 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.

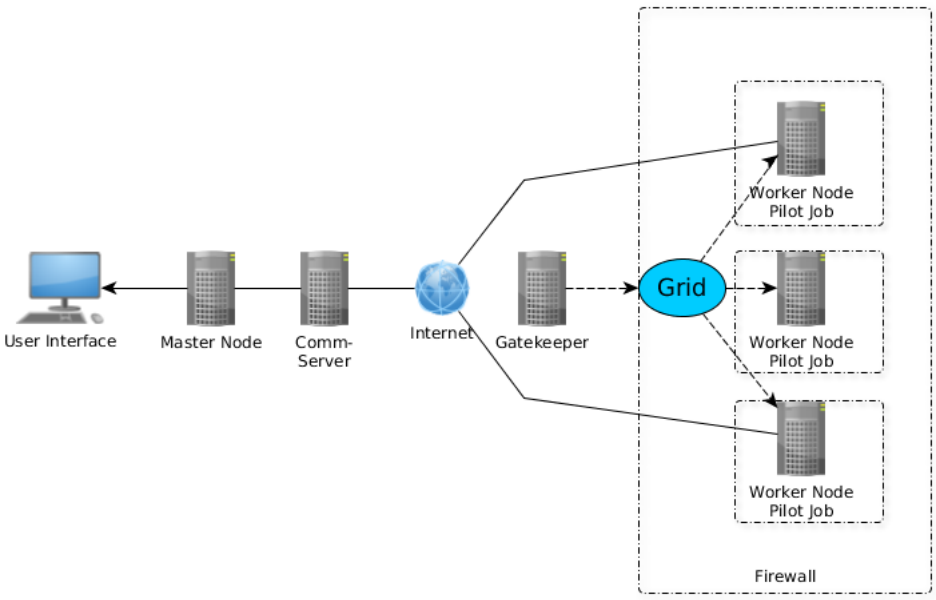


Abbildung 1: allgemeiner Aufbau GriScha

1.2 Ziel

Ziel ist zu zeigen, ob eine einfache Schachengine durch den Verteilungsaspekt auf viele Rechner, trotzdem gut spielen kann. Um dies zu zeigen wird GriSchas Kommunikation als Echtzeitanwendung untersucht und optimiert, sodass durch die verbesserte Kommunikation, mehr verfügbare Ressource verfügbar sind. Das wiederum soll laut These die Spielstärke verbessern.

2 Installation & Setup

GriScha kann als git Repository bezogen werden:

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

Das Passwort gibt es auf Anfrage. Im Repository sind folgende Branches gelistet:

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

Alle hier gelisteten Branches benötigt grundlegend Java (ab JDK6), darüber hinaus werden keine Abhängigkeiten der Branches mitgeliefert. Die einzelnen Branches wurden mit unterschiedlichen IDE und Build Tools entwickelt, Hinweise darauf gibt es zumeist in der jeweiligen README.md.

2.1 Master Branch

Der Master Branch nutzt das Protokoll Simon um zwischen Worker Nodes und Master Node zu kommunizieren. Der Build erfoglt wie folgt:

Listing 1: Build mit Make

```
% make clean
% make jar
```

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

Listing 2: Deploy SIMON

```
% cd >>path_to_GriScha<<
% mkdir -p bench/version_1 bench/version_2 \
&& echo bench/version_1 bench/version_2 | xargs -n 1 cp GriScha.jar
```

```
% java -jar bench/version_1/GriScha.jar server -p 4711 #master node
% java -jar bench/version_2/GriScha.jar wn -s 127.0.0.1 -p 4711 #worder node
% >>path_to_xboard<< -tc 15 \
-fcp "java -jar "path_to_grischa"/bench/version_1/GriScha.jar xboard \
-s 127.0.0.1 -p 4711"
\ -fd >>path_to_GriScha</bench/version_1
```

Für die Ausführung im Grid:

2.2 Redis Branch

Folgendes sollte vorher auf dem System verfügbar sein:

- Ant – Build Tool
- Redis
- folgende Abhängigkeiten müssen aufgelöst werden:
  - 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 in org.json vorhandenen Klassen müssen ebenfalls kompiliert werden

Die zusätzlichen Bibliotheken müssen im Verzeichnis „libs“ abgelegt werden.

Listing 3: JSON Build

```
% git submodule update --init
```

Mit Ant kann GriScha gebaut werden:

Listing 4: Build

```
% ant gnode gclient # or
% ant grischa #  to build all at once
```

Es werden zwei JAR-Archive erstellt, eines gnode.jar für die Pilot Jobs in der Grid und ein Client für das User-Interface zu Xboard. Der Deploy erfoglt folgendermaßen:

Listing 5: Deploy

```
% java -jar gnode.jar --user username@example.org --password password
% xboard -fcp "java -jar gclient.jar"
```

2.3 XMPP Branch

2.4 ZMQ

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 zusammengefasst.

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])

In Abbildung 2 kann der Aufbau einer Grid-Infrastruktur nachvollzogen werden.

3.1 Grid Middleware gLite

–TODO

4 Schachlogik

–TODO

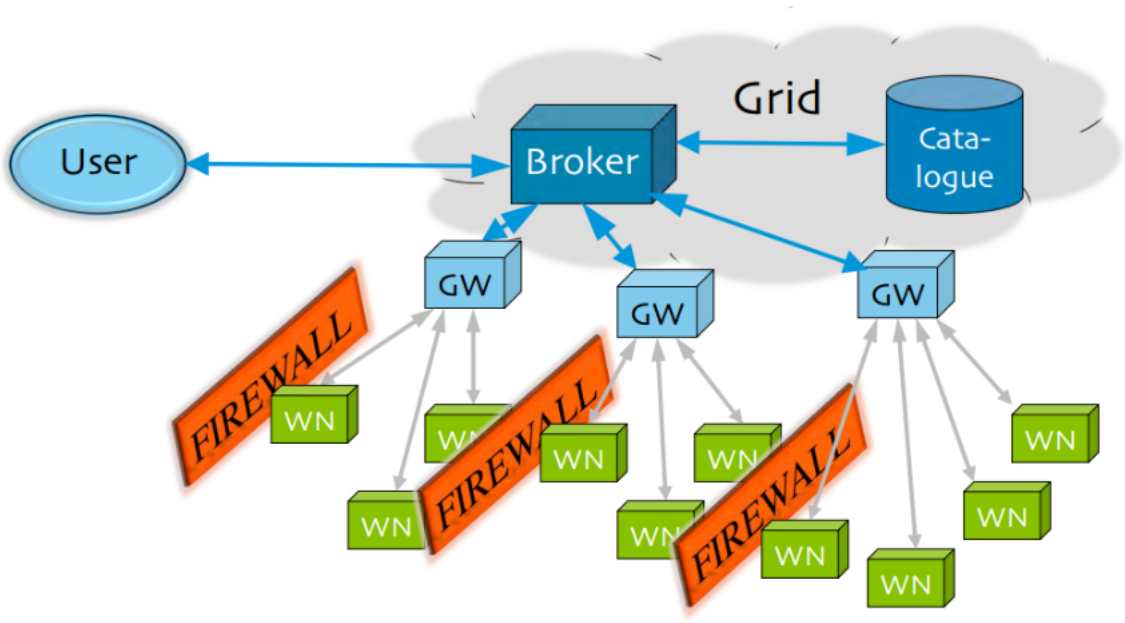


Abbildung 2: Aufbau eines Grid [Heß13]

4.1 Mini-Max

4.2 Alpha-Beta-Algorithmus

5 Architektur

Abbildung 3 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 – Hauptprgramm interagiert mit dem User Interface
  - **worerk** 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 einkommenden Nachrichten zu evaluieren und das beste Ergebnis an das User Interface zu geben.

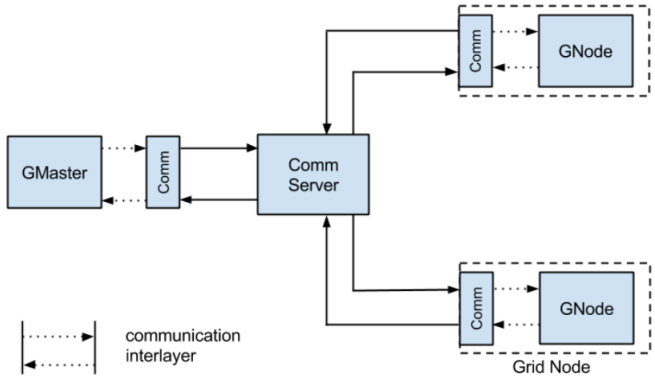


Abbildung 3: allgemeine Architektur [Ste14][S. 21]

5.1 Kommunikationsablauf

Abbildung 4 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

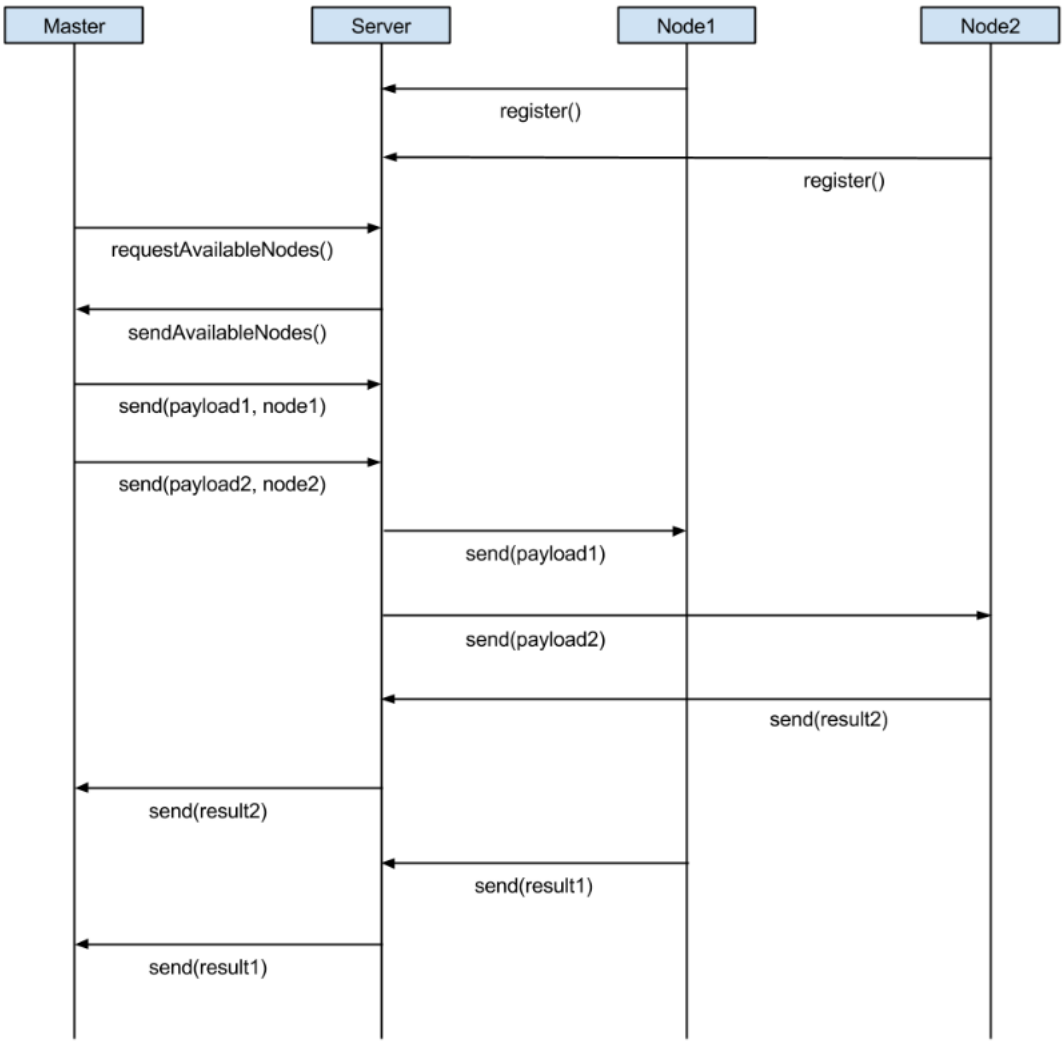


Abbildung 4: Ablaufdiagramm der Kommunikation nach [Ste14]

5.2 Simon

–TODO Abbildung 5 zeigt die Architektur mit Simon als Kommunikationsschicht, wobei

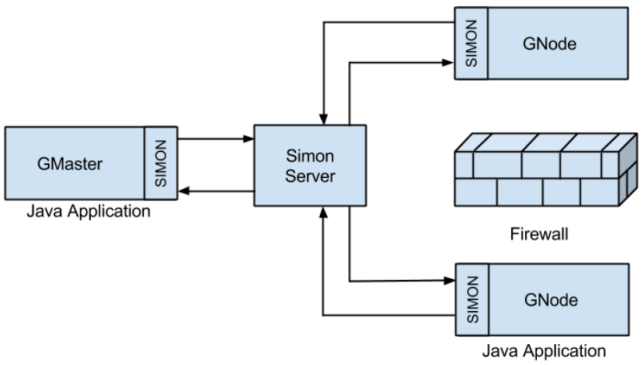


Abbildung 5: allgemeine Architektur [Ste14][S. 19]

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.

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

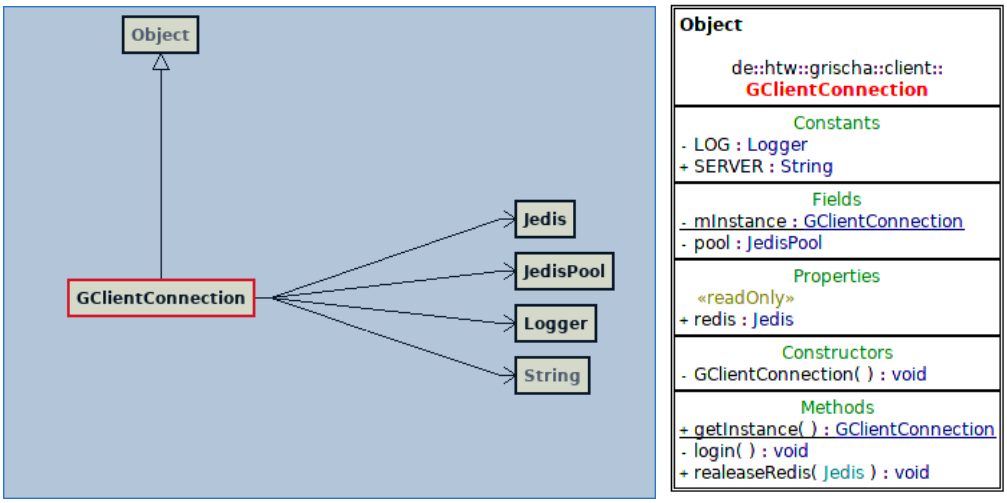
5.6 ZMQ

–TODO

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.



(a) UML (b) Klasse GclientConnection

Abbildung 6: Singleton GclientConnection die den Pool für Redis-Verbindugen bereitstellt

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

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